Developer's Guide

Borland®

Kylix™ 3

Delphi™ and C++ for Linux®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249

Printed in the U.S.A.

HDE7030WW21001 3E3R0502
02030405-9 8 7 6 5 4 3 2 1
D3

Contents

Tables

Figures

Chapter 1

Introduction

The *Developer's Guide* describes intermediate and advanced development topics, such as building client/server database applications, creating Internet Web server applications, writing custom components, and including support for industry-standard specifications such as SOAP and TCP/IP. Many of the advanced features that support Web development, advanced XML technologies, and database development require components or wizards that are not available in all editions of Kylix.

The *Developer's Guide* assumes you are familiar with using Linux and understand fundamental Delphi or C++ programming techniques. For an introduction to Kylix programming and the integrated development environment (IDE), see the *Quick Start* and the online Help.

In Kylix, Delphi refers to the Delphi programming language, previously called the Object Pascal programming language. The text and code examples for both the Delphi and C++ programming languages are combined so you can use either IDE.

What's in this manual?

This manual contains four parts, as follows:

Part I, "Programming with Kylix," describes how to build general-purpose Delphi and C++ applications. This part provides details on programming techniques you can use in any Kylix application. For example, it describes how to use common Component Library for Cross-Platform (CLX) objects that make user interface programming easy, such as handling strings and manipulating text. This section includes chapters on working with graphics, error and exception handling, using shared objects, writing cross-platform and international applications, and deploying your applications.

While it rarely matters that Kylix's underlying CLX is written in Delphi, there are a few instances where it affects your C++ programs. The chapter on C++ language support and CLX describes language issues such as how C++ class instantiation differs when using CLX classes and the C++ language extensions added to support the Kylix “component-property-event” model of programming.

- **Part II, “Developing database applications,”** describes how to build database applications using database tools and components. You can access several SQL server databases using the data access mechanism, dbExpress. To implement the more advanced database applications, you need the Kylix features that are not available in all editions. Your edition of Kylix comes with a set of dbExpress drivers for connecting to specific databases. Additional dbExpress drivers for connecting to other databases are available for purchase separately.
- **Part III, “Writing Internet applications,”** describes how to create applications that are distributed over the Internet. Kylix includes a wide array of tools for writing Web server applications, including: Web Broker, an architecture with which you can create cross-platform server applications; WebSnap, with which you can design Web pages in a GUI environment; support for working with XML documents; and BizSnap, an architecture for using SOAP-based Web Services. For lower-level support that underlies much of the messaging in Internet applications, this section also describes how to work with socket components.
- **Part IV, “Creating custom components,”** describes how to design and implement your own components, and how to make them available on the Component palette of the IDE. A component can be almost any program element that you want to manipulate at design time. Implementing custom components entails deriving a new class from an existing class type in the CLX class library.

Manual conventions

This manual uses the typefaces and symbols described in Table 1.1 to indicate special text.

Table 1.1 Typefaces and symbols

Typeface or symbol	Meaning
`Monospace type`	Monospaced text represents text as it appears on screen or in code. It also represents anything you must type.
[ ]	Square brackets in text or syntax listings enclose optional items. Text of this sort should not be typed verbatim.
Boldface	Boldfaced words in text or code listings represent reserved words or compiler options.
Italics	Italicized words in text represent Delphi or C++ identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
Keycaps	This typeface indicates a key on your keyboard. For example, “Press Esc to exit a menu.”
D	The Delphi printed icon represents Delphi programming language text and code examples.
C++	The C++ printed icon represents C++ programming language text and code examples.

Developer support services

Borland offers a variety of support options, including free services on the Internet, where you can search our extensive information base and connect with other users of Borland products, technical support, and fee-based consultant-level support.

For more information about Borland's developer support services, please see our Web site at http://www.borland.com/devsupport/kylix, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064. For customers outside of the United States of America, see our Web site at http://www.borland.com/bww.

When contacting support, be prepared to provide complete information about your environment, the version and edition of the product you are using, and a detailed description of the problem.

Part

I

Programming with Kylix

The chapters in “Programming with Kylix” introduce concepts and skills necessary for creating basic applications using any edition.

Chapter

2

Developing applications with Kylix

Borland Kylix is an object-oriented, visual programming environment to develop 32-bit applications for deployment on Windows and Linux. Using Kylix, you can create highly efficient applications with a minimum of manual coding.

Kylix provides a comprehensive class library called the Borland Component Library for Cross-Platform (CLX) and a suite of Rapid Application Development (RAD) design tools, including programming wizards and application and form templates.

- CLX includes objects that encapsulate the Qt library, and is used to run cross-platform applications on Windows or Linux.

This chapter briefly describes the Kylix development environment and how it fits into the development life cycle. The rest of this manual provides technical details on developing general-purpose, database, Internet and Intranet applications, and writing your own components.

Integrated development environment

When you start Kylix, you can either start Kylix for Delphi or Kylix for C++. You are immediately placed within the integrated development environment, also called the IDE. Either IDE provides all the tools you need to design, develop, test, debug, and deploy applications, allowing rapid prototyping and a shorter development time.

The IDE includes all the tools necessary to start designing applications, such as the:

- Form Designer, or *form*, a blank window on which to design the user interface for your application.
- Component palette for displaying visual and nonvisual components you can use to design your user interface.
- Object Inspector for examining and changing an object’s properties and events.
- Object TreeView for displaying and changing a components’ logical relationships.
- Code editor for writing and editing the underlying program logic.
- Project Manager for managing the files that make up one or more projects.

- Integrated debugger for finding and fixing errors in your code.
- Many other tools such as property editors to change the values for an object's property.
- Command-line tools including compilers, linkers, and other utilities.
- Extensive class libraries with many reusable objects. Many of the objects provided in the class library are accessible in the IDE from the Component palette. By convention, the names of objects in the class library begin with a T, such as *TStatusBar*. Names of objects that begin with a Q are based on the Qt library.

Some tools may not be included in all editions of the product.

You can write code in either the Delphi or C++ programming languages.

A more complete overview of the development environment is presented in the *Quick Start* manual included with the product. In addition, the online Help system provides help on all menus, dialog boxes, and windows.

Designing applications

You can use Kylix to design any kind of 32-bit application—from general-purpose utilities to sophisticated data access programs or distributed applications.

As you visually design the user interface for your application, the Code editor generates the underlying Delphi or C++ code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment.

In Kylix, you can create your own components. Most of the components provided are written in Delphi. You can add components that you write to the Component palette and customize the palette for your use by including new tabs if needed.

You can also use Kylix to design applications that run on both Windows and Linux by using CLX. CLX contains a set of classes that allows your program to port between Windows and Linux. Refer to Chapter 15, "Developing cross-platform applications," for details about cross-platform programming and the differences between the Windows and Linux environments.

Chapter 8, "Building applications and shared objects," introduces Kylix's support for different types of applications.

Creating projects

All of Kylix's application development revolves around projects. When you create an application in Kylix you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code.

You can view the contents of a project in a project management tool called the Project Manager. The Project Manager lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools in Kylix.

At the top of the project hierarchy is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the Project Manager. Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

Project files, which describe individual projects, files, and associated options, have a .dpr (Delphi) or .bpr (C++) extension. Project files contain directions for building an application or shared object. When you add and remove files using the Project Manager, the project file is updated. You specify project options using a Project Options dialog which has tabs for various aspects of your project such as forms, application, and compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of a Kylix application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines.

If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the uses clause (Delphi) or an include statement (C++) of the project file. Kylix automatically handles this as you add units to a project.

When you compile a project, it does not matter where the files that make up the project reside. The compiler treats shared files the same as those created by the project itself.

Editing code

The Kylix Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the Object Inspector. But other programming tasks, such as writing event handlers for objects, must be done by typing the code.

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changes and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor, and continue adjusting the form from there.

The Kylix code generation and property streaming systems are completely open to inspection. The source code for everything that is included in your final executable file—all of the CLX objects, RTL sources, and project files—can be viewed and edited in the Code editor.

Compiling applications

When you have finished designing your application interface on the form and writing additional code so it does what you want, you can compile the project from the IDE or from the command line.

All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to build when you've changed global compiler directives to ensure that all code compiles in the proper state.You can also test the validity of your source code without attempting to compile the project.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles those changed modules and relinks your application.

If you have grouped several projects together, you can compile or build all projects in a single project group at once. Choose Project|Compile All Projects or Project|Build All Projects with the project group selected in the Project Manager.

Debugging applications

Kylix provides an integrated debugger that helps you find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging.

The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the variable values, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed. The debugger is described in online Help.

You can also use exception handling to recognize, locate, and deal with errors. Exceptions in Kylix are classes, like other classes in Kylix, except, by convention, they begin with an initial E rather than a T. See Chapter 13, “Exception handling” for details on exception handling.

Deploying applications

Kylix includes add-on tools to help with application deployment. For example, InstallShield Express (not available in all editions) helps you to create an installation package for your application that includes all of the files needed for running a distributed application. TeamSource software (not available in all editions) is also available for tracking application updates.

Note Not all editions of Kylix have deployment capabilities.

Refer to Chapter 18, “Deploying applications,” for specific information on deployment.

Chapter

3

Using the class libraries

This chapter presents an overview of the class libraries and introduces some of the components that you can use while developing applications. The class libraries are collectively called CLX (Component Library for Cross-Platform). The hierarchy is extensive, containing both components that you can work with in the IDE and classes that you create and use in runtime code.

Understanding the class libraries

CLX is a class library made up of objects that you use when developing applications. It is composed of several sublibraries, each of which serves a different purpose. These sublibraries are listed in Table 3.1:

Table 3.1 CLX libraries

Part	Description
BaseCLX	Low-level classes and routines available for all CLX applications. BaseCLX includes the CLX Runtime Library up to and including the Classes unit.
DataCLX	Client data-access components. These components are used in applications that access databases. They can access data from a file on disk or from a database server using dbExpress.
NetCLX	Components for building Web Server applications. These include support for applications that use Apache or CGI Web Servers.
VisualCLX	GUI components and graphics classes. VisualCLX classes make use of an underlying widget library (Qt).

All CLX classes descend from *TObject*. *TObject* introduces methods that implement fundamental behavior like construction, destruction, and message handling.

Components are a subset of CLX that descend from the class *TComponent*. You can place components on a form or data module and manipulate them at design time.

Using the Object Inspector, you can assign property values without writing code. Most components are either visual or nonvisual, depending on whether they are visible at runtime. Some components appear on the Component palette.

Visual components, such as *TForm* and *TSpeedButton*, are called *controls* and descend from *TControl*. Controls are used in GUI applications, and appear to the user at runtime. *TControl* provides properties that specify the visual attributes of controls, such as their height and width.

Nonvisual components are used for a variety of tasks. For example, if you are writing an application that connects to a database, you can place a *TDataSource* component on a form to connect a control and a dataset used by the control. This connection is not visible to the user, so *TDataSource* is nonvisual. At design time, nonvisual components are represented by an icon. This allows you to manipulate their properties and events just as you would a visual control.

Classes that are not components (that is, CLX classes that descend from *TObject* but not *TComponent*) are also used for a variety of tasks. Typically, these classes are used for accessing system objects (such as a file or the clipboard) or for transient tasks (such as storing data in a list). You can't create instances of these classes at design time, although they are sometimes created by the components that you add in the forms designer.

Detailed reference material on all CLX objects is accessible through online Help while you are programming. In the Code editor, place the cursor anywhere on the object and press F1 to display the Help topic.

Properties, methods, and events

CLX is a hierarchy of objects that are tied to the IDE, where you can develop applications quickly. The classes in CLX are based on properties, methods, and events. Each class includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events). CLX is written in the Delphi language, although it is available to C++ applications as well.

Properties

Properties are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the *Visible* property determines whether an object can be seen in an application interface. Well-designed properties make your components easier for others to use and easier for you to maintain.

Here are some of the useful features of properties:

- Unlike methods, which are only available at runtime, you can see and change some properties at design time and get immediate feedback as the components change in the IDE.
- You can access some properties in the Object Inspector, where you can modify the values of your object visually. Setting properties at design time is easier than writing code and makes your code easier to maintain.
- Because the data is encapsulated, it is protected and private to the actual object.

- The calls to get and set the values of properties can be methods, so special processing can be done that is invisible to the user of the object. For example, data could reside in a table, but could appear as a normal data member to the programmer.
- You can implement logic that triggers events or modifies other data during the access of a property. For example, changing the value of one property may require you to modify another. You can change the methods created for the property.
- Properties can be virtual.
- A property is not restricted to a single object. Changing one property on one object can affect several objects. For example, setting the *Checked* property on a radio button affects all of the radio buttons in the group.

Methods

A *method* is a function that is a member of a class. Methods define the behavior of an object. Methods can access all the *public*, *protected*, and *private* properties and data members of the class and are commonly referred to as member functions. (For dtails on public, protected, and private members, see "Controlling access" on page 36-4.) Although most methods belong to an instance of a class, some methods belong instead to the class type. In Delphi, these are called class methods, while in C++ they use the static keyword.

Events

An *event i*s an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. In a program, the programmer has no way of predicting the exact sequence of actions a user will perform. For example, the user may choose a menu item, click a button, or mark some text. You can write code to handle the events in which you are interested, rather than writing code that always executes in the same restricted order.

Regardless of how an event is triggered, CLX objects look to see if you have written any code to handle that event. If you have, that code is executed; otherwise, the default event handling behavior takes place.

The kinds of events that can occur can be divided into three main categories:

- User events
- System events
- Internal events

User events

User events are actions that the user initiates. Examples of user events are *OnClick* (the user clicked the mouse), *OnKeyPress* (the user pressed a key on the keyboard), and *OnDblClick* (the user double-clicked a mouse button).

System events

System events are events that the operating system fires for you. For example, the *OnTimer* event (which the Timer component issues whenever a predefined interval has elapsed), the *OnPaint* event (a component or window needs to be redrawn), and so on. Usually, system events are not directly initiated by a user action.

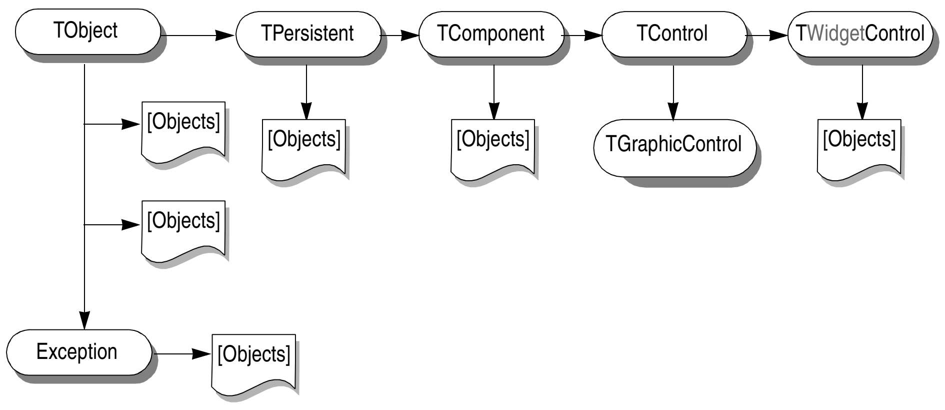
Internal events

Internal events are events that are generated by the objects in your application. An example of an internal event is the *OnPost* event that a dataset generates when your application tells it to post the current record.

Objects, components, and controls

Figure 3.1 is a greatly simplified view of the inheritance hierarchy that illustrates the relationship between objects, components, and controls.

Figure 3.1 A simplified hierarchy diagram



Every object (class) inherits from *TObject*. Objects that can appear in the forms designer inherit from *TPersistent* or *TComponent*. Controls, which appear to the user at runtime, inherit from *TControl*. There are two types of controls, graphic controls, which inherit from *TGraphicControl*, and widget controls, which inherit from *TWidgetControl*. A control like *TCheckBox* inherits all the functionality of *TObject*, *TPersistent*, *TComponent*, *TControl*, and *TWidgetControl*, and adds specialized capabilities of its own.

The figure shows several important base classes, which are described in the following table:

Table 3.2 Important base classes

Class	Description
TObject	Signifies the base class and ultimate ancestor of everything in CLX. *TObject* encapsulates the fundamental behavior common to all CLX objects by introducing methods that perform basic functions such as creating, maintaining, and destroying an instance of an object.
Exception	Specifies the base class of all classes that relate to CLX exceptions. *Exception* provides a consistent interface for error conditions, and enables applications to handle error conditions gracefully.
TPersistent	Specifies the base class for all objects that implement publishable properties. Classes under *TPersistent* deal with sending data to streams and allow for the assignment of classes.
TComponent	Specifies the base class for all components. Components can be added to the Component palette and manipulated at design time. Components can own other components.
TControl	Represents the base class for all controls that are visible at runtime. *TControl* is the common ancestor of all visual components and provides standard visual controls like position and cursor. This class also provides events that respond to mouse actions.
TWidgetControl	Specifies the base class of all controls that can have keyboard focus. Controls under *TWidgetControl* are called widgets.

The next few sections present a general description of the types of classes that each branch contains. For a complete overview of the CLX object hierarchy, refer to the CLX Object Hierarchy wall chart included with this product.

TObject branch

The *TObject* branch includes all CLX classes that descend from *TObject* but not from *TPersistent*. Much of the powerful capability of CLX is established by the methods that *TObject* introduces. *TObject* encapsulates the fundamental behavior common to all classes in CLX by introducing methods that provide:

- The ability to respond when object instances are created or destroyed.
- Class type and instance information on an object, and runtime type information (RTTI) about its published properties.
- Support for handling notifications.

TObject is the immediate ancestor of many simple classes. Classes in the *TObject* branch have one common, important characteristic: they are transitory. This means that these classes do not have a method to save the state that they are in prior to destruction; they are not persistent.

One of the main groups of classes in this branch is the *Exception* class. This class provides a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions.

Another group in the *TObject* branch is classes that encapsulate data structures, such as:

- *TBits,* a class that stores an "array" of Boolean values.
- *TList,* a linked list class.
- *TStack,* a class that maintains a last-in first-out array of pointers.
- *TQueue,* a class that maintains a first-in first-out array of pointers.

Another group in the *TObject* branch are wrappers for external objects like *TPrinter*, which encapsulates a printer interface, and *TIniFile,* which lets a program read from or write to an ini file.

TStream is a good example of another type of class in this branch. *TStream* is the base class type for stream objects that can read from or write to various kinds of storage media, such as disk files, dynamic memory, and so on. (see "Using streams" on page 5-2 for information on streams)

See Chapter 5, "Using BaseCLX," for information on many of the classes in the *TObject* branch (as well as on many global routines in the CLX Runtime Library).

TPersistent branch

The *TPersistent* branch includes all CLX classes that descend from *TPersistent* but not from *TComponent*. Persistence determines what gets saved with a form file or data module and what gets loaded into the form or data module when it is retrieved from memory.

Because of their persistence, objects from this branch can appear at design time. However, they can't exist independantly. Rather, they implement properties for components. Properties are only loaded and saved with a form if they have an owner. The owner must be some component. *TPersistent* introduces the *GetOwner* method, which lets the form designer determine the owner of the object.

Classes in this branch are also the first to include a published section where properties can be automatically loaded and saved. A *DefineProperties* method lets each class indicate how to load and save properties.

Following are some of the classes in the *TPersistent* branch of the hierarchy:

- Graphics objects such as: *TBrush*, *TFont*, and *TPen*.
- Classes such as *TBitmap* and *TIcon*, which store and display visual images.
- String lists, such as *TStringList*, which represent text or lists of strings that can be assigned at design time.
- *TClipboard,* a class that contains text or graphics that have been cut or copied from an application.
- Collections and collection items, which descend from *TCollection* or *TCollectionItem.* These classes maintain indexed collections of specially defined items that belong to a component. Examples include *THeaderSections* and *THeaderSection* or *TListColumns* and *TListColumn*.

TComponent branch

The *TComponent* branch contains classes that descend from *TComponent* but not *TControl*. Objects in this branch are components that you can manipulate on forms at design time but which do not appear to the user at runtime. They are persistent objects that can do the following:

- Appear on the Component palette and be changed in the form designer.
- Own and manage other components.
- Load and save themselves.

Several methods introduced by *TComponent* dictate how components act during design time and what information gets saved with the component. Streaming (the saving and loading of form files, which store information about the property values of objects on a form) is introduced in this branch. Properties are persistent if they are published and published properties are automatically streamed.

The *TComponent* branch also introduces the concept of ownership that is propagated throughout CLX. Two properties support ownership: *Owner* and *Components*. Every component has an *Owner* property that references another component as its owner. A component may own other components. In this case, all owned components are referenced in the component's *Components* property.

The constructor for every component takes a parameter that specifies the new component's owner. If the passed-in owner exists, the new component is added to that owner's *Components* list. Aside from using the *Components* list to reference owned components, this property also provides for the automatic destruction of owned components. As long as the component has an owner, it will be destroyed when the owner is destroyed. For example, since *TForm* is a descendant of *TComponent*, all components owned by a form are destroyed and their memory freed when the form is destroyed. (Assuming, of course, that the components have properly designed destructors that clean them up correctly.)

If a property type is a *TComponent* or a descendant, the streaming system creates an instance of that type when reading it in. If a property type is *TPersistent* but not *TComponent*, the streaming system uses the existing instance available through the property and reads values for that instance's properties.

Some of the classes in the *TComponent* branch include:

- *TActionList,* a class that maintains a list of actions, which provides an abstraction of the responses your program can make to user input.
- *TMainMenu,* a class that provides a menu bar and its accompanying drop-down menus for a form.
- *TOpenDialog*, *TSaveDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog*, and so on, classes that display and gather information from commonly used dialog boxes.
- *TScreen,* a class that keeps track of the forms and data modules that an application creates, the active form, the active control within that form, the size and resolution of the screen, and the cursors and fonts available for the application to use.

Components that do not need a visual interface can be derived directly from *TComponent*. To make a tool such as a *TTimer* device, you can derive from *TComponent*. This type of component resides on the Component palette but performs internal functions that are accessed through code rather than appearing in the user interface at runtime.

See Chapter 6, "Working with components," for details on setting properties, calling methods, and working with events for components.

TControl branch

The *TControl* branch consists of components that descend from *TControl* but not *TWidgetControl*. Classes in this branch are controls: visual objects that the user can see and manipulate at runtime. All controls have properties, methods, and events in common that relate to how the control looks, such as its position, the cursor associated with the control's window, methods to paint or move the control, and events to respond to mouse actions. Controls in this branch, however, can never receive keyboard input.

Whereas *TComponent* defines behavior for all components, *TControl* defines behavior for all visual controls. This includes drawing routines, standard events, and containership.

TControl introduces many visual properties that all controls inherit. These include the *Caption*, *Color*, *Font*, and *HelpContext* or *HelpKeyword*. While these properties inherited from *TControl*, they are only published—and hence appear in the Object Inspector—for controls to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays. *TControl* also introduces the *Parent* property, which specifies another control that visually contains the control.

Classes in the *TControl* branch often called graphic controls, because they all descend from *TGraphicControl*, which is an immediate descendant of *TControl*. Although these controls appear to the user at runtime, graphic controls do not have their own underlying widget. Instead, they use their parent's widget. It is because of this limitation that graphic controls cant receive keyboard input or act as a parent to other controls. However, because they do not have their own widget, graphic controls use fewer system resources. For details on many of the classes in the *TControl* branch, see "Graphic controls" on page 10-15.

See Chapter 7, "Working with controls," for details on how to interact with controls at runtime.

TWidgetControl branch

Most controls fall into the *TWidgetControl* branch. Unlike graphic controls, controls in this branch have their own associated widget. Because of this, they are sometimes called widget controls. Widget controls all descend from *TWidgetControl*.

Controls in the *TWidgetControl* branch:

- Can receive focus while an application is running, which means they can receive keyboard input from the application user. In comparison, graphic controls can only display data and respond to the mouse.

- Can be the parent of one or more child controls.
- Have a handle, or unique identifier, that allows them to access the underlying widget.

The *TWidgetControl* branch includes both controls that are drawn automatically (such as *TEdit*, *TListBox*, *TComboBox*, *TPageControl*, and so on) and custom controls that do not correspond directly to a single underlying widget. Controls in this latter category, which includes classes like *TStringGrid* and *TDBNavigator*, must handle the details of painting themselves. Because of this, they descend from *TCustomControl*, which introduces a *Canvas* property on which they can paint themselves.

For details on many of the controls in the *TWidgetControl* branch, see Chapter 10, "Types of controls,".

Chapter

4

Using the object model in Delphi programming

The Delphi language is a set of object-oriented extensions to standard Pascal. Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you define a class, you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

This chapter is a brief introduction of object-oriented concepts for programmers who are just starting out with the Delphi language. Most of this material should be familiar if you already know an object-oriented language such as C++. For more details on object-oriented programming for programmers who want to write components that can be installed on the Component palette, see Chapter 35, "Overview of component creation."

Note This chapter only addresses object-oriented programming in Delphi. It is assumed that C++ programmers are already familiar with object-oriented concepts. (Programmers who are starting out with C++ are advised to use one of the many available texts on the C++ language.) However, the C++ programmer may want to look over this chapter for information on how classes interact with the IDE, and to gain insight into some of the peculiarities of the Delphi language, in which CLX is written. For details on Borland extensions to C++ that support its use for CLX, see Chapter 14, "C++ language support for CLX."

What is an object?

A *class* is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements. An *object* is an instance of a class. That is, it is a value whose type is a class. The term object is often used more loosely in this documentation and where the distinction between a class and an instance of the class is not important, the term "object" may also refer to a class.

You can begin to understand objects if you understand Pascal *records* or *structures* in C. Records are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects—unlike records—contain procedures and functions that operate on their data. These procedures and functions are called *methods*.

An object's data elements are accessed through *properties*. The properties of many Delphi objects have values that you can change at design time without writing code. If you want a property value to change at runtime, you need to write only a small amount of code.

The combination of data and functionality in a single unit is called *encapsulation*. In addition to encapsulation, object-oriented programming is characterized by *inheritance* and *polymorphism*. Inheritance means that objects derive functionality from other objects (called *ancestors*); objects can modify their inherited behavior. Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably.

Examining a Delphi object

When you create a new project, the IDE displays a new form for you to customize. In the Code editor, the automatically generated unit declares a new class type for the form and includes the code that creates the new form instance. The generated code looks like this:

```
unit Unit1;
interface

uses SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs;

type
  TForm1 = class(TForm){ The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end;{ The type declaration of the form ends here }

var
  Form1: TForm1;

implementation{ Beginning of implementation part }
{$R *.xfm}
end.{ End of implementation part and unit}
```

The new class type is *TForm1*, and it is derived from type *TForm*, which is also a class.

A class is like a record in that they both contain data fields, but a class also contains methods—code that acts on the object's data. So far, *TForm1* appears to contain no fields or methods, because you haven't added any components (the fields of the new object) to the form and you haven't created any event handlers (the methods of the new object). *TForm1* does contain inherited fields and methods, even though you don't see them in the type declaration.

This variable declaration declares a variable named *Form1* of the new type *TForm1*.

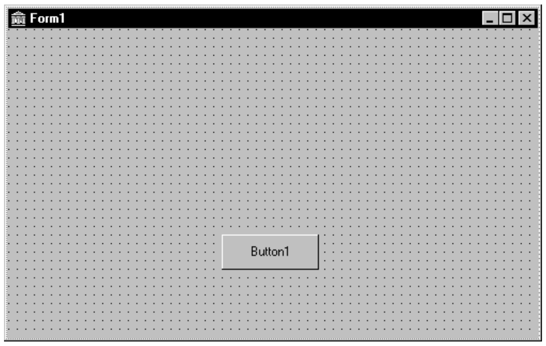
```
var
  Form1: TForm1;
```

Form1 represents an instance, or object, of the class type *TForm1*. You can declare more than one instance of a class type; you might want to do this, for example, to create multiple child windows in a Multiple Document Interface (MDI) application. Each instance maintains its own data, but all instances use the same code to execute methods.

Although you haven't added any components to the form or written any code, you already have a complete GUI application that you can compile and run. All it does is display a blank form.

Suppose you add a button component to this form and write an *OnClick* event handler that changes the color of the form when the user clicks the button. The result might look like this:

Figure 4.1 A simple form



When the user clicks the button, the form's color changes to green. This is the event-handler code for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

Objects can contain other objects as data fields. Each time you place a component on a form, a new field appears in the form's type declaration. If you create the application described above and look at the code in the Code editor, this is what you see:

```
unit Unit1;

interface

uses SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs;
```

```
type
  TForm1 = class(TForm)
    Button1: TButton;{ New data field }
    procedure Button1Click(Sender: TObject);{ New method declaration }
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.Button1Click(Sender: TObject);{ The code of the new method }
begin
  Form1.Color := clGreen;
end;

end.
```

TForm1 has a *Button1* field that corresponds to the button you added to the form. *TButton* is a class type, so *Button1* refers to an object.

All the event handlers you write using the IDE are methods of the form object. Each time you create an event handler, a method is declared in the form object type. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared in the *TForm1* type declaration. The code that implements the *Button1Click* method appears in the implementation part of the unit.

Changing the name of a component

You should always use the Object Inspector to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorWindow*. When you change the form's *Name* property in the Object Inspector, the new name is automatically reflected in the form's .xfm file (which you usually don't edit manually) and in the source code that the IDE generates:

```
unit Unit1;

interface

uses SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs;

type
  TColorWindow = class(TForm){ Changed from TForm1 to TColorWindow }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  ColorWindow: TColorWindow;{ Changed from Form1 to ColorWindow }

implementation

{$R *.xfm}

procedure TColorWindow.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;{ The reference to Form1 didn't change! }
end;

end.
```

Note that the code in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form:

```
procedure TColorWindow.Button1Click(Sender: TObject);
begin
  ColorWindow.Color := clGreen;
end;
```

Inheriting data and code from an object

The *TForm1* object seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of the component *TForm*. When you add a new form to your project, you start with *TForm* and customize it by adding components, changing property values, and writing event handlers. To customize any object, you first derive a new object from the existing one; when you add a new form to your project, the IDE automatically derives a new form from the *TForm* type:

```
TForm1 = class(TForm)
```

A derived class inherits all the properties, events, and methods of the class from which it derives. The derived class is called a *descendant* and the class from which it derives is called an *ancestor*. If you look up *TForm* in the online Help, you'll see lists of its properties, events, and methods, including the ones that *TForm* inherits from *its* ancestors. A Delphi clss can have only one immediate ancestor, but it can have many direct descendants.

Scope and qualifiers

Scope determines the accessibility of an object's fields, properties, and methods. All members declared in a class are available to that class and, as is discussed later, often to its descendants. Although a method's implementation code appears outside of the class declaration, the method is still within the scope of the class because it is declared in the class declaration.

When you write code to implement a method that refers to properties, methods, or fields of the class where the method is declared, you don't need to preface those identifiers with the name of the class. For example, if you put a button on a new form, you could write this event handler for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

The first statement is equivalent to

```
Form1.Color := clFuchsia
```

You don't need to qualify *Color* with *Form1* because the *Button1Click* method is part of *TForm1*; identifiers in the method body therefore fall within the scope of the *TForm1* instance where the method is called. The second statement, in contrast, refers to the color of the button object (not of the form where the event handler is declared), so it requires qualification.

The IDE creates a separate unit (source code) file for each form. If you want to access one form's components from another form's unit file, you need to qualify the component names, like this:

```
Form2.Edit1.Color := clLime;
```

In the same way, you can access a component's methods from another form. For example,

```
Form2.Edit1.Clear;
```

To access *Form2*'s components from *Form1*'s unit file, you must also add *Form2*'s unit to the uses clause of *Form1*'s unit.

The scope of a class extends to its descendants. You can, however, redeclare a field, property, or method in a descendant class. Such redeclarations either hide or override the inherited member.

For more information about scope, inheritance, and the uses clause, see the *Delphi Language Guide*.

Private, protected, public, and published declarations

A class type declaration contains three or four possible sections that control the accessibility of its fields and methods:

```
Type
  TClassName = Class(TObject)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

- The public section declares fields and methods with no access restrictions. Class instances and descendant classes can access these fields and methods. A public member is accessible from wherever the class it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.
- The protected section includes fields and methods with some access restrictions. A protected member is accessible within the unit where its class is declared and by any descendant class, regardless of the descendant class's unit.
- The private section declares fields and methods that have rigorous access restrictions. A private member is accessible only within the unit where it is declared. Private members are often used in a class to implement other (public or published) methods and properties.
- For classes that descend from *TPersistent*, a published section declares properties and events that are available at design time. A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the Object Inspector at design time.

When you declare a field, property, or method, the new member is added to one of these four sections, which gives it its *visibility*: private, protected, public, or published.

For more information about visibility, see the *Delphi Language Guide*.

Using object variables

You can assign one object variable to another object variable if the variables are of the same type or are assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable to which you are assigning is an ancestor of the type of the variable being assigned. For example, here is a *TSimpleForm* type declaration and a variable declaration section declaring two variables, *AForm* and *Simple*:

```
type
  TSimpleForm = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AForm: TForm;
  SimpleForm: TSimpleForm;
```

AForm is of type *TForm*, and *SimpleForm* is of type *TSimpleForm*. Because *TSimpleForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := SimpleForm;
```

Suppose you write an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called. Each event handler has a *Sender* parameter of type *TObject*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
⋮
end;
```

Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. The value of *Sender* is always the control or component that responds to the event. You can test *Sender* to find the type of component or control that called the event handler using the reserved word is. For example,

```
if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;
```

Creating, instantiating, and destroying objects

Many of the objects you use in the forms designer, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and datasource components, have no visual representation at runtime.

You may want to create your own classes. For example, you could create a *TEmployee* class that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Double;
  end;
```

In addition to the fields, properties, and methods you've defined, *TEmployee* inherits all the methods of *TObject*. You can place a type declaration like this one in either the interface or implementation part of a unit, and then create instances of the new class by calling the *Create* method that *TEmployee* inherits from *TObject*:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

The *Create* method is called a *constructor*. It allocates memory for a new instance object and returns a reference to the object.

Components on a form are created and destroyed automatically. However, if you write your own code to instantiate objects, you are responsible for disposing of them as well. Every object inherits a *Destroy* method (called a *destructor*) from *TObject*. To destroy an object, however, you should call the *Free* method (also inherited from *TObject*), because *Free* checks for a nil reference before calling *Destroy*. For example,

```
Employee.Free;
```

destroys the *Employee* object and deallocates its memory.

Components and ownership

Delphi components have a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's memory is freed. The owner of a component—the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed.

Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

Defining new classes

Although there are many classes in the object hierarchy, you are likely to need to create additional classes if you are writing object-oriented programs. The classes you write must descend from *TObject* or one of its descendants.

The advantage of using classes comes from being able to create new classes as descendants of existing ones. Each descendant class inherits the fields and methods of its parent and ancestor classes. You can also declare methods in the new class that override inherited ones, introducing new, more specialized behavior.

The general syntax of a descendant class is as follows:

```
Type
  TClassName = Class (TParentClass)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

If no parent class name is specified, the class inherits directly from *TObject*. *TObject* defines only a handful of methods, including a basic constructor and destructor.

To define a class:

1 In the IDE, start with a project open and choose File | New | Unit to create a new unit where you can define the new class.

2 Add the uses clause and type section to the interface section.

3 In the type section, write the class declaration. You need to declare all the member variables, properties, methods, and events.

```
TMyClass = class; {This implicitly descends from TObject}
public
  .
  .
  .
  .
  .
  .
private
  .
  .
  .
published {If descended from TPersistent or below}
  .
  .
  .
```

If you want the class to descend from a specific class, you need to indicate that class in the definition:

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

For example:

```
type TMyButton = class(TButton)
  property Size: Integer;
  procedure DoSomething;
end;
```

4 Some versions of the IDE include a feature called class completion that simplifies the work of defining and implementing new classes by generating skeleton code for the class members you declare. If you have code completion, invoke it to finish the class declaration: place the cursor within a method definition in the interface section and press Ctrl+Shift+C (or right-click and select Complete Class at Cursor). Any unfinished property declarations are completed, and for any methods that require an implementation, empty methods are added to the implementation section.

If you do not have class completion, you need to write the code yourself, completing property declarations and writing the methods.

Given the example above, if you have class completion, read and write specifiers are added to your declaration, including any supporting fields or methods:

```
type TMyButton = class(TButton)
  property Size: Integer read FSize write SetSize;
  procedure DoSomething;
private
  FSize: Integer;
  procedure SetSize(const Value: Integer);
```

The following code is also added to the implementation section of the unit.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin

end;
procedure TMyButton.SetSize(const Value: Integer);
begin
   FSize := Value;
end;
```

5 Fill in the methods. For example, to make it so the button beeps when you call the DoSomething method, add the Beep between begin and end.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
  Beep;
end;

procedure TMyButton.SetSize(const Value: Integer);
begin
  if fsize < > value then
  begin
    FSize := Value;
    DoSomething;
  end;
end;
```

Note that the button also beeps when you call SetSize to change the size of the button.

For more information about the syntax, language definitions, and rules for classes, see the *Delphi Language Guide*.

Using interfaces in Delphi

Delphi is a single-inheritance language. That means that any class has only a single direct ancestor. However, there are times you want a new class to inherit properties and methods from more than one base class so that you can use it sometimes like one and sometimes like the other. Interfaces let you achieve something like this effect.

An interface is like a class that contains only abstract methods (methods with no implementation) and a clear definition of their functionality. Interface method definitions include the number and types of their parameters, their return type, and their expected behavior. By convention, interfaces are named according to their behavior and prefaced with a capital *I*. For example, an *IMalloc* interface would allocate, free, and manage memory. Similarly, an *IPersist* interface could be used as a general base interface for descendants, each of which defines specific method prototypes for loading and saving the state of an object to a storage, stream, or file.

An interface has the following syntax:

```
IMyObject = interface
  procedure MyProcedure;
end;
```

A simple example of an interface declaration is:

```
type
IEdit = interface
  procedure Copy;
  procedure Cut;
  procedure Paste;
  function Undo: Boolean;
end;
```

Interfaces can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, define a class that declares the interface in its ancestor list, indicating that it will implement all of the methods of that interface:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy;
  procedure Cut;
  procedure Paste;
  function Undo: Boolean;
end;
```

While interfaces define the behavior and signature of their methods, they do not define the implementations. As long as the class's implementation conforms to the interface definition, the interface is fully polymorphic, meaning that accessing and using the interface is the same for any implementation of it.

For more details about the syntax, language definitions and rules for interfaces, see the *Delphi Language Guide*

Using interfaces across the hierarchy

Using interfaces lets you separate the way a class is used from the way it is implemented. Two classes can implement the same interface without descending from the same base class. By obtaining an interface from either class, you can call the same methods without having to know the type of the class. This polymorphic use of the same methods on unrelated objects is possible because the objects implement the same interface. For example, consider the interface,

```
IPaint = interface
  procedure Paint;
```

```
end;
```

and the two classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Whether or not the two classes share a common ancestor, they are still assignment compatible with a variable of *IPaint* as in

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

This could have been accomplished by having *TCircle* and *TSquare* descend from a common ancestor (say, *TFigure*), which declares a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. In the previous example, *IPaint* could be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

IRotate makes sense for the rectangle but not the circle. The classes would look like

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Later, you could create a class *TFilledCircle* that implements the *IRotate* interface to allow rotation of a pattern that fills the circle without having to add rotation to the simple circle.

Note For these examples, the immediate base class or an ancestor class is assumed to have implemented the methods of *IInterface*, the base interface from which all interfaces descend. For more information on *IInterface*, see "Implementing IInterface" on page 4-14 and "Memory management of interface objects" on page 4-18.

Using interfaces with procedures

Interfaces allow you to write generic procedures that can handle objects without requiring that the objects descend from a particular base class. Using the *IPaint* and *IRotate* interfaces defined previously, you can write the following procedures,

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

RotateObjects does not require that the objects know how to paint themselves and *PaintObjects* does not require the objects know how to rotate. This allows the generic procedures to be used more often than if they were written to only work against a *TFigure* class.

Implementing IInterface

Just as all objects descend, directly or indirectly, from *TObject*, all interfaces derive from the *IInterface* interface. *IInterface* provides for dynamic querying and lifetime management of the interface. This is established in the three *IInterface* methods:

- *QueryInterface* dynamically queries a given object to obtain interface references for the interfaces that the object supports.
- *_AddRef* is a reference counting method that increments the count each time a call to *QueryInterface* succeeds. While the reference count is nonzero the object must remain in memory.
- *_Release* is used with *_AddRef* to allow an object to track its own lifetime and determine when it is safe to delete itself. Once the reference count reaches zero, the object is freed from memory.

Every class that implements interfaces must implement the three *IInterface* methods, as well as all of the methods declared by any other ancestor interfaces, and all of the methods declared by the interface itself. You can, however, inherit the implementations of methods of interfaces declared in your class.

By implementing these methods yourself, you can provide an alternative means of lifetime management, disabling reference-counting. This is a powerful technique that lets you decouple interfaces from reference-counting.

TInterfacedObject

When defining a class that supports one or more interfaces, it is convenient to use *TInterfacedObject* as a base class because it implements the methods of *IInterface*. *TInterfacedObject* class is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendant of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```
type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;
```

Because it implements the methods of *IInterface*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see "Memory management of interface objects" on page 4-18, which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

Using the as operator with interfaces

Classes that implement interfaces can use the as operator for dynamic binding on the interface. In the following example:

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;

begin
  X := P as IPaint;
{ statements }
end;
```

the variable *P* of type *TInterfacedObject*, can be assigned to the variable *X*, which is an *IPaint* interface reference. Dynamic binding makes this assignment possible. For this assignment, the compiler generates code to call the *QueryInterface* method of *P*'s *IInterface* interface. This is because the compiler cannot tell from *P*'s declared type whether *P*'s instance actually supports *IPaint*. At runtime, *P* either resolves to an

IPaint reference or an exception is raised. In either case, assigning *P* to *X* will not generate a compile-time error as it would if *P* was of a class type that did not implement *IInterface*.

When you use the as operator for dynamic binding on an interface, you should be aware of the following requirements:

- Explicitly declaring *IInterface*: Although all interfaces derive from *IInterface*, it is not sufficient, if you want to use the as operator, for a class to simply implement the methods of *IInterface*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IInterface* in its interface list.
- Using an IID: Interfaces can use an identifier that is based on a GUID (globally unique identifier). GUIDs that are used to identify interfaces are referred to as interface identifiers (IIDs). If you are using the as operator with an interface, it must have an associated IID. To create a new GUID in your source code you can use the *Ctrl+Shift+G* editor shortcut key.

Reusing code and delegation

One approach to reusing code with interfaces is to have one interfaced object contain, or be contained by another. Using properties that are object types provides an approach to containment and code reuse. To support this design for interfaces, the Delphi language has a keyword implements, that makes if easy to write code to delegate all or part of the implementation of an interface to a subobject.

Aggregation is another way of reusing code through containment and delegation. In aggregation, an outer object uses an inner object that implements interfaces which are exposed only by the outer object.

Using implements for delegation

Many classes have properties that are subobjects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface) you can use the keyword implements to specify that the methods of that interface are delegated to the object or interface reference which is the value of the property. The delegate only needs to provide implementation for the methods. It does not have to declare the interface support. The class containing the property must include the interface in its ancestor list.

By default, using the implements keyword delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods to override this default behavior.

The following example uses the implements keyword in the design of a color adapter object that converts an 8-bit RGB color value to a *Color* reference:

```
unit cadapt;

type
IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;

ICColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;

{ TRGB8ColorRefAdapter   map an IRGB8bit to an IColorRef }
  TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

For more information about the syntax, implementation details, and language rules of the implements keyword, see the *Delphi Language Guide*.

Aggregation

Aggregation offers a modular approach to code reuse through sub-objects that make up the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. At a minimum, it must implement *IInterface*. The inner object, or objects, also implement one or more interfaces. However, only the outer object exposes the interfaces. That is, the outer object exposes both the interfaces it implements and the ones that its contained objects implement.

Clients know nothing about inner objects. While the outer object provides access to the inner object interfaces, their implementation is completely transparent. Therefore, the outer object class can exchange the inner object class type for any class that implements the same interface. Correspondingly, the code for the inner object classes can be shared by other classes that want to use it.

The aggregation model defines explicit rules for implementing *IInterface* using delegation. The inner object must implement two versions of the *IInterface* methods.

- It must implement *IInterface* on itself, controlling its own reference count. This implementation of *IInterface* tracks the relationship between the outer and the inner object. For example, when an object of its type (the inner object) is created, the creation succeeds only for a requested interface of type *IInterface*.
- It also implements a second *IInterface* for all the interfaces it implements that the outer object exposes. This second *IInterface* delegates calls to *QueryInterface*, *_AddRef*, and *_Release* to the outer object. The outer *IInterface* is referred to as the "controlling Unknown."

Memory management of interface objects

One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *_AddRef* and *_Release* methods of *IInterface* provide a way to implement this lifetime management. *_AddRef* and *_Release* track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero. This lifetime management model is optional, but a convention for most interfaced objects.

Using reference counting

The Delphi compiler provides most of the *IInterface* memory management for you by its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from *TInterfacedObject*. If you decide to use reference counting, then you must be careful to only hold the object as an interface reference, and to be consistent in your reference counting. For example:

```
procedure beep(x: ITest);

function test_func()
var
  y: ITest;
begin
  y := TTest.Create; // because y is of type ITest, the reference count is one
  beep(y); // the act of calling the beep function increments the reference count
  // and then decrements it when it returns
  y.something; // object is still here with a reference count of one
end;
```

This is the cleanest and safest approach to memory management; and if you use *TInterfacedObject* it is handled automatically. If you do not follow this rule, your object can unexpectedly disappear, as demonstrated in the following code:

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // no count on the object yet
  beep(x as ITest); // count is incremented by the act of calling beep
  // and decremented when it returns
  x.something; // surprise, the object is gone
end;
```

Note In the examples above, the *beep* procedure, as it is declared, increments the reference count (call *_AddRef*) on the parameter, whereas either of the following declarations do not:

```
procedure beep(const x: ITest);
```

or

```
procedure beep(var x: ITest);
```

These declarations generate smaller, faster code.

One case where you cannot use reference counting, because it cannot be consistently applied, is if your object is a component or a control owned by another component. In that case, you can still use interfaces, but you should not use reference counting because the lifetime of the object is not dictated by its interfaces.

Not using reference counting

If your object is a component or a control that is owned by another component, then it is part of a different memory management system that is based in *TComponent*. Although some classes mix the object lifetime management approaches of *TComponent* and interface reference counting, this is very tricky to implement correctly.

To create a component that supports interfaces but bypasses the interface reference counting machanism, you must implement the *_AddRef* and *_Release* methods in code such as the following:

```
function TMyObject._AddRef: Integer;
begin
  Result := -1;
end;

function TMyObject._Release: Integer;
begin
  Result := -1;
end;
```

You would still implement *QueryInterface* as usual to provide dynamic querying on your object.

Note that, because you implement *QueryInterface*, you can still use the as operator for interfaces, as long as you create an interface identifier (IID). You can also use aggregation. If the outer object is a component, the inner object implements reference counting as usual, by delegating to the "controlling Unknown." It is at the level of the outer object that the decision is made to circumvent the *_AddRef* and *_Release* methods, and to handle memory management via another approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a as its containing outer object one that does not follow the interface lifetime model.

Note The “controlling Unknown” is the *IUnknown* implemented by the outer object and the one for which the reference count of the entire object is maintained. *IUnknown* is the same as *IInterface*. For more information distinguishing the various implementations of the *IUnknown* or *IInterface* interface by the inner and outer objects, see “Aggregation” on page 4-17.

Chapter

5

Using BaseCLX

There are a number of units in CLX that provide the underlying support for most of the component library. These units include the global routines that make up the CLX runtime library, a number of utility classes such as those that represent streams and lists, and the classes *TObject*, *TPersistent*, and *TComponent*. Collectively, these units are called BaseCLX. BaseCLX does not include any of the components that appear on the component palette. Rather, the classes and routines in BaseCLX are used by the components that do appear on the component palette and are available for you to use in application code or when you are writing your own classes.

Do not confuse the CLX runtime library that is part of BaseCLX with C++ runtime library. Many of the routines in the CLX runtime liabrary perform functions similar to those in the C++ runtime library, but can be distinguished because the function names begin with a capital letter and they are declared in the header of a unit.

The following topics discuss many of the classes and routines that make up BaseCLX and illustrate how to use them. These uses include:

- Using streams
- Working with files
- Working with .ini files
- Working with lists
- Working with string lists
- Working with strings
- Converting measurements
- Creating drawing spaces
- Defining custom variants in Delphi ~DefiningCustomVariants

Note This list of tasks is not exhaustive. The runtime library in BaseCLX contains many routines to perform tasks that are not mentioned here. These include a host of mathematical functions (defined in the Math unit), routines for working with date/time values (defined in the SysUtils and DateUtils units), and routines for working with Variant values (defined in the Variants unit).

Using streams

Streams are classes that let you read and write data. They provide a common interface for reading and writing to different media such as memory, strings, sockets, and BLOB fields in databases. There are several stream classes, which all descend from *TStream*. Each stream class is specific to one media type. For example, *TMemoryStream* reads from or writes to a memory image; *TFileStream* reads from or writes to a file.

Using streams to read or write data

Stream classes all share several methods for reading and writing data. These methods are distinguished by whether they

- Return the number of bytes read or written.
- Require the number of bytes to be known.
- Raise an exception on error.

Stream methods for reading and writing

The *Read* method reads a specified number of bytes from the stream, starting at its current *Position*, into a buffer. *Read* then advances the current position by the number of bytes actually transferred. The prototype for *Read* is

```
function Read(var Buffer; Count: Longint): Longint;
```

```
virtual int __fastcall Read(void *Buffer, int Count);
```

Read is useful when the number of bytes in the file is not known. *Read* returns the number of bytes actually transferred, which may be less than *Count* if the stream did not contain *Count* bytes of data past the current position.

The *Write* method writes *Count* bytes from a buffer to the stream, starting at the current *Position*. The prototype for *Write* is:

```
function Write(const Buffer; Count: Longint): Longint;
```

```
virtual int __fastcall Write(const void *Buffer, int Count);
```

After writing to the file, *Write* advances the current position by the number bytes written, and returns the number of bytes actually written, which may be less than *Count* if the end of the buffer is encountered or the stream can't accept any more bytes.

The counterpart procedures are *ReadBuffer* and *WriteBuffer* which, unlike *Read* and *Write*, do not return the number of bytes read or written. These procedures are useful in cases where the number of bytes is known and required, for example when reading in structures. *ReadBuffer* and *WriteBuffer* raise an exception (*EReadError* and *EWriteError*) if the byte count can not be matched exactly. This is in contrast to the *Read* and *Write* methods, which can return a byte count that differs from the requested value. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

```
virtual int __fastcall ReadBuffer(void *Buffer, int Count);
```

```
virtual int __fastcall WriteBuffer(const void *Buffer, int Count);
```

These methods call the *Read* and *Write* methods to perform the actual reading and writing.

Reading and writing components

TStream defines specialized methods, *ReadComponent* and *WriteComponent*, for reading and writing components. You can use them in your applications as a way to save components and their properties when you create or alter them at runtime.

ReadComponent and *WriteComponent* are the methods that the IDE uses to read components from or write them to form files. When streaming components to or from a form file, stream classes work with the *TFiler* classes, *TReader* and *TWriter*, to read objects from the form file or write them out to disk. For more information about using the component streaming system, see the online Help on the *TStream*, *TFiler*, *TReader*, *TWriter*, and *TComponent* classes.

Reading and writing strings in Delphi

In Delphi code, if you are passing a string to a read or write function, you need to be aware of the correct syntax. The *Buffer* parameters for the read and write routines are **var** and **const** types, respectively. These are untyped parameters, so the routine takes the address of a variable.

The most commonly used type when working with strings is a long string. However, passing a long string as the *Buffer* parameter does not produce the correct result. Long strings contain a size, a reference count, and a pointer to the characters in the string. Consequently, dereferencing a long string does not result in the pointer element. You need to first cast the string to a *Pointer* or *PChar,* and then dereference it. For example:

```
procedure caststring;
var
  fs: TFileStream;
const
  s: string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s));// this will give you garbage
  fs.Write(PChar(s)^, Length(s));// this is the correct way
end;
```

Copying data from one stream to another

When copying data from one stream to another, you do not need to explicitly read and then write the data. Instead, you can use the *CopyFrom* method, as illustrated in the following example.

The application includes two edit controls (From and To) and a Copy File button.

Delphi example

```
procedure TForm1.CopyFileClick(Sender: TObject);
var
  stream1, stream2:TStream;
begin
  stream1:=TFileStream.Create(From.Text,fmOpenRead or fmShareDenyWrite);
  try
    stream2 := TFileStream.Create(To.Text fmOpenCreate or fmShareDenyRead);
    try
      stream2.CopyFrom(Stream1,Stream1.Size);
    finally
      stream2.Free;
  finally
    stream1.Free
end;
```

C++ example

```
void __fastcall TForm1::CopyFileClick(TObject *Sender)
{
  TStream* stream1= new TFileStream(From->Text,fmOpenRead | fmShareDenyWrite);
  try
  {
    TStream* stream2 = new TFileStream(To->Text, fmOpenWrite | fmShareDenyRead);
    try
    {
      stream2 -> CopyFrom(stream1, stream1->Size);
    }
    __finally
    {
      delete stream2;
    }
  }
  __finally
  {
    delete stream1;
  }
}
```

Specifying the stream position and size

In addition to methods for reading and writing, streams permit applications to seek to an arbitrary position in the stream or change the size of the stream. Once you seek to a specified position, the next read or write operation starts reading from or writing to the stream at that position.

Seeking to a specific position

The *Seek* method is the most general mechanism for moving to a particular position in the stream. There are two overloads for the *Seek* method:

```
function Seek(Offset: Longint; Origin: Word): Longint;
function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
```

```
virtual int __fastcall Seek(int Offset, Word Origin);
virtual __int64 __fastcall Seek(const __int64 Offset, TSeekOrigin Origin);
```

Both overloads work the same way. The difference is that one version uses a 32-bit integer to represent positions and offsets, while the other uses a 64-bit integer.

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

Value	Meaning
soFromBeginning	Offset is from the beginning of the resource. Seek moves to the position Offset. Offset must be >= 0.
soFromCurrent	Offset is from the current position in the resource. Seek moves to Position + Offset.
soFromEnd	Offset is from the end of the resource. Offset must be <= 0 to indicate a number of bytes before the end of the file.

Seek resets the current stream position, moving it by the indicated offset. *Seek* returns the new current position in the stream.

Using Position and Size properties

All streams have properties that hold the current position and size of the stream. These are used by the *Seek* method, as well as all the methods that read from or write to the stream.

The *Position* property indicates the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
property Position: Int64;
__property __int64 Position = {read=GetPosition, write=SetPosition, nodefault};
```

The *Size* property indicates the size of the stream in bytes. It can be used to determine the number of bytes available for reading, or to truncate the data in the stream. The declaration for *Size* is:

```
property Size: Int64;
__property __int64 Size = {read=GetSize, write=SetSize64, nodefault};
```

Size is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the data in the stream. For example, on a file stream, setting *Size* inserts an end of file marker to truncate the file. If the *Size* of the stream cannot be changed, an exception is raised. For example, trying to change the *Size* of a read-only file stream raises an exception.

Working with files

BaseCLX supports several ways of working with files. The previous section, “Using streams,” has already mentioned that you can use specialized streams to read from or write to files. In addition to using file streams, there are several runtime library routines for performing file I/O. Both file streams and the global routines for reading from and writing to files are described in "Approaches to file I/O" on page 5-6.

In addition to input/output operations, you may want to manipulate files on disk. Support for operations on the files themselves rather than their contents is described in "Manipulating files" on page 5-8.

D In Delphi, remember that although the Delphi language is not case sensitive, the Linux operating system is. When using objects and routines that work with files, be attentive to the case of file names.

Approaches to file I/O

There are a few different approaches you can take when reading from and writing to files:

- The recommended approach for working with files is to use file streams. File streams are instances of the *TFileStream* class used to access information in disk files. File streams are a portable and high-level approach to file I/O. Because file streams make the file handle available, this approach can be combined with the next one. The next section, "Using file streams" discusses *TFileStream* in detail.
- You can work with files using a handle-based approach. File handles are provided by the operating system when you create or open a file to work with its contents. The SysUtils unit defines a number of file-handling routines that work with files using file handles. To use a handle-based approach, you first open a file using the *FileOpen* function or create a new file using the *FileCreate* function. Once you have the handle, use handle-based routines to work with its contents (write a line, read text, and so on).
- In Delphi, the System unit defines a number of file I/O routines that work with file variables, usually of the format "F: Text:" or "F: File". File variables can have one of three types: typed, text, and untyped. A number of file-handling routines, such as *AssignPrn* and *writeln,* use them. The use of file variables is deprecated, and these file types are supported only for backward compatibility. They are incompatible with file handles. If you need to work with them, see the *Delphi Language Guide*.
- The C runtime library and standard C++ library include a number of functions and classes for working with files. These have the advantage that they can be used in applications that do not use CLX. For information on these functions, see the online documentation for the C runtime library or the standard C++ library.

Using file streams

The *TFileStream* class enables applications to read from and write to a file on disk. Because *TFileStream* is a stream object, it shares the common stream methods. You can use these methods to read from or write to the file, copy data to or from other stream classes, and read or write components values. See "Using streams" on page 5-2 for details on the capabilities that files streams inherit by being stream classes.

In addition, file streams give you access to the file handle, so that you can use them with global file handling routines that require the file handle.

Creating and opening files using file streams

To create or open a file and get access to its handle, you simply instantiate a *TFileStream*. This opens or creates a specified file and provides methods to read from or write to it. If the file cannot be opened, the *TFileStream* constructor raises an exception.

```
(constructor Create(const filename: string; Mode: Word);
```

```
__fastcall TFileStream(const AnsiString FileName, Word Mode);
```

The *Mode* parameter specifies how the file should be opened when creating the file stream. The *Mode* parameter consists of an open mode and a share mode OR'ed together. The open mode must be one of the following values:

Table 5.1 Open modes

Value	Meaning
fmCreate	TFileStream a file with the given name. If a file with the given name exists, open the file in write mode.
fmOpenRead	Open the file for reading only.
fmOpenWrite	Open the file for writing only. Writing to the file completely replaces the current contents.
fmOpenReadWrite	Open the file to modify the current contents rather than replace them.

The share mode can be one of the following values with the restrictions listed below:

Table 5.2 Share modes

Value	Meaning
fmShareCompat	Sharing is compatible with the way FCBs are opened.
fmShareExclusive	Other applications can not open the file for any reason.
fmShareDenyWrite	Other applications can open the file for reading but not for writing.
fmShareDenyRead	Other applications can open the file for writing but not for reading.
fmShareDenyNone	No attempt is made to prevent other applications from reading from or writing to the file.

Note that which share mode you can use depends on which open mode you used. The following table shows shared modes that are available for each open mode.

Table 5.3 Shared modes available for each open mode

Open Mode	fmShareCompat	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead	fmShareDenyNone
fmOpenRead	Can't use	Can't use	Available	Can't use	Available
fmOpenWrite	Available	Available	Can't use	Available	Available
fmOpenReadWrite	Available	Available	Available	Available	Available

The file open and share mode constants are defined in the SysUtils unit.

Using the file handle

When you instantiate *TFileStream* you get access to the file handle. The file handle is contained in the *Handle* property. *Handle* is read-only and reflects the mode in which the file was opened. If you want to change the attributes of the file *Handle*, you must create a new file stream object.

Some file manipulation routines take a file handle as a parameter. Once you have a file stream, you can use the *Handle* property in any situation in which you would use a file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

Manipulating files

Several common file operations are built into the BaseCLX runtime library. The routines for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead.

Although the Delphi language is not case sensitive, the Linux operating system is. Be attentive to case when working with files.

Deleting a file

Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm before deleting files. To delete a file, pass the name of the file to the *DeleteFile* function:

```
DeleteFile(FileName);
```

DeleteFile returns true if it deleted the file and false if it did not (for example, if the file did not exist or if it was read-only). *DeleteFile* erases the file named by *FileName* from the disk.

Finding a file

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. You should always use *FindClose* to terminate a *FindFirst*/*FindNext* sequence. If you want to know if a file exists, a *FileExists* function returns true if the file exists, false otherwise.

The three file find routines take a *TSearchRec* as one of the parameters. *TSearchRec* defines the file information searched for by *FindFirst* or *FindNext*. If a file is found, the fields of the *TSearchRec* type parameter are modified to describe the found file.

On field of *TSearchRec* that is of particular interest is the *Attr* field. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

Table 5.4 Attribute constants and values

Constant	Value	Description
faReadOnly	1	Read-only files
faHidden	2	Hidden files
faSysFile	4	System files
faVolumeID	8	Volume ID files
faDirectory	16	Directory files
faArchive	32	Archive files
faAnyFile	79	Any file

To test for an attribute, combine the value of the *Attr* field with the attribute constant using the and (Delphi) or & (C++) operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to true

```
(SearchRec.Attr and faHidden > 0).
```

```
(SearchRec.Attr & faHidden > 0).
```

Attributes can be combined by OR'ing their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass the following value as the *Attr* parameter:

```
(faReadOnly or faHidden).
```

```
(faReadOnly | faHidden).
```

The following example illustrates the use of the three file find routines. It uses a label, a button named *Search*, and a button named *Again* on a form. When the user clicks the *Search* button, the first file in the specified path is found, and the name and the number of bytes in the file appear in the label's caption. Each time the user clicks the *Again* button, the next matching filename and size is displayed in the label.

Delphi example

```
var
  SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('/usr/local/MyProgram/*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
end;

procedure TForm1.AgainClick(Sender: TObject);
begin
  if FindNext(SearchRec) = 0 then
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size'
  else
```

```
    FindClose(SearchRec);
end;
```

C++ example

```
  TSearchRec SearchRec; // global variable

void __fastcall TForm1::SearchClick(TObject *Sender)
{
  FindFirst("/usr/local/MyProgram/*.*", faAnyFile, SearchRec);
  Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in size";
}

void __fastcall TForm1::AgainClick(TObject *Sender)
{
  if (FindNext(SearchRec) == 0)
    Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in
size";
  else
    FindClose(SearchRec);
}
```

Renaming a file

To change a file name, use the *RenameFile* function:

D
```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

```
extern PACKAGE bool __fastcall RenameFile(const AnsiString OldName, const AnsiString
NewName);
```

RenameFile changes a file name, identified by *OldFileName*, to the name specified by *NewFileName*. If the operation succeeds, *RenameFile* returns true. If it cannot rename the file (for example, if a file called *NewFileName* already exists), *RenameFile* returns false. For example:

D
```
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');
```

```
if (!RenameFile("OLDNAME.TXT","NEWNAME.TXT"))
  ErrorMsg("Error renaming file!");
```

File date-time routines

The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or an error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or -1 if the handle is invalid.

As with most of the file manipulating routines, *FileAge* uses a string filename. *FileGetDate* and *FileSetDate*, however, use a *Handle* type as a parameter in Delphi, or an integer parameter which takes a file handle in C++. To get the file handle either

- Use the *FileOpen* or *FileCreate* function to create a new file or open an existing file. Both *FileOpen* and *FileCreate* return the file handle.

- Instantiate *TFileStream* to create or open a file. Then use its *Handle* property. See "Using file streams" on page 5-6 for more information.

Working with ini files

Many applications use ini files to store configuration information. BaseCLX includes two classes for working with ini files: *TIniFile* and *TMemIniFile*.

TMemIniFile and *TIniFile* are identical. When you instantiate the *TIniFile* or *TMemIniFile* object, you pass the name of the ini file as a parameter to the constructor. If the file does not exist, it is automatically created. You are then free to read values using the various read methods, such as *ReadString*, *ReadDate*, *ReadInteger*, or *ReadBool*. Alternatively, if you want to read an entire section of the ini file, you can use the *ReadSection* method. Similarly, you can write values using methods such as *WriteBool*, *WriteInteger*, *WriteDate*, or *WriteString*.

Following is an example of reading configuration information from an ini file in a form's *OnCreate* event handler (Delphi) or constructor (C++) and writing values in the *OnClose* event handler.

D Delphi example

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create( ChangeFileExt( Application.ExeName, '.INI' ) );
  try
    Top     := Ini.ReadInteger( 'Form', 'Top', 100 );
    Left    := Ini.ReadInteger( 'Form', 'Left', 100 );
    Caption := Ini.ReadString( 'Form', 'Caption', 'New Form' );
    if Ini.ReadBool( 'Form', 'InitMax', false ) then
      WindowState = wsMaximized
    else
      WindowState = wsNormal;
  finally
    TIniFile.Free;
  end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action TCloseAction)
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create( ChangeFileExt( Application.ExeName, '.INI' ) );
  try
    Ini.WriteInteger( 'Form', 'Top', Top);
    Ini.WriteInteger( 'Form', 'Left', Left);
    Ini.WriteString( 'Form', 'Caption', Caption );
    Ini.WriteBool( 'Form', 'InitMax', WindowState = wsMaximized );
  finally
    TIniFile.Free;
  end;
end;
```

C++ example

```
__fastcall TForm1::TForm1(TComponent *Owner) : TForm(Owner)
{
   TIniFile *ini;
   ini = new TIniFile( ChangeFileExt( Application->ExeName, ".INI" ) );

   Top     =  ini->ReadInteger( "Form", "Top", 100 );
   Left    =  ini->ReadInteger( "Form", "Left", 100 );
   Caption =  ini->ReadString( "Form", "Caption",
                               "Default Caption" );
   ini->ReadBool( "Form", "InitMax", false ) ?
         WindowState = wsMaximized :
         WindowState = wsNormal;

  delete ini;
}

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   TIniFile *ini;
   ini = new TIniFile(ChangeFileExt( Application->ExeName, ".INI" ) );
   ini->WriteInteger( "Form", "Top", Top );
   ini->WriteInteger( "Form", "Left", Left );
   ini->WriteString ( "Form", "Caption", Caption );
   ini->WriteBool   ( "Form", "InitMax",
                    WindowState == wsMaximized );

   delete ini;
}
```

Each of the *Read* routines takes three parameters. The first parameter identifies the section of the ini file. The second parameter identifies the value you want to read, and the third is a default value in case the section or value doesn't exist in the ini file. Just as the *Read* methods gracefully handle the case when a section or value does not exist, the *Write* routines create the section and/or value if they do not exist. The example code creates an ini file the first time it is run that looks like this:

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

On subsequent execution of this application, the ini values are read in when the form is created and written back out in the *OnClose* event.

Working with lists

BaseCLX includes many classes that represents lists or collections of items. They vary depending on the types of items they contain, what operations they support, and whether they are persistent.

The following table lists various list classes, and indicates the types of items they contain:

Table 5.5 Classes for managing lists

Object	Maintains
TList	A list of pointers
TThreadList	A thread-safe list of pointers
TBucketList	A hashed list of pointers
TObjectBucketList	A hashed list of object instances
TObjectList	A memory-managed list of object instances
TComponentList	A memory-managed list of components (that is, instances of classes descended from *TComponent*)
TClassList	A list of class references (metaclasses)
TInterfaceList	A list of interface pointers.
TQueue	A first-in first-out list of pointers
TStack	A last-in first-out list of pointers
TObjectQueue	A first-in first-out list of objects
TObjectStack	A last-in first-out list of objects
TCollection	Base class for many specialized classes of typed items.
TStringList	A list of strings
THashedStringList	A list of strings with the form Name=Value, hashed for performance.

Common list operations

Although the various list classes contain different types of items and have different ancestries, most of them share a common set of methods for adding, deleting, rearranging, and accessing the items in the list.

Adding list items

Most list classes have an *Add* method, which lets you add an item to the end of the list (if it is not sorted) or to its appropriate position (if the list is sorted). Typically, the *Add* method takes as a parameter the item you are adding to the list and returns the position in the list where the item was added. In the case of bucket lists (*TBucketList* and *TObjectBucketList*), Add takes not only the item to add, but also a datum you can associate with that item. In the case of collections, *Add* takes no parameters, but creates a new item that it adds. The *Add* method on collections returns the item it added, so that you can assign values to the new item's properties.

Some list classes have an *Insert* method in addition to the *Add* method. Insert works the same way as the *Add* method, but has an additional parameter that lets you specify the position in the list where you want the new item to appear. If a class has an Add method, it also has an Insert method unless the position of items is predetermined For example, you can't use *Insert* with sorted lists because items must go in sort order, and you can't use *Insert* with bucket lists because the hash algorithm determines the item position.

The only classes that do not have an *Add* method are the ordered lists. Ordered lists are queues and stacks. To add items to an ordered list, use the *Push* method instead. *Push*, like *Add*, takes an item as a parameter and inserts it in the correct position.

Deleting list items

To delete a single item from one of the list classes, use either the *Delete* method or the *Remove* method. *Delete* takes a single parameter, the index of the item to remove. *Remove* also takes a single parameter, but that parameter is a reference to the item to remove, rather than its index. Some list classes support only a *Delete* method, some support only a *Remove* method, and some have both.

As with adding items, ordered lists behave differently than all other lists. Instead of using a *Delete* or *Remove* method, you remove an item from an ordered list by calling its *Pop* method. *Pop* takes no arguments, because there is only one item that can be removed.

If you want to delete all of the items in the list, you can call the *Clear* method. Clear is available for all lists except ordered lists.

Accessing list items

All list classes (except *TThreadList* and the ordered lists) have a property that lets you access the items in the list. Typically, this property is called *Items*. For string lists, the property is called *Strings*, and for bucket lists it is called *Data*. The *Items*, *Strings*, or *Data* property is an indexed property, so that you can specify which item you want to access.

On *TThreadList*, you must lock the list before you can access items. When you lock the list, the *LockList* method returns a *TList* object that you can use to access the items.

Ordered lists only let you access the "top" item of the list. You can obtain a reference to this item by calling the *Peek* method.

Rearranging list items

Some list classes have methods that let you rearrange the items in the list. Some have an *Exchange* method, that swaps the position of two items. Some have a *Move* method that lets you move an item to a specified location. Some have a *Sort* method that lets you sort the items in the list.

To see what methods are available, check the Online help for the list class you are using.

Persistent lists

Persistent lists can be saved to a form file. Because of this, they are often used as the type of a published property on a component. You can add items to the list at design time, and those items are saved with the object so that they are there when the component that uses them is loaded into memory at runtime. There are two main types of persistent lists: string lists and collections.

Examples of string lists include *TStringList* and *THashedStringList*. String lists, as the name implies, contain strings. They provide special support for strings of the form Name=Value, so that you can look up the value associated with a name. In addition, most string lists let you associate an object with each string in the list. String lists are described in more detail in "Working with string lists" on page 5-15.

Collections descend from the class *TCollection*. Each *TCollection* descendant is specialized to manage a specific class of items, where that class descends from *TCollectionItem*. Collections support many of the common list operations. All collections are designed to be the type of a published property, and many can not function independently of the object that uses them to implement on of its properties. At design time, the property whose value is a collection can use the collection editor to let you add, remove, and rearrange items. The collection editor provides a common user interface for manipulating collections.

Working with string lists

One of the most commonly used types of list is a list of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. BaseCLX provides a common interface to any list of strings through an object called *TStrings* and its descendants such as *TStringList* and *THashedStringList. TStringList* implements the abstract properties and methods introduced by *TStrings*, and introduces properties, events, and methods to

- Sort the strings in the list.
- Prohibit duplicate strings in sorted lists.
- Respond to changes in the contents of the list.

In addition to providing functionality for maintaining string lists, these objects allow easy interoperability; for example, you can edit the lines of a memo (which are a *TStrings* descendant) and then use these lines as items in a combo box (also a *TStrings* descendant).

A string-list property appears in the Object Inspector with *TStrings* in the Value column. Double-click *TStrings* to open the String List editor, where you can edit, add, or delete lines.

You can also work with string-list objects at runtime to perform such tasks as

- Loading and saving string lists
- Creating a new string list
- Manipulating strings in a list
- Associating objects with a string list

Loading and saving string lists

String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes.

The following example loads a copy of the MyFile.ini file into a memo field and makes a backup copy called MyFile.bak.

Delphi example

```
procedure EditWinIni;
var
  FileName: string;{ storage for file name }
begin
  FileName := '/usr/local/MyProgram/MyFile.ini'; { set the file name }
  with Form1.Memo1.Lines do
  begin
    LoadFromFile(FileName);{ load from file }
    SaveToFile(ChangeFileExt(FileName, '.bak'));{ save into backup file }
  end;
end;
```

C++ example

```
void __fastcall EditWinIni()
{
  AnsiString FileName = "C:/usr/local/MyProgram/MyFile.ini";// set the file name
  Form1->Memo1->Lines->LoadFromFile(FileName);      // load from file
  Form1->Memo1->Lines->SaveToFile(ChangeFileExt(FileName, ".bak"));  // save to backup
}
```

Creating a new string list

A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.

Short-term string lists

If you use a string list only for the duration of a single routine, you can create it, use it, and destroy it all in one place. This is the safest way to work with string lists. Because the string-list object allocates memory for itself and its strings, you should use a try...finally block (Delphi) or try...__finally block (C++) to ensure that the memory is freed even if an exception occurs.

1 Construct the string-list object.
2 In the try part of a try...finally block, use the string list.
3 In the finally (__finally) part, free the string-list object.

The following event handler responds to a button click by constructing a string list, using it, and then destroying it.

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TempList: TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
  try
    { use the string list }
  finally
    TempList.Free;{ destroy the list object }
  end;
end;
```

C++ example

```
void __fastcall TForm1::ButtonClick1(TObject *Sender)
{
   TStringList *TempList = new TStringList; // declare the list
   try{
      //use the string list
   }
   __finally{
    delete TempList; // destroy the list object
   }
}
```

Long-term string lists

If a string list must be available at any time while your application runs, construct the list at start-up and destroy it before the application terminates.

1 In the unit file for your application's main form, add a field of type *TStrings* to the form's declaration.

2 Write an event handler for the main form's *OnCreate* event that executes before the form appears. It should create a string list and assign it to the field you declared in the first step. In C++, if *OldCreateOrder* is true, you should create the string list in the *constructor* for the main form instead.

3 Write an event handler that frees the string list for the form's *OnClose* event.

This example uses a long-term string list to record the user's mouse clicks on the main form, then saves the list to a file before the application terminates.

Delphi example

```
unit Unit1;
```

```
interface
uses SysUtils, Variants, Classes, QGraphics, WControls, QForms, QDialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;{ declare the field }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ClickList := TStringList.Create;{ construct the list }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.log'));{ save the list }
  ClickList.Free;{ destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ClickList.Add(Format('Click at (%d, %d)', [X, Y]));{ add a string to the list }
end;

end.
```

C++ example

```
//---------------------------------------------------------------------------
#include <clx.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
       : TForm(Owner)
{
   ClickList = new TStringList;
}
```

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   ClickList->SaveToFile(ChangeFileExt(Application->ExeName, ".log"));//Save the list
   delete ClickList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
   TVarRec v[] = {X,Y};
   ClickList->Add(Format("Click at (%d, %d)",v,ARRAYSIZE(v) - 1));//add a string to the list
}
```

Manipulating strings in a list

Operations commonly performed on string lists include

- Counting the strings in a list
- Accessing a particular string
- Finding the position of a string in the list
- Iterating through strings in a list
- Adding a string to a list
- Moving a string within a list
- Deleting a string from a list
- Copying a complete string list

Counting the strings in a list

The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

Accessing a particular string

The *Strings* array property contains the strings in the list, referenced by a zero-based index.

D In Delphi, because *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus

```
StringList1.Strings[0] := 'This is the first string.';
```

is equivalent to

```
StringList1[0] := 'This is the first string.';
```

In C++, you can use the [] operator for a similar effect. That is,

```
StringList1->Strings[0] = "This is the first string.";
```

is equivalent to

```
(*StringList1)[0] = "This is the first string.";
```

Locating items in a string list

To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns –1 if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself.

For example, you could use *IndexOf* to determine whether a given file name is found among the *Items* of a list box:

```
if FileListBox1.Items.IndexOf(TargetFileName) > -1 ...
```

```
if (FileListBox1->Items->IndexOf(TargetFileName) > -1) ...
```

Iterating through strings in a list

To iterate through the strings in a list, use a for loop that runs from zero to *Count* – 1.

This example converts each string in a list box to uppercase characters.

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Index: Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for (int i = 0; i < ListBox1->Items->Count; i++)
    ListBox1->Items->Strings[i] = UpperCase(ListBox1->Items->Strings[i]);
}
```

Adding a string to a list

To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string “Three” the third string in a list, you would use:

```
Insert(2, 'Three');
```

```
StringList1->Insert(2, "Three");
```

To append the strings from one list onto another, call *AddStrings*:

```
StringList1.AddStrings(StringList2);  { append the strings from StringList2 to StringList1 }
```

```
StringList1->AddStrings(StringList2);  // append the strings from StringList2 to StringList1
```

Moving a string within a list

To move a string in a string list, call the *Move* method, passing two parameters: the current index of the string and the index you want assigned to it. For example, to move the third string in a list to the fifth position, you would use:

```
StringListObject.Move(2, 4);
```

```
StringListObject->Move(2, 4);
```

Deleting a string from a list

To delete a string from a string list, call the list's *Delete* method, passing the index of the string you want to delete. If you don't know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method.

This example uses *IndexOf* and *Delete* to find and delete a string:

Delphi example

```
with ListBox1.Items do
begin
  BIndex := IndexOf('bureaucracy');
  if BIndex > -1 then
    Delete(BIndex);
end;
```

C++ example

```
int BIndex = ListBox1->Items->IndexOf("bureaucracy");
if (BIndex > -1)
  ListBox1->Items->Delete(BIndex);
```

Copying a complete string list

You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,

```
Memo1.Lines.Assign(ComboBox1.Items);    { overwrites original strings }
```

```
Memo1->Lines->Assign(ComboBox1->Item)s; //overwrites original strings
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
Memo1.Lines.AddStrings(ComboBox1.Items);   { appends strings to end }
```

```
Memo1->Lines->AddStrings(ComboBox1->Items);//appends strings to end
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you assign one string-list variable to another—

```
StringList1 := StringList2;
```

```
StringList1 = StringList2;
```

—the original string-list object will be lost, often with unpredictable results.

Associating objects with a string list

In addition to the strings stored in its *Strings* property, a string list can maintain references to objects, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls.

Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear,* and *Move* operate on both strings and objects; for example, deleting a string removes the corresponding object (if there is one).

To associate an object with an existing string, assign the object to the *Objects* property at the same index. You cannot add an object without adding a corresponding string.

Working with strings

The BaseCLX runtime library provides many specialized string-handling routines specific to a string type. These are routines for wide strings, long strings (AnsiStrings), and null-terminated strings (PChar in Delphi or char * in C++). Routines that deal with null-terminated strings use the null-termination to determine the length of the string.

D There are no categories of routines listed for the Delphi *ShortString* type. However, some Delphi built-in compiler routines deal with the *ShortString* type. These include, for example, the *Low* and *High* standard functions. For more details about the various string types, see the *Delphi Language Guide*.

The following topics provide an overview of many of the string-handling routines in the runtime library.

Wide character routines

Wide strings are used in a variety of situations. Some technologies, such as XML, use wide strings as a native type. You may also choose to use wide strings because they simplify some of the string-handling issues in applications that have multiple target locales. Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second part of a character for the start of a different character.

The following functions convert between standard single-byte character strings (or MBCS strings) and Unicode strings:

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString

- WideCharToStrVar

In addition, the following functions translate between WideStrings and other representations:

- UCS4StringToWideString
- WideStringToUCS4String
- VarToWideStr
- VarToWideStrDef

The following routines work directly with WideStrings:

- WideCompareStr
- WideCompareText
- WideSameStr
- WideSameText
- WideSameCaption (CLX only)
- WideFmtStr
- WideFormat
- WideLowerCase
- WideUpperCase

Finally, some routines include overloads for working with wide strings:

- UniqueString
- Length
- Trim
- TrimLeft
- TrimRight

Commonly used routines for AnsiStrings

The Delphi long string type is represented as the AnsiString class in C++. This string type is one of the most common ways to represent string values in CLX. The BaseCLX runtime library includes a number of routines for dealing with these strings, which we will call AnsiStrings.

AnsiString-handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether they use a particular criterion in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string

Where appropriate, the tables also provide columns indicating whether a routine satisfies the following criteria.

- Uses case sensitivity: If locale settings are used, it determines the definition of case. If the routine does not use locale settings, analyses are based upon the ordinal values of the characters. If the routine is case-insensitive, there is a logical merging of upper and lower case characters that is determined by a predefined pattern.

- Uses locale settings: Locale settings allow you to customize your application for specific locales, in particular, for Asian language environments. Most locale settings consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters. Routines that use the system locale are typically prefaced with Ansi (that is, Ansi*XXX*).
- Supports the multi-byte character set (MBCS): MBCSs are used when writing code for far eastern locales. Multi-byte characters are represented by one or more character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS parse one- and multi-byte characters.

  ByteType and *StrByteType* determine whether a particular byte is the lead byte of a multibyte character. Be careful when using multibyte characters not to truncate a string by cutting a character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character cannot be predetermined. Pass, instead, a pointer to a to a character or string. For more information about MBCS, see "Enabling application code" on page 17-2 of Chapter 17, "Creating international applications."

Table 5.6 String comparison routines

Routine	Case-sensitive	Uses locale settings	Supports MBCS
AnsiCompareStr	yes	yes	yes
AnsiCompareText	no	yes	yes
AnsiCompareFileName	no	yes	yes
AnsiMatchStr	yes	yes	yes
AnsiMatchText	no	yes	yes
AnsiContainsStr	yes	yes	yes
AnsiContainsText	no	yes	yes
AnsiStartsStr	yes	yes	yes
AnsiStartsText	no	yes	yes
AnsiEndsStr	yes	yes	yes
AnsiEndsText	no	yes	yes
AnsiIndexStr	yes	yes	yes
AnsiIndexText	no	yes	yes
CompareStr	yes	no	no
CompareText	no	no	no
AnsiResemblesText	no	no	no

Table 5.7 Case conversion routines

Routine	Uses locale settings	Supports MBCS
AnsiLowerCase	yes	yes
AnsiLowerCaseFileName	yes	yes
AnsiUpperCaseFileName	yes	yes
AnsiUpperCase	yes	yes
LowerCase	no	no
UpperCase	no	no

Note The routines used for string file names: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the system locale. You should always use file names that are portable because the locale (character set) used for file names can and might differ from the default user interface.

Table 5.8 String modification routines

Routine	Case-sensitive	Supports MBCS
AdjustLineBreaks	NA	yes
AnsiQuotedStr	NA	yes
AnsiReplaceStr	yes	yes
AnsiReplaceText	no	yes
StringReplace	optional by flag	yes
ReverseString	NA	no
StuffString	NA	no
Trim	NA	yes
TrimLeft	NA	yes
TrimRight	NA	yes
WrapText	NA	yes

Table 5.9 Sub-string routines

Routine	Case-sensitive	Supports MBCS
AnsiExtractQuotedStr	NA	yes
AnsiPos	yes	yes
IsDelimiter	yes	yes
IsPathDelimiter	yes	yes
LastDelimiter	yes	yes
LeftStr	NA	no
RightStr	NA	no
MidStr	NA	no
QuotedStr	no	no

Commonly used routines for null-terminated strings

The null-terminated string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string
- Copying

Where appropriate, the tables also provide columns indicating whether the routine is case-sensitive, uses the current locale, and/or supports multi-byte character sets.

Table 5.10 Null-terminated string comparison routines

Routine	Case-sensitive	Uses locale settings	Supports MBCS
AnsiStrComp	yes	yes	yes
AnsiStrIComp	no	yes	yes
AnsiStrLComp	yes	yes	yes
AnsiStrLIComp	no	yes	yes
StrComp	yes	no	no
StrIComp	no	no	no
StrLComp	yes	no	no
StrLIComp	no	no	no

Table 5.11 Case conversion routines for null-terminated strings

Routine	Uses locale settings	Supports MBCS
AnsiStrLower	yes	yes
AnsiStrUpper	yes	yes
StrLower	no	no
StrUpper	no	no

Table 5.12 String modification routines

Routine
StrCat
StrLCat

Table 5.13 Sub-string routines

Routine	Case-sensitive	Supports MBCS
AnsiStrPos	yes	yes
AnsiStrScan	yes	yes
AnsiStrRScan	yes	yes

Table 5.13 Sub-string routines

Routine	Case-sensitive	Supports MBCS
StrPos	yes	no
StrScan	yes	no
StrRScan	yes	no

Table 5.14 String copying routines

Routine
StrCopy
StrLCopy
StrECopy
StrMove
StrPCopy
StrPLCopy

D Declaring and initializing strings in Delphi

In Delphi, when you declare a long string:

```
S: string;
```

you do not need to initialize it. Long strings are automatically initialized to empty. To test a string for empty you can either use the *EmptyStr* variable:

```
  S = EmptyStr;
```

or test against an empty string:

```
  S = '';
```

An empty string has no valid data. Therefore, trying to index an empty string is like trying to access nil and will result in an access violation:

```
var
  S: string;
begin
  S[i];    // this will cause an access violation
  // statements
end;
```

Similarly, if you cast an empty string to a *PChar*, the result is a nil pointer. So, if you are passing such a *PChar* to a routine that needs to read or write to it, be sure that the routine can handle nil:

```
var
  S: string;   // empty string
begin
  proc(PChar(S));  // be sure that proc can handle nil
  // statements
end;
```

If it cannot, then you can either initialize the string:

```
S := 'No longer nil';
proc(PChar(S));// proc does not need to handle nil now
```

or set the length, using the *SetLength* procedure:

```
SetLength(S, 100);//sets the dynamic length of S to 100
proc(PChar(S));// proc does not need to handle nil now
```

When you use *SetLength*, existing characters in the string are preserved, but the contents of any newly allocated space is undefined. Following a call to *SetLength*, *S* is guaranteed to reference a unique string, that is a string with a reference count of one. To obtain the length of a string, use the *Length* function.

Remember when declaring a string that:

```
S: string[n];
```

implicitly declares a short string, not a long string of *n* length. To declare a long string of specifically *n* length, declare a variable of type string and use the *SetLength* procedure.

```
S: string;
SetLength(S, n);
```

Mixing and converting Delphi string types

Short, long, and wide strings can be mixed in assignments and expressions, and the Delphi compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable.

Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.

Additional functions (*CopyQStringListToTstrings, Copy TStringsToQStringList, QStringListToTStringList*) are provided for converting underlying Qt string types and CLX string types. These functions are located in Qtypes.pas.

String to PChar conversions

Delphi long string to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic:

- Long strings are reference-counted, while *PChars* are not.
- Assigning to a string copies the data, while a *PChar* is a pointer to memory.
- Long strings are null-terminated and also contain the length of the string, while *PChars* are simply null-terminated.

Situations in which these differences can cause subtle errors are discussed in the following topics.

String dependencies

Sometimes you need convert a Delphi long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. Because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also disappear, while the routine you passed it to may still be using it. For example:

```
procedure my_func(x: string);
begin
  // do something with x
  some_proc(PChar(x)); // cast the string to a PChar
  // you now need to guarantee that the string remains
  // as long as the some_proc procedure needs to use it
end;
```

Returning a PChar local variable

A common error when working with the Delphi *PChar* type is to store a local variable in a data structure or return it as a value. When your routine ends, the *PChar* disappears because it is a pointer to memory, and not a reference-counted copy of the string. For example:

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format('title - %d', [n]);
  Result := PChar(s); // DON'T DO THIS
end;
```

This example returns a pointer to string data that is freed when the *title* function returns.

Passing a local variable as a PChar

Consider the case where you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local array of char and pass it to the function, then assign that variable to the string:

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
  . . .
end;

// assume MAX_SIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := FillBuffer(0, buf, SizeOf(buf));// treats buf as a PChar
  S := buf;
  //statements
end;
```

This approach is useful if the size of the buffer is relatively small, since it is allocated on the stack. It is also safe, since the conversion between an array of char and a string is automatic. The *Length* of the string is automatically set to the right value after assigning *buf* to the string.

To eliminate the overhead of copying the buffer, you can cast the string to a *PChar* (if you are certain that the routine does not need the *PChar* to remain in memory). However, synchronizing the length of the string does not happen automatically, as it does when you assign an array of char to a string. You should reset the string *Length* so that it reflects the actual width of the string. If you are using a function that returns the number of bytes copied, you can do this safely with one line of code:

```
var
  S: string;
begin
  SetLength(S, MAX_SIZE;// when casting to a PChar, be sure the string is not empty
  SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
  // statements
end;
```

D Delphi Compiler directives for strings

The following compiler directives affect character and string types.

Table 5.15 Compiler directives for strings

Directive	Description
{$H+/-}	A compiler directive, $H, controls whether the reserved word string represents a short string or a long string. In the default state, {$H+}, string represents a long string. You can change it to a *ShortString* by using the {$H-} directive.
{$P+/-}	The $P directive is meaningful only for code compiled in the {$H-} state, and is provided for backwards compatibility. $P controls the meaning of variable parameters declared using the string keyword in the {$H-} state. In the {$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {$P+} state, they are open string parameters. Regardless of the setting of the $P directive, the OpenString identifier can always be used to declare open string parameters.

Table 5.15 Compiler directives for strings (continued)

Directive	Description

{$V+/-}	The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types.
	In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example:
	`var S: string[3];`
	`procedure Test(var T: string);`
	`begin`
	`  T := '1234';`
	`end;`
	`begin`
	`  Test(S);`
	`end.`

{$X+/-}	The {$X+} compiler directive enables support for null-terminated strings by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with *Write*, *Writeln*, *Val*, *Assign*, and *Rename* from the System unit.)

Creating drawing spaces

The *TCanvas* class encapsulates a paint device (Qt painter). This class handles all drawing for forms, visual containers (such as panels) and the printer object (see "Printing" on page 5-32). Using the canvas object, you need not worry about allocating pens, brushes, palettes, and so on—all the allocation and deallocation are handled for you.

TCanvas includes a large number of primitive graphics routines to draw lines, shapes, polygons, fonts, etc. onto any control that contains a canvas. For example, here is a button event handler that draws a line from the upper left corner to the middle of the form and outputs some raw text onto the form:

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Canvas.Pen.Color := clBlue;
   Canvas.MoveTo( 10, 10 );
   Canvas.LineTo( 100, 100 );
   Canvas.Brush.Color := clBtnFace;
   Canvas.Font.Name := 'Arial';
   Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y,'This is the end of the line' );
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo( 10, 10 );
   Canvas->LineTo( 100, 100 );
   Canvas->Brush->Color = clBtnFace;
   Canvas->Font->Name = "Arial";
   Canvas->TextOut( Canvas->PenPos.x, Canvas->PenPos.y,"This is the end of the line" );
}
```

TCanvas is used everywhere in CLX that drawing is required or possible, and makes drawing graphics both fail-safe and easy.

See *TCanvas* in the online help reference for a complete listing of properties and methods.

Printing

The *TPrinter* object is a paint device that paints on a printer. It generates postscript and sends that to lpr, lp, or another print command.

To get a list of installed and available printers, use the *Printers* property. *TPrinter* uses a *TCanvas* (which is identical to the form's *TCanvas*). This means that anything that can be drawn on a form can be printed as well. To print an image, call the *BeginDoc* method followed by whatever canvas graphics you want to print (including text through the *TextOut* method) and send the job to the printer by calling the *EndDoc* method.

This example uses a button and a memo on a form. When the user clicks the button, the content of the memo is printed with a 200-pixel border around the page.

To run this example successfully, add QPrinters to your uses clause (Delphi) or include <QPrinters.hpp> in your unit file (C++).

D Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRect;
  i: Integer;
begin
  with Printer do
    begin
      r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
      BeginDoc;
      Canvas.Brush.Style := bsClear;
      for i := 0 to Memo1.Lines.Count do
       Canvas.TextOut(200,200 + (i *
                    Canvas.TextHeight(Memo1.Lines.Strings[i])),
                    Memo1.Lines.Strings[i]);
      Canvas.Brush.Color := clBlack;
      Canvas.FrameRect(r);
      EndDoc;
    end;
```

```
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TPrinter *Prntr = Printer();
  TRect r = Rect(200,200,Prntr->PageWidth - 200,Prntr->PageHeight- 200);
  Prntr->BeginDoc();
  Canvas->Brush->Style := bsClear;
  for( int i = 0; i < Memo1->Lines->Count; i++)
    Prntr->Canvas->TextOut(200,200 + (i *
  Prntr->Canvas->TextHeight(Memo1->Lines->Strings[i])),
  Memo1->Lines->Strings[i]);
  Prntr->Canvas->Brush->Color = clBlack;
  Prntr->Canvas->FrameRect(r);
  Prntr->EndDoc();
}
```

For more information on the use of the *TPrinter* object, look in the online help under *TPrinter*.

Converting measurements

The ConvUtils unit declares a general-purpose *Convert* function that you can use to convert a measurement from one set of units to another. You can perform conversions between compatible units of measurement such as feet and inches or days and weeks. Units that measure the same types of things are said to be in the same *conversion family*. The units you're converting must be in the same conversion family. For information on doing conversions, see "Performing conversions" on page 5-33 and refer to *Convert* in the online Help.

The StdConvs unit defines several conversion families and measurement units within each family. In addition, you can create customized conversion families and associated units using the *RegisterConversionType* and *RegisterConversionFamily* functions. For information on extending conversion and conversion units, see "Adding new measurement types" on page 5-34 and refer to Convert in the online Help.

Performing conversions

You can use the *Convert* function to perform both simple and complex conversions. It includes a simple syntax and a second syntax for performing conversions between complex measurement types.

Performing simple conversions

You can use the *Convert* function to convert a measurement from one set of units to another. The *Convert* function converts between units that measure the same type of thing (distance, area, time, temperature, and so on).

To use *Convert*, you must specify the units from which to convert and to which to convert. You use the *TConvType* type to identify the units of measurement.

For example, this converts a temperature from degrees Fahrenheit to degrees Kelvin:

```
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

```
TempInKelvin = Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

Performing complex conversions

You can also use the *Convert* function to perform more complex conversions between the ratio of two measurement types. Examples of when you might need to use this this are when converting miles per hour to meters per minute for calculating speed or when converting gallons per minute to liters per hour for calculating flow.

For example, the following call converts miles per gallon to kilometers per liter:

```
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

```
double nKPL = Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

The units you're converting must be in the same conversion family (they must measure the same thing). If the units are not compatible, *Convert* raises an *EConversionError* exception. You can check whether two *TConvType* values are in the same conversion family by calling *CompatibleConversionTypes*.

The StdConvs unit defines several families of *TConvType* values. See Conversion family variables in the online Help for a list of the predefined families of measurement units and the measurement units in each family.

Adding new measurement types

If you want to perform conversions between measurement units not already defined in the StdConvs unit, you need to create a new conversion family to represent the measurement units (*TConvType* values). When two *TConvType* values are registered with the same conversion family, the *Convert* function can convert between measurements made using the units represented by those *TConvType* values.

You first need to obtain *TConvFamily* values by registering a conversion family using the *RegisterConversionFamily* function. After you get a *TConvFamily* value (by registering a new conversion family or using one of the global variables in the StdConvs unit), you can use the *RegisterConversionType* function to add the new units to the conversion family. The following examples show how to do this.

Creating a simple conversion family and adding units

One example of when you could create a new conversion family and add new measurement types might be when performing conversions between long periods of time (such as months to centuries) where a loss of precision can occur.

To explain this further, the *cbTime* family uses a day as its base unit. The base unit is the one that is used when performing all conversions within that family. Therefore, all conversions must be done in terms of days. An inaccuracy can occur when performing conversions using units of months or larger (months, years, decades, centuries, millennia) because there is not an exact conversion between days and months, days and years, and so on. Months have different lengths; years have correction factors for leap years, leap seconds, and so on.

If you are only using units of measurement greater than or equal to months, you can create a more accurate conversion family with years as its base unit. This example creates a new conversion family called *cbLongTime*.

Declare variables

First, you need to declare variables for the identifiers. The identifiers are used in the new LongTime conversion family, and the units of measurement that are its members:

Delphi example

```
var
   cbLongTime: TConvFamily;
   ltMonths: TConvType;
   ltYears: TConvType;
   ltDecades: TConvType;
   ltCenturies: TConvType;
   ltMillennia: TConvType;
```

C++ example

```
tConvFamily cbLongTime;
TConvType ltMonths;
TConvType ltYears;
TConvType ltDecades;
TConvType ltCenturies;
TConvType ltMillennia;
```

Register the conversion family

Next, register the conversion family:

```
cbLongTime := RegisterConversionFamily ('Long Times');
```

```
cbLongTime = RegisterConversionFamily ("Long Times");
```

Although an *UnregisterConversionFamily* procedure is provided, you don't need to unregister conversion families unless the unit that defines them is removed at runtime. They are automatically cleaned up when your application shuts down.

Register measurement units

Next, you need to register the measurement units within the conversion family that you just created. You use the *RegisterConversionType* function, which registers units of measurement within a specified family. You need to define the base unit which in the example is years, and the other units are defined using a factor that indicates their relation to the base unit. So, the factor for *ltMonths* is 1/12 because the base unit for the LongTime family is years. You also include a description of the units to which you are converting.

The code to register the measurement units is shown here:

Delphi example

```
ltMonths:=RegisterConversionType(cbLongTime,'Months',1/12);
ltYears:=RegisterConversionType(cbLongTime,'Years',1);
ltDecades:=RegisterConversionType(cbLongTime,'Decades',10);
ltCenturies:=RegisterConversionType(cbLongTime,'Centuries',100);
ltMillennia:=RegisterConversionType(cbLongTime,'Millennia',1000);
```

C++ example

```
ltMonths = RegisterConversionType(cbLongTime,"Months",1/12);
ltYears = RegisterConversionType(cbLongTime,"Years",1);
ltDecades = RegisterConversionType(cbLongTime,"Decades",10);
ltCenturies = RegisterConversionType(cbLongTime,"Centuries",100);
ltMillennia = RegisterConversionType(cbLongTime,"Millennia",1000);
```

Use the new units

You can now use the newly registered units to perform conversions. The global *Convert* function can convert between any of the conversion types that you registered with the *cbLongTime* conversion family.

So instead of using the following *Convert* call,

```
Convert(StrToFloat(Edit1.Text),tuMonths,tuMillennia);
```

```
Convert(StrToFloat(Edit1->Text),tuMonths,tuMillennia);
```

you can now use this one for greater accuracy:

```
Convert(StrToFloat(Edit1.Text),ltMonths,ltMillennia);
```

```
Convert(StrToFloat(Edit1->Text),ltMonths,ltMillennia);
```

Using a conversion function

For cases when the conversion is more complex, you can use a different syntax to specify a function to perform the conversion instead of using a conversion factor. For example, you can't convert temperature values using a conversion factor, because different temperature scales have a different origins.

This example, which comes from the StdConvs unit, shows how to register a conversion type by providing functions to convert to and from the base units. (The C++ version is a translation of the code in the StdConvs unit.)

Declare variables

First, declare variables for the identifiers. The identifiers are used in the *cbTemperature* conversion family, and the units of measurement are its members:

Delphi example

```
var
```

```
cbTemperature: TConvFamily;
tuCelsius: TConvType;
tuKelvin: TConvType;
tuFahrenheit: TConvType;
```

C++ example

```
TConvFamily cbTemperature;
TConvType tuCelsius;
TConvType tuKelvin;
TConvType tuFahrenheit;
```

Note The units of measurement listed here are a subset of the temperature units actually registered in the *StdConvs* unit.

Register the conversion family

Next, register the conversion family:

```
cbTemperature := RegisterConversionFamily ('Temperature');
```

```
cbTemperature = RegisterConversionFamily ("Temperature");
```

Register the base unit

Next, define and register the base unit of the conversion family, which in the example is degrees Celsius. Note that in the case of the base unit, we can use a simple conversion factor, because there is no actual conversion to make:

```
tuCelsius:=RegisterConversionType(cbTemperature,'Celsius',1);
```

```
tuCelsius = RegisterConversionType(cbTemperature,"Celsius",1);
```

Write methods to convert to and from the base unit

You need to write the code that performs the conversion from each temperature scale to and from degrees Celsius, because these do not rely on a simple conversion factor. These functions are taken from the StdConvs unit:

Delphi example

```
function FahrenheitToCelsius(const AValue: Double): Double;
begin
  Result := ((AValue - 32) * 5) / 9;
end;

function CelsiusToFahrenheit(const AValue: Double): Double;
begin
  Result := ((AValue * 9) / 5) + 32;
end;

function KelvinToCelsius(const AValue: Double): Double;
begin
  Result := AValue - 273.15;
end;

function CelsiusToKelvin(const AValue: Double): Double;
begin
```

```
  Result := AValue + 273.15;
end;
```

C++ example

```
double __fastcall FahrenheitToCelsius(const double AValue)
{
  return (((AValue - 32) * 5) / 9);
}
double __fastcall CelsiusToFahrenheit(const double AValue)
{
  return (((AValue * 9) / 5) + 32);
}
double __fastcall KelvinToCelsius(const double AValue)
{
  return (AValue - 273.15);
}
double __fastcall CelsiusToKelvin(const double AValue)
{
  return (AValue + 273.15);
}
```

Register the other units

Now that you have the conversion functions, you can register the other measurement units within the conversion family. You also include a description of the units.

The code to register the other units in the family is shown here:

Delphi example

```
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius,
CelsiusToKelvin);
  tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit', FahrenheitToCelsius,
CelsiusToFahrenheit);
```

C++ example

```
tuKelvin = RegisterConversionType(cbTemperature, "Kelvin", KelvinToCelsius,
CelsiusToKelvin);
  tuFahrenheit = RegisterConversionType(cbTemperature, "Fahrenheit", FahrenheitToCelsius,
CelsiusToFahrenheit);
```

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the conversion types that you registered with the *cbTemperature* conversion family. For example the following code converts a value from degrees Fahrenheit to degrees Kelvin.

```
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

Using a class to manage conversions

You can always use conversion functions to register a conversion unit. There are times, however, when this requires you to create an unnecessarily large number of functions that all do essentially the same thing.

If you can write a set of conversion functions that differ only in the value of a parameter or variable, you can create a class to handle those conversions. For example, there is a set standard techniques for converting between the various European currencies since the introduction of the Euro. Even though the conversion factors remain constant (unlike the conversion factor between, say, dollars and Euros), you can't use a simple conversion factor approach to properly convert between European currencies for two reasons:

- The conversion must round to a currency-specific number of digits.
- The conversion factor approach uses an inverse factor to the one specified by the standard Euro conversions.

However, this can all be handled by the conversion functions such as the following:

Delphi example

```
function FromEuro(const AValue: Double, Factor; FRound: TRoundToRange): Double;
begin
  Result := RoundTo(AValue * Factor, FRound);
end;

function ToEuro(const AValue: Double, Factor): Double;
begin
  Result := AValue / Factor;
end;
```

C++ example

```
double __fastcall FromEuro(const double AValue, const double Factor, TRoundToRange FRound)
{
  return(RoundTo(AValue * Factor, FRound));
}

double __fastcall ToEuro(const double AValue, const double Factor)
{
  return (AValue / Factor);
}
```

The problem is, this approach requires extra parameters on the conversion function, which means you can't simply register the same function with every European currency. In order to avoid having to write two new conversion functions for every European currency, you can make use of the same two functions by making them the members of a class.

Creating the conversion class

The class must be a descendant of *TConvTypeFactor*. *TConvTypeFactor* defines two methods, *ToCommon* and *FromCommon*, for converting to and from the base units of a conversion family (in this case, to and from Euros). Just as with the functions you use directly when registering a conversion unit, these methods have no extra parameters, so you must supply the number of digits to round off and the conversion factor as private members of your conversion class:

Delphi example

```
type
  TConvTypeEuroFactor = class(TConvTypeFactor)
  private
    FRound: TRoundToRange;
  public
    constructor Create(const AConvFamily: TConvFamily;
      const ADescription: string; const AFactor: Double;
      const ARound: TRoundToRange);
    function ToCommon(const AValue: Double): Double; override;
    function FromCommon(const AValue: Double): Double; override;
  end;
end;
```

C++ example

```
class PASCALIMPLEMENTATION TConvTypeEuroFactor : public Convutils::TConvTypeFactor
{
  private:
    TRoundToRange FRound;
  public:
    __fastcall TConvTypeEuroFactor(const TConvFamily AConvFamily,
        const AnsiString ADescription, const double AFactor, const TRoundToRange ARound);
              TConvTypeFactor(AConvFamily, ADescription, AFactor);
    virtual double ToCommon(const double AValue);
    virtual double FromCommon(const double AValue);
}
```

The constructor assigns values to those private members:

Delphi example

```
constructor TConvTypeEuroFactor.Create(const AConvFamily: TConvFamily;
  const ADescription: string; const AFactor: Double;
  const ARound: TRoundToRange);
begin
  inherited Create(AConvFamily, ADescription, AFactor);
  FRound := ARound;
end;
```

C++ example

```
__fastcall TConvTypeEuroFactor::TConvTypeEuroFactor(const TConvFamily AConvFamily,
        const AnsiString ADescription, const double AFactor, const TRoundToRange ARound):
               TConvTypeFactor(AConvFamily, ADescription, AFactor);
{
  FRound = ARound;
}
```

The two conversion functions simply use these private members:

Delphi example

```
function TConvTypeEuroFactor.FromCommon(const AValue: Double): Double;
begin
  Result := RoundTo(AValue * Factor, FRound);
end;

function TConvTypeEuroFactor.ToCommon(const AValue: Double): Double;
begin
  Result := AValue / Factor;
end;
```

C++ example

```
virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (RoundTo(AValue * Factor, FRound));
}

virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (AValue / Factor);
}
```

Declare variables

Now that you have a conversion class, begin as with any other conversion family, by declaring identifiers:

Delphi example

```
var
  euEUR: TConvType; { EU euro }
  euBEF: TConvType; { Belgian francs }
  euDEM: TConvType; { German marks }
  euGRD: TConvType; { Greek drachmas }
  euESP: TConvType; { Spanish pesetas }
  euFFR: TConvType; { French francs }
  euIEP: TConvType; { Irish pounds }
  euITL: TConvType; { Italian lire }
  euLUF: TConvType; { Luxembourg francs }
  euNLG: TConvType; { Dutch guilders }
  euATS: TConvType; { Austrian schillings }
  euPTE: TConvType; { Portuguese escudos }
  euFIM: TConvType; { Finnish marks }
  cbEuro: TConvFamily;
```

C++ example

```
TConvFamily cbEuro;
TConvType euEUR; // EU euro
TConvType euBEF; // Belgian francs
TConvType euDEM; // German marks
TConvType euGRD; // Greek drachmas
TConvType euESP; // Spanish pesetas
TConvType euFFR; // French francs
TConvType euIEP; // Irish pounds
TConvType euITL; // Italian lire
TConvType euLUF; // Luxembourg francs
TConvType euNLG; // Dutch guilders
TConvType euATS; // Austrian schillings
TConvType euPTE; // Protuguese escudos
TConvType euFIM; // Finnish marks
```

Register the conversion family and the other units

Now you are ready to register the conversion family and the European monetary units, using your new conversion class. Register the conversion family the same way you registered the other conversion families:

```
cbEuro := RegisterConversionFamily ('European currency');
```

```
cbEuro = RegisterConversionFamily ("European currency");
```

To register each conversion type, create an instance of the conversion class that reflects the factor and rounding properties of that currency, and call the RegisterConversionType method:

Delphi example

```
var
  LInfo: TConvTypeInfo;
begin
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'EUEuro', 1.0, -2);
  if not RegisterConversionType(LInfo, euEUR) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'BelgianFrancs', 40.3399, 0);
  if not RegisterConversionType(LInfo, euBEF) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'GermanMarks', 1.95583, -2);
  if not RegisterConversionType(LInfo, euDEM) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'GreekDrachmas', 340.75, 0);
  if not RegisterConversionType(LInfo, euGRD) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'SpanishPesetas', 166.386, 0);
  if not RegisterConversionType(LInfo, euESP) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'FrenchFrancs', 6.55957, -2);
  if not RegisterConversionType(LInfo, euFFR) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'IrishPounds', 0.787564, -2);
  if not RegisterConversionType(LInfo, euIEP) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'ItalianLire', 1936.27, 0);
  if not RegisterConversionType(LInfo, euITL) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'LuxembourgFrancs', 40.3399, -2);
  if not RegisterConversionType(LInfo, euLUF) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'DutchGuilders', 2.20371, -2);
  if not RegisterConversionType(LInfo, euNLG) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'AustrianSchillings', 13.7603, -2);
  if not RegisterConversionType(LInfo, euATS) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'PortugueseEscudos', 200.482, -2);
  if not RegisterConversionType(LInfo, euPTE) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'FinnishMarks', 5.94573, 0);
  if not RegisterConversionType(LInfo, euFIM) then
    LInfo.Free;
end;
```

C++ example

```
TConvTypeInfo *pInfo = new TConvTypeEuroFactor(cbEuro, "EUEuro", 1.0, -2);
if (!RegisterConversionType(pInfo, euEUR))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "BelgianFrancs", 40.3399, 0);
if (!RegisterConversionType(pInfo, euBEF))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "GermanMarks", 1.95583, -2);
if (!RegisterConversionType(pInfo, euDEM))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "GreekDrachmas", 340.75, 0);
if (!RegisterConversionType(pInfo, euGRD)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "SpanishPesetas", 166.386, 0);
if (!RegisterConversionType(pInfo, euESP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FrenchFrancs", 6.55957, -2);
if (!RegisterConversionType(pInfo, euFFR)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "IrishPounds", 0.787564, -2);
if (!RegisterConversionType(pInfo, euIEP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "ItalianLire", 1936.27, 0);
if (!RegisterConversionType(pInfo, euITL)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "LuxembourgFrancs", 40.3399, -2);
if (!RegisterConversionType(pInfo, euLUF)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "DutchGuilders", 2.20371, -2);
if (!RegisterConversionType(pInfo, euNLG)
  delete pInfo;
```

```
pInfo = new TConvTypeEuroFactor(cbEuro, "AutstrianSchillings", 13.7603, -2);
if (!RegisterConversionType(pInfo, euATS)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "PortugueseEscudos", 200.482, -2);
if (!RegisterConversionType(pInfo, euPTE)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FinnishMarks", 5.94573, 0);
if (!RegisterConversionType(pInfo, euFIM)
  delete pInfo;
```

Note The ConvertIt demo provides an expanded version of this example that includes other currencies (that do not have fixed conversion rates) and more error checking.

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the European currencies you have registered with the new cbEuro family. For example, the following code converts a value from Italian Lire to German Marks:

```
Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));
```

```
Edit2->Text = FloatToStr(Convert(StrToFloat(Edit1->Text), euITL, euDEM));
```

Defining custom variants in Delphi

One powerful built-in type of the Delphi language is the Variant type. Variants represent values whose type is not determined at compile time. Instead, the type of their value can change at runtime. Variants can mix with other variants and with integer, real, string, and boolean values in expressions and assignments; the compiler automatically performs type conversions.

By default, variants can't hold values that are records, sets, static arrays, files, classes, class references, or pointers. You can, however, extend the Variant type to work with any particular example of these types. All you need to do is create a descendant of the *TCustomVariantType* class that indicates how the Variant type performs standard operations.

Note Defining custom Variants is not fully supported in the C++ language.

To create a Variant type in Delphi:

1. Map the storage of the variant's data on to the *TVarData* record.
2. Declare a class that descends from *TCustomVariantType*. Implement all required behavior (including type conversion rules) in the new class.
3. Write utility methods for creating instances of your custom variant and recognizing its type.

The above steps extend the Variant type so that the standard operators work with your new type and the new Variant type can be cast to other data types. You can further enhance your new Variant type so that it supports properties and methods that you define. When creating a Variant type that supports properties or methods, use *TInvokeableVariantType* or *TPublishableVariantType* as a base class rather than *TCustomVariantType*.

Storing a custom variant type's data

Variants store their data in the *TVarData* record type. This type is a record that contains 16 bytes. The first word indicates the type of the variant, and the remaining 14 bytes are available to store the data. While your new Variant type can work directly with a *TVarData* record, it is usually easier to define a record type whose members have names that are meaningful for your new type, and cast that new type onto the *TVarData* record type.

For example, the VarConv unit defines a custom variant type that represents a measurement. The data for this type includes the units (*TConvType*) of measurement, as well as the value (a double). The VarConv unit defines its own type to represent such a value:

```
TConvertVarData = packed record
  VType: TVarType;
  VConvType: TConvType;
  Reserved1, Reserved2: Word;
  VValue: Double;
end;
```

This type is exactly the same size as the *TVarData* record. When working with a custom variant of the new type, the variant (or its *TVarData* record) can be cast to *TConvertVarData*, and the custom Variant type simply works with the *TVarData* record as if it were a *TConvertVarData* type.

Note When defining a record that maps onto the *TVarData* record in this way, be sure to define it as a packed record.

If your new custom Variant type needs more than 14 bytes to store its data, you can define a new record type that includes a pointer or object instance. For example, the VarCmplx unit uses an instance of the class *TComplexData* to represent the data in a complex-valued variant. It therefore defines a record type the same size as *TVarData* that includes a reference to a *TComplexData* object:

```
TComplexVarData = packed record
  VType: TVarType;
  Reserved1, Reserved2, Reserved3: Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

Object references are actually pointers (two Words), so this type is the same size as the *TVarData* record. As before, a complex custom variant (or its *TVarData* record), can be cast to *TComplexVarData*, and the custom variant type works with the *TVarData* record as if it were a *TComplexVarData* type.

Creating a class to enable the custom variant type

Custom variants work by using a special helper class that indicates how variants of the custom type can perform standard operations. You create this helper class by writing a descendant of *TCustomVariantType*. This involves overriding the appropriate virtual methods of *TCustomVariantType*.

Enabling casting

One of the most important features of the custom variant type for you to implement is typecasting. The flexibility of variants arises, in part, from their implicit typecasts.

There are two methods for you to implement that enable the custom Variant type to perform typecasts: *Cast*, which converts another variant type to your custom variant, and *CastTo*, which converts your custom Variant type to another type of Variant.

When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from which you are casting may be another custom Variant type. To handle this situation, you can try casting to one of the built-in Variant types as an intermediate step.

For example, the following *Cast* method, from the *TComplexVariantType* class uses the type Double as an intermediate type:

```
procedure TComplexVariantType.Cast(var Dest: TVarData; const Source: TVarData);
var
  LSource, LTemp: TVarData;
begin
  VarDataInit(LSource);
  try
    VarDataCopyNoInd(LSource, Source);
    if VarDataIsStr(LSource) then
      TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
    else
    begin
      VarDataInit(LTemp);
      try
        VarDataCastTo(LTemp, LSource, varDouble);
        TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
      finally
        VarDataClear(LTemp);
      end;
    end;
    Dest.VType := VarType;
  finally
    VarDataClear(LSource);
  end;
end;
```

In addition to the use of Double as an intermediate Variant type, there are a few things to note in this implementation:

- The last step of this method sets the *VType* member of the returned *TVarData* record. This member gives the Variant type code. It is set to the *VarType* property of *TComplexVariantType*, which is the Variant type code assigned to the custom variant.
- The custom variant's data (*Dest*) is typecast from *TVarData* to the record type that is actually used to store its data (*TComplexVarData*). This makes the data easier to work with.
- The method makes a local copy of the source variant rather than working directly with its data. This prevents side effects that may affect the source data.

When casting from a complex variant to another type, the *CastTo* method also uses an intermediate type of Double (for any destination type other than a string):

```
procedure TComplexVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
  const AVarType: TVarType);
var
  LTemp: TVarData;
begin
  if Source.VType = VarType then
    case AVarType of
      varOleStr:
        VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
      varString:
        VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
    else
      VarDataInit(LTemp);
      try
        LTemp.VType := varDouble;
        LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
        VarDataCastTo(Dest, LTemp, AVarType);
      finally
        VarDataClear(LTemp);
      end;
    end
  else
    RaiseCastError;
end;
```

Note that the *CastTo* method includes a case where the source variant data does not have a type code that matches the *VarType* property. This case only occurs for empty (unassigned) source variants.

Implementing binary operations

To allow the custom variant type to work with standard binary operators (+, -, *, /, div, mod, shl, shr, and, or, xor listed in the System unit), you must override the *BinaryOp* method. *BinaryOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the operator. Implement this method to perform the operation and return the result using the same variable that contained the left-hand operand.

For example, the following *BinaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
  const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Left.VType of
      varString:
        case Operator of
          opAdd: Variant(Left) := Variant(Left) + TComplexVarData(Right).VComplex.AsString;
        else
          RaiseInvalidOp;
        end;
      else
        if Left.VType = VarType then
          case Operator of
            opAdd:
              TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
            opSubtract:
              TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
            opMultiply:
              TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
            opDivide:
              TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
            else
              RaiseInvalidOp;
          end
        else
          RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

There are several things to note in this implementation:

This method only handles the case where the variant on the right side of the operator is a custom variant that represents a complex number. If the left-hand operand is a complex variant and the right-hand operand is not, the complex variant forces the right-hand operand first to be cast to a complex variant. It does this by overriding the *RightPromotion* method so that it always requires the type in the *VarType* property:

```
function TComplexVariantType.RightPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { Complex Op TypeX }
  RequiredVarType := VarType;
  Result := True;
end;
```

The addition operator is implemented for a string and a complex number (by casting the complex value to a string and concatenating), and the addition, subtraction, multiplication, and division operators are implemented for two complex numbers using the methods of the *TComplexData* object that is stored in the complex variant's data. This is accessed by casting the *TVarData* record to a *TComplexVarData* record and using its *VComplex* member.

Attempting any other operator or combination of types causes the method to call the *RaiseInvalidOp* method, which causes a runtime error. The *TCustomVariantType* class includes a number of utility methods such as *RaiseInvalidOp* that can be used in the implementation of custom variant types.

BinaryOp only deals with a limited number of types: strings and other complex variants. It is possible, however, to perform operations between complex numbers and other numeric types. For the *BinaryOp* method to work, the operands must be cast to complex variants before the values are passed to this method. We have already seen (above) how to use the *RightPromotion* method to force the right-hand operand to be a complex variant if the left-hand operand is complex. A similar method, *LeftPromotion*, forces a cast of the left-hand operand when the right-hand operand is complex:

```
function TComplexVariantType.LeftPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { TypeX Op Complex }
  if (Operator = opAdd) and VarDataIsStr(V) then
    RequiredVarType := varString
  else
    RequiredVarType := VarType;
  Result := True;
end;
```

This *LeftPromotion* method forces the left-hand operand to be cast to another complex variant, unless it is a string and the operation is addition, in which case *LeftPromotion* allows the operand to remain a string.

Implementing comparison operations

There are two ways to enable a custom variant type to support comparison operators (=, <>, <, <=, >, >=). You can override the *Compare* method, or you can override the *CompareOp* method.

The *Compare* method is easiest if your custom variant type supports the full range of comparison operators. *Compare* takes three parameters: the left-hand operand, the right-hand operand, and a var Parameter that returns the relationship between the two. For example, the *TConvertVariantType* object in the VarConv unit implements the following *Compare* method:

```
procedure TConvertVariantType.Compare(const Left, Right: TVarData;
  var Relationship: TVarCompareResult);
const
  CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
    (crLessThan, crEqual, crGreaterThan);
var
  LValue: Double;
  LType: TConvType;
  LRelationship: TValueRelationship;
begin
  // supports...
  //   convvar cmp number
  //     Compare the value of convvar and the given number
  //   convvar1 cmp convvar2
  //     Compare after converting convvar2 to convvar1's unit type
  //  The right can also be a string.  If the string has unit info then it is
  //    treated like a varConvert else it is treated as a double
  LRelationship := EqualsValue;
  case Right.VType of
    varString:
      if TryStrToConvUnit(Variant(Right), LValue, LType) then
        if LType = CIllegalConvType then
          LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
        else
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                                 TConvertVarData(Left).VConvType, LValue, LType)
        else
          RaiseCastError;
    varDouble:
      LRelationship := CompareValue(TConvertVarData(Left).VValue, TVarData(Right).VDouble);
    else
      if Left.VType = VarType then
        LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                    TConvertVarData(Left).VConvType, TConvertVarData(Right).VValue,
                    TConvertVarData(Right).VConvType)
      else
        RaiseInvalidOp;
  end;
  Relationship := CRelationshipToRelationship[LRelationship];
end;
```

If the custom type does not support the concept of “greater than” or “less than,” only “equal” or “not equal,” however, it is difficult to implement the *Compare* method, because *Compare* must return *crLessThan*, *crEqual*, or *crGreaterThan*. When the only valid response is “not equal,” it is impossible to know whether to return *crLessThan* or *crGreaterThan*. Thus, for types that do not support the concept of ordering, you can override the *CompareOp* method instead.

CompareOp has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the comparison operator. Implement this method to perform the operation and return a boolean that indicates whether the comparison is true. You can then call the *RaiseInvalidOp* method when the comparison makes no sense.

For example, the following *CompareOp* method comes from the *TComplexVariantType* object in the VarCmplx unit. It supports only a test of equality or inequality:

```
function TComplexVariantType.CompareOp(const Left, Right: TVarData;
  const Operator: Integer): Boolean;
begin
  Result := False;
  if (Left.VType = VarType) and (Right.VType = VarType) then
    case Operator of
      opCmpEQ:
        Result := TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
      opCmpNE:
```

```
      Result := not TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Note that the types of operands that both these implementations support are very limited. As with binary operations, you can use the *RightPromotion* and *LeftPromotion* methods to limit the cases you must consider by forcing a cast before *Compare* or *CompareOp* is called.

Implementing unary operations

To allow the custom variant type to work with standard unary operators (-, not), you must override the *UnaryOp* method. *UnaryOp* has two parameters: the value of the operand and the operator. Implement this method to perform the operation and return the result using the same variable that contained the operand.

For example, the following *UnaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.UnaryOp(var Right: TVarData; const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Operator of
      opNegate:
        TComplexVarData(Right).VComplex.DoNegate;
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Note that for the logical **not** operator, which does not make sense for complex values, this method calls *RaiseInvalidOp* to cause a runtime error.

Copying and clearing custom variants

In addition to typecasting and the implementation of operators, you must indicate how to copy and clear variants of your custom Variant type.

To indicate how to copy the variant’s value, implement the *Copy* method. Typically, this is an easy operation, although you must remember to allocate memory for any classes or structures you use to hold the variant’s value:

```
procedure TComplexVariantType.Copy(var Dest: TVarData; const Source: TVarData;
  const Indirect: Boolean);
begin
  if Indirect and VarDataIsByRef(Source) then
    VarDataCopyNoInd(Dest, Source)
  else
```

```
    with TComplexVarData(Dest) do
    begin
      VType := VarType;
      VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
    end;
end;
```

Note The *Indirect* parameter in the *Copy* method signals that the copy must take into account the case when the variant holds only an indirect reference to its data.

Tip If your custom variant type does not allocate any memory to hold its data (if the data fits entirely in the *TVarData* record), your implementation of the Copy method can simply call the *SimplisticCopy* method.

To indicate how to clear the variant's value, implement the *Clear* method. As with the Copy method, the only tricky thing about doing this is ensuring that you free any resources allocated to store the variant's data:

```
procedure TComplexVariantType.Clear(var V: TVarData);
begin
  V.VType := varEmpty;
  FreeAndNil(TComplexVarData(V).VComplex);
end;
```

You will also need to implement the *IsClear* method. This way, you can detect any invalid values or special values that represent "blank" data:

```
function TComplexVariantType.IsClear(const V: TVarData): Boolean;
begin
  Result := (TComplexVarData(V).VComplex = nil) or
            TComplexVarData(V).VComplex.IsZero;
end;
```

Loading and saving custom variant values

By default, when the custom variant is assigned as the value of a published property, it is typecast to a string when that property is saved to a form file, and converted back from a string when the property is read from a form file. You can, however, provide your own mechanism for loading and saving custom variant values in a more natural representation. To do so, the *TCustomVariantType* descendant must implement the *IVarStreamable* interface from Classes.pas.

IVarStreamable defines two methods, *StreamIn* and *StreamOut*, for reading a variant's value from a stream and for writing the variant's value to the stream. For example, *TComplexVariantType*, in the VarCmplx unit, implements the *IVarStreamable* methods as follows:

```
procedure TComplexVariantType.StreamIn(var Dest: TVarData; const Stream: TStream);
begin
  with TReader.Create(Stream, 1024) do
  try
    with TComplexVarData(Dest) do
    begin
      VComplex := TComplexData.Create;
      VComplex.Real := ReadFloat;
      VComplex.Imaginary := ReadFloat;
    end;
  finally
    Free;
  end;
end;

procedure TComplexVariantType.StreamOut(const Source: TVarData; const Stream: TStream);
begin
  with TWriter.Create(Stream, 1024) do
  try
    with TComplexVarData(Source).VComplex do
    begin
      WriteFloat(Real);
      WriteFloat(Imaginary);
    end;
  finally
    Free;
  end;
end;
```

Note how these methods create a Reader or Writer object for the *Stream* parameter to handle the details of reading or writing values.

Using the TCustomVariantType descendant

In the initialization section of the unit that defines your *TCustomVariantType* descendant, create an instance of your class. When you instantiate your object, it automatically registers itself with the variant-handling system so that the new Variant type is enabled. For example, here is the initialization section of the VarCmplx unit:

```
initialization
  ComplexVariantType := TComplexVariantType.Create;
```

In the finalization section of the unit that defines your *TCustomVariantType* descendant, free the instance of your class. This automatically unregisters the variant type. Here is the finalization section of the VarCmplx unit:

```
finalization
  FreeAndNil(ComplexVariantType);
```

Writing utilities to work with a custom variant type

Once you have created a *TCustomVariantType* descendant to implement your custom variant type, it is possible to use the new Variant type in applications. However, without a few utilities, this is not as easy as it should be.

It is a good idea to create a method that creates an instance of your custom variant type from an appropriate value or set of values. This function or set of functions fills out the structure you defined to store your custom variant's data. For example, the following function could be used to create a complex-valued variant:

```
function VarComplexCreate(const AReal, AImaginary: Double): Variant;
begin
  VarClear(Result);
  TComplexVarData(Result).VType := ComplexVariantType.VarType;
  TComplexVarData(ADest).VComplex := TComplexData.Create(ARead, AImaginary);
end;
```

This function does not actually exist in the VarCmplx unit, but is a synthesis of methods that do exist, provided to simplify the example. Note that the returned variant is cast to the record that was defined to map onto the *TVarData* structure (*TComplexVarData*), and then filled out.

Another useful utility to create is one that returns the variant type code for your new Variant type. This type code is not a constant. It is automatically generated when you instantiate your *TCustomVariantType* descendant. It is therefore useful to provide a way to easily determine the type code for your custom variant type. The following function from the VarCmplx unit illustrates how to write one, by simply returning the *VarType* property of the *TCustomVariantType* descendant:

```
function VarComplex: TVarType;
begin
  Result := ComplexVariantType.VarType;
end;
```

Two other standard utilities provided for most custom variants check whether a given variant is of the custom type and cast an arbitrary variant to the new custom type. Here is the implementation of those utilities from the VarCmplx unit:

```
function VarIsComplex(const AValue: Variant): Boolean;
begin
  Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;

function VarAsComplex(const AValue: Variant): Variant;
begin
  if not VarIsComplex(AValue) then
    VarCast(Result, AValue, VarComplex)
  else
    Result := AValue;
end;
```

Note that these use standard features of all variants: the *VType* member of the *TVarData* record and the *VarCast* function, which works because of the methods implemented in the *TCustomVariantType* descendant for casting data.

In addition to the standard utilities mentioned above, you can write any number of utilities specific to your new custom variant type. For example, the VarCmplx unit defines a large number of functions that implement mathematical operations on complex-valued variants.

Supporting properties and methods in custom variants

Some variants have properties and methods. For example, when the value of a variant is an interface, you can use the variant to read or write the values of properties on that interface and call its methods. Even if your custom variant type does not represent an interface, you may want to give it properties and methods that an application can use in the same way.

Using TInvokeableVariantType

To provide support for properties and methods, the class you create to enable the new custom variant type should descend from *TInvokeableVariantType* instead of directly from *TCustomVariantType*.

TInvokeableVariantType defines four methods:

- *DoFunction*
- *DoProcedure*
- *GetProperty*
- *SetProperty*

that you can implement to support properties and methods on your custom variant type.

For example, the VarConv unit uses *TInvokeableVariantType* as the base class for *TConvertVariantType* so that the resulting custom variants can support properties. The following example shows the property getter for these properties:

```
function TConvertVariantType.GetProperty(var Dest: TVarData;
  const V: TVarData; const Name: String): Boolean;
var
  LType: TConvType;
begin
  // supports...
  //   'Value'
  //   'Type'
  //   'TypeName'
  //   'Family'
  //   'FamilyName'
  //   'As[Type]'
  Result := True;
  if Name = 'VALUE' then
    Variant(Dest) := TConvertVarData(V).VValue
  else if Name = 'TYPE' then
    Variant(Dest) := TConvertVarData(V).VConvType
  else if Name = 'TYPENAME' then
    Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
  else if Name = 'FAMILY' then
    Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
  else if Name = 'FAMILYNAME' then
    Variant(Dest) := ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
  else if System.Copy(Name, 1, 2) = 'AS' then
  begin
    if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
                         System.Copy(Name, 3, MaxInt), LType) then
      VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).VValue,
                             TConvertVarData(V).VConvType, LType), LType)
    else
      Result := False;
  end
  else
    Result := False;
end;
```

The *GetProperty* method checks the *Name* parameter to determine what property is wanted. It then retrieves the information from the *TVarData* record of the Variant (*V*), and returns it as a Variant (*Dest*). Note that this method supports properties whose names are dynamically generated at runtime (As[Type]), based on the current value of the custom variant.

Similarly, the *SetProperty*, *DoFunction*, and *DoProcedure* methods are sufficiently generic that you can dynamically generate method names, or respond to variable numbers and types of parameters.

Using TPublishableVariantType

If the custom variant type stores its data using an object instance, then there is an easier way to implement properties, as long as they are also properties of the object that represents the variant's data. If you use *TPublishableVariantType* as the base class for your custom variant type, then you need only implement the *GetInstance* method, and all the published properties of the object that represents the variant's data are automatically implemented for the custom variants.

For example, as was seen in "Storing a custom variant type's data" on page 5-45, *TComplexVariantType* stores the data of a complex-valued variant using an instance of *TComplexData*. *TComplexData* has a number of published properties (*Real*, *Imaginary*, *Radius*, *Theta*, and *FixedTheta*), that provide information about the complex value. *TComplexVariantType* descends from *TPublishableVariantType*, and implements the *GetInstance* method to return the *TComplexData* object (in TypInfo.pas) that is stored in a complex-valued variant's *TVarData* record:

```
function TComplexVariantType.GetInstance(const V: TVarData): TObject;
begin
  Result := TComplexVarData(V).VComplex;
end;
```

TPublishableVariantType does the rest. It overrides the *GetProperty* and *SetProperty* methods to use the runtime type information (RTTI) of the *TComplexData* object for getting and setting property values.

Note For *TPublishableVariantType* to work, the object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the {$M+} compiler directive, or descend from *TPersistent*.

Chapter

6

Working with components

<~TOPIC<~HEAD
^# pcbUsingComponents
^A Usingcomponents
^X
VCLStandardComponents;Developingtheapplicationuserinterface;Usingcomponents;
CreatingAndManagingMenus;C++languagesupportforthevcl
^$ Working with components
^+ pcb:000
^K components
^T 2_DESIGN
^C 3
HEAD~>Working with components

Many components are provided in the integrated development environment (IDE) on the Component palette. You select components from the Component palette and drop them onto a form or data module. You design the application's user interface by arranging the visual components such as buttons and list boxes on a form. You can also place nonvisual components such as data access components on either a form or a data module.

At first glance, Galileothe IDE's components appear to be just like any other classes. But there are differences between CLX components and the standard class hierarchies that most programmers work with. Some differences are described here:

- All components descend from *TComponent*.
- Components are most often used as is and are changed through their properties, rather than serving as "base classes" to be subclassed to add or change functionality. When a component is inherited, it is usually to add specific code to existing event handling member functions.
- {bmc IC_C16.bmp} In C++, CLX components can only be allocated on the heap, not on the stack (that is, they *must* be created with the **new** operator).
- Properties of components intrinsically contain runtime type information.

- Components can be added to the Component palette in the IDE and manipulated on a form.

Components often achieve a better degree of encapsulation than is usually found in standard classes. For example, consider the use of a dialog containing a push button. In a Windows program developed using CLX components, when a user clicks on the button, the system generates a WM_LBUTTONDOWN message. The program must catch this message (typically in a **switch** statement, a message map, or a response table) and dispatch it to a routine that will execute in response to the message.

Most <~JMPWindows messages (WinCLX)~!ALink(HandlingMessages,1) JMP~> or Linux system events (VisualCLX) are handled by CLX components. When you want to respond to a message or system event, you only need to provide an event handler.

Chapter 9, "Developing the application user interface," provides details on using forms such as creating modal forms dynamically, passing parameters to forms, and retrieving data from forms.

TOPIC~>

Setting component properties

<~TOPIC<~HEAD
^# IDH_USEOP_settingComponentProperties
^$ Setting component properties
^A Setting component properties;SettingComponentProperties
^X Using the Object Inspector;Setting properties at runtime
^K properties:setting
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Setting component properties

To set published properties at design time, you can use the Object Inspector and, in some cases, special property editors. To set properties at runtime, assign their values in your application source code.

For information about the properties of each component, see the online Help.

TOPIC~>

Setting properties at design time

<~TOPIC<~HEAD
^# IDH_USEOP_propertyDisplay
^$ Setting properties at design time
^A Using the Object Inspector;
^X Using property editors
^K Object Inspector;published properties, setting
^+pcb:000

^T 2_DESIGN
^C 3
HEAD~>Setting properties at design time
When you select a component on a form at design time, the Object Inspector displays its published properties and (when appropriate) allows you to edit them. Use the *Tab* key to toggle between the left-hand Property column and the right-hand Value column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column.

If a plus (+) symbol appears next to a property name, clicking the plus symbol or typing '+' when the property has focus displays a list of subvalues for the property. Similarly, if a minus (-) symbol appears next to the property name, clicking the minus symbol or typing '-' hides the subvalues.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the Object Inspector and choose View. For more information, see <~JMPProperty and event categories in the Object Inspector~!ALink(PropertyAndEventCategoriesInTheObjectInspector,1) JMP~>"property categories" in the online Help.

When more than one component is selected, the Object Inspector displays all properties—except *Name*—that are shared by the selected components. If the value for a shared property differs among the selected components, the Object Inspector displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

Changing code-related properties, such as the name of an event handler, in the Object Inspector automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, is immediately reflected in the Object Inspector.

TOPIC~>

Using property editors

<~TOPIC<~HEAD
^# IDH_USEOP_usingPropertyEditors
^$ Using property editors
^A Using property editors;
^K property editors
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Using property editors
Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the Object Inspector. To open the property editor, double-click in the Value column, click the ellipsis mark, or type *Ctrl+Enter* when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor.

Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.

TOPIC~>

Setting properties at runtime

<~TOPIC<~HEAD
^# IDH_USEOP_settingPropertiesAtRuntime
^$ Setting properties at runtime
^A Setting properties at runtime;
^X Setting component properties
^K properties:setting
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Setting properties at runtime
Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

{bmc IC_D16.bmp}

```
Form1.Caption := MyString;
```

{bmc IC_C16.bmp}

```
Form1->Caption = MyString;
```

TOPIC~>

Calling methods

<~TOPIC<~HEAD
^# IDH_USEOP_callingMethods
^$ Calling methods
^A Calling methods;
^X UsingObjectPascalWithTheVCL
^K methods
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Calling methods

Methods are called just like ordinary procedures and functions. For example, visual controls have a *Repaint* method that refreshes the control's image on the screen. You could call the *Repaint* method in a draw-grid object like this:

{bmc IC_D16.bmp}

```
DrawGrid1.Repaint;
```

{bmc IC_C16.bmp}

```
DrawGrid1->Repaint;
```

As with properties, the scope of a method name determines the need for qualifiers. If you want, for example, to repaint a form within an event handler of one of the form's child controls, you don't have to prepend the name of the form to the method call:

{bmc IC_D16.bmp}

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Repaint;
end;
```

For more information about scope in Delphi, see "Scope and qualifiers" on page 4-5<~JMP Scope and qualifiers~!AL(Scope and qualifiers,1) JMP~>.

{bmc IC_C16.bmp}

C++ example

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Repaint;
}
```

TOPIC~>

Working with events and event handlers

<~TOPIC<~HEAD
^# IDH_USEOP_workingWithEventHandlers
^$ Working with events and event handlers
^A Working with events and event handlers;eventhandlers
^X UsingObjectPascalWithTheVCL
^K events:handling;event handlers
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Working with events and event handlers

In GalileoKylix, almost all the code you write is executed, directly or indirectly, in response to *events*. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is a procedure (Delphi) or a method of an object (C++). The sections that follow show how to:

- <~JMP Generate a new event handler.~!AL(Generating a new event handler,1) JMP~>
- <~JMP Generate a handler for a component's default event.~!AL(Generating a handler for a components default event,1) JMP~>
- <~JMP Locate event handlers.~!AL(Locating event handlers,1) JMP~>

- <~JMP Associate an event with an existing event handler.~!AL(Associating an event with an existing event handler,1) JMP~>
- <~JMP Associate menu events with event handlers.~!AL(Associating menu events with event handlers,1) JMP~>
- <~JMP Delete event handlers.~!AL(Deleting event handlers,1) JMP~>

TOPIC~>

Generating a new event handler

<~TOPIC<~HEAD
^# IDH_USEOP_generatingANewEventHandler
^$ Generating a new event handler
^A Generating a new event handler;eventhandlers

^X Working with events and event handlers
^K events:handling;event handlers;skeleton code
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Generating a new event handler
GalileoKylix can generate skeleton event handlers for forms and other components. To create an event handler,

1 Select a component.
2 Click the Events tab in the Object Inspector. The Events page of the Object Inspector displays all events defined for the component.
3 Select the event you want, then double-click the Value column or press *Ctrl+Enter*. The Code editor opens with the cursor inside the skeleton event handler.
4 Type the code that you want to execute when the event occurs.

TOPIC~>

Generating a handler for a component's default event

<~TOPIC<~HEAD
^# IDH_USEOP_generatingTheDefaultEventHandler
^$ Generating a handler for a component's default event
^A Generating a handler for a components default event;eventhandlers
^X Working with events and event handlers
^K events:default;events:handling;event handlers
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Generating a handler for a component's default event
Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button's default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this

generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code.

Not all components have a default event. Some components, such as *TBevel*, don't respond to any events. Other components respond differently when you double-click them in the Form Designer. For example, many components open a default property editor or other dialog when they are double-clicked at design time.

TOPIC~>

Locating event handlers

<~TOPIC<~HEAD
^# IDH_USEOP_locatingEventHandlers
^$ Locating event handlers
^A Locating event handlers;eventhandlers
^X Working with events and event handlers
^K events:handling;event handlers
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Locating event handlers
If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.

To locate an event handler that's not the default,

1 In the form, select the component whose event handler you want to locate.
2 In the Object Inspector, click the Events tab.
3 Select the event whose handler you want to view and double-click in the Value column. The Code editor opens with the cursor inside the skeleton event handler.

TOPIC~>

Associating an event with an existing event handler

<~TOPIC<~HEAD
^# IDH_USEOP_eventHandlerAssociation
^$ Associating an event with an existing event handler
^A Associating an event with an existing event handler;eventhandlers
^X Working with events and event handlers;Using the Sender parameter;Displaying and coding shared events;Associating menu events with event handlers
^K events:handling;event handlers
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Associating an event with an existing event handler

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.

To associate an event with an existing event handler,

1 On the form, select the component whose event you want to handle.

2 On the Events page of the Object Inspector, select the event to which you want to attach a handler.

3 Click the down arrow in the Value column next to the event to open a list of previously written event handlers. (The list includes only event handlers written for events of the same name on the same form.) Select from the list by clicking an event-handler name.

The previous procedure is an easy way to reuse event handlers. <~JMP*Action lists*~!AL(UsingActionLists,1) JMP~> and <~JMP*action bands*~!AL(SettingUpActionBands,1) JMP~>, however, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications, whereas action bands cannot. For more information about action lists and action bands, see "Organizing actions for toolbars and menus" on page 9-21.

TOPIC~>

Using the Sender parameter

<~TOPIC<~HEAD
^# IDH_USEOP_usingTheSenderParameter
^$ Using the Sender parameter
^A Using the Sender parameter;
^X Working with events and event handlers;Associating an event with an existing event handler
^K events:handling;Sender parameter;handling events
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Using the Sender parameter
In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter.

D {bmc IC_D16.bmp} In Delphi, use the *Sender* parameter in an **if...then...else** statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
if Sender = Button1 then
  AboutBox.Caption := 'About ' + Application.Title
```

```
else
  AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

TOPIC~>

Displaying and coding shared events

<~TOPIC<~HEAD
^# IDH_USEOP_displayingAndCodingSharedEvents
^$ Displaying and coding shared events
^A Displaying and coding shared events;eventhandlers
^X Working with events and event handlers;Associating an event with an existing event handler;UsingActionLists
^K events:handling;event handlers;events:shared
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Displaying and coding shared events
When components share events, you can display their shared events in the Object Inspector. First, select the components by holding down the *Shift* key and clicking on them in the Form Designer; then choose the Events tab in the Object Inspector. From the Value column in the Object Inspector, you can now create a new event handler for, or assign an existing event handler to, any of the shared events.

TOPIC~>

Associating menu events with event handlers

<~TOPIC<~HEAD
^# IDH_USEOP_associatingMenuEventsWithCode
^$ Associating menu events with event handlers
^A Associating menu events with event handlers;eventhandlers
^X Working with events and event handlers;
^K events:menu;menu events;event handlers
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Associating menu events with event handlers
The Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This sectiontopic explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see "Creating and managing menus" on page 9-29<~JMP Creating and managing menus~!AL(CreatingAndManagingMenus,1) JMP~>.

To create an event handler for a menu item,

1 Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* component.
2 Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
3 From the Menu Designer, double-click the menu item. The Code editor opens with the cursor inside the skeleton event handler.
4 Type the code that you want to execute when the user selects the menu command.

To associate a menu item with an existing *OnClick* event handler,

1 Open the Menu Designer by double-clicking a *MainMenu* or *PopupMenu* component.
2 Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
3 On the Events page of the Object Inspector, click the down arrow in the Value column next to *OnClick* to open a list of previously written event handlers. (The list includes only event handlers written for *OnClick* events on this form.) Select from the list by clicking an event handler name.

TOPIC~>

Deleting event handlers

<~TOPIC<~HEAD
^# IDH_USEOP_deletingEventHandlers
^$ Deleting event handlers
^A Deleting event handlers;eventhandlers
^X Working with events and event handlers
^K event handlers, deleting
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Deleting event handlers
When you delete a component from a form using the Form Designer, the Code editor removes the component from the form's type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method's forward declaration (the unit's interface section in Delphi; the header file in C++) and its implementation (the unit's implementation section in Delphi; the cpp file in C++). Otherwise you'll get a compiler error when you build your project.

TOPIC~>

Cross-platform and non-cross-platform cComponents on the Component palette

<~TOPIC<~HEAD
^# IDH_USEOP_standardComponents
^$ Cross-platform and non-cross-platform cComponents on the Component palette
^A VCL standard components;VCLStandardComponents
^X UsingComponents
^K components:cross-platform;cross-platform components
^+ pcb:000
^T 2_DESIGN
^C 3
HEAD~>Cross-platform and non-cross-platform cComponents on the Component palette

The Component palette contains a selection of components that handle a wide variety of programming tasks. The components are arranged in pages according to their purpose and functionality. For example, commonly used components such as those to create menus, edit boxes, or buttons are located on the Standard page. Which pages appear in the default configuration depends on the edition of the product you are running.

The following tableTable 3.3 lists typical default pages and components available for creating applications, including those that are not cross-platform. You can use all CLX components, except WinCLX and VisualCLX, in both Windows and Linux applications. You can use WinCLX components in Windows-only applications and VisualCLX in cross-platform-only applications; however, the applications will not be cross-platform unless you isolate these portions of the code.

Table 6.1 Component palette pages

Page name	Description	Cross-platform?
Standard	Standard controls, menus.	Yes
Additional	Specialized controls.	Yes, except ApplicationEvents, ActionManager, ActionMainMenuBar, ActionToolBar, and CustomizeDlg. LCDNumber is in VisualCLX only.
Win32 (WinCLX)/ Common Controls (VisualCLX)	Common Windows controls used for developing a graphical user interface.	Many of the same components on the Win32 page are on the Common Controls page that appears when creating a cross-platform application. RichEdit, UpDown, HotKey, Animate, DataTimePicker, MonthCalendar, Coolbar, PageScroller, and ComboBoxEx are in WinCLX only. TextBrowser, TextViewer, IconViewer, and SpinEdit are in VisualCLX only.

Table 6.1 Component palette pages (continued)

Page name	Description	Cross-platform?
System	Components and controls for system-level access, including timers, multimedia, and DDE.	Yes, but uses different components such as DirectoryTreeView, FilesListView. WHAT ARE THESE FOR?No, except for Timer and PaintBox, which are on the Additional page when creating a CLX application.
Data Access	Components for working with database data that are not tied to any particular data access mechanism.	Yes, except for XMLTransform, XMLTransformProvider, and XMLTransformClient.
Data Controls	Visual, data-aware controls to design user interface.	Yes, except for DBRichEdit, DBCtrlGrid, and DBChart.
dbExpress	Database controls that use dbExpress, a cross-platform, database-independent layer that provides methods for dynamic SQL processing. It defines a common interface for accessing SQL servers.	Yes
DataSnap	Components that enable you to build multi-tier database applications.	No
BDE	Components that provide data access through the Borland Database Engine.	No
ADO	Components that provide data access through the ADO framework.	No
InterBase	Components that provide direct access to the InterBase database.	Yes
InterBaseAdmin	Components that access InterBase Services API calls.	No
InternetExpress	Components that are simultaneously a Web server application and the client of a multi-tiered database application.	No
Internet	Components for Internet communication protocols and Web applications.	Yes
WebSnap	Components for building Web server applications.	Yes
FastNet	NetMasters Internet controls.	No
QReport	QuickReport components for creating embedded reports.	No
Dialogs	Commonly used dialog boxes.	Yes, except for OpenPictureDialog, SavePictureDialog, PrintDialog, and PrinterSetupDialog.
Win 3.1	Old style Win 3.1 components.	No
Samples	Sample custom components.	No
ActiveX	Sample ActiveX controls; see Microsoft documentation (msdn.microsoft.com).	No
COM+	Component for handling COM+ events.	No
WebServices	Components for writing applications that implement or use SOAP-based Web Services.	Yes

Table 6.1 Component palette pages (continued)

Page name	Description	Cross-platform?
Servers	Components for embedding the Microsoft Internet Browser. NOT CORRECT. 2/22/02	No
Indy Clients	Cross-platform Internet components for the client (open source Winshoes Internet components).	Yes
Indy Servers	Cross-platform Internet components for the server (open source Winshoes Internet components).	Yes
Indy Misc	Additional cross-platform Internet components (open source Winshoes Internet components).	Yes
Office2K	COM Server examples for Microsoft Excel, Word, and so on (see Microsoft MSDN documentation).	No

You can add, remove, and rearrange components on the palette, and you can create component <~JMP*templates*~!Alink(CreatingComponentTemplates,1) JMP~> and <~JMP*frames*~!Alink(CreatingFrames,1) JMP~> that group several components. Some of the components on the ActiveX, Servers, and Samples pages, however, are provided as examples only and are not documented.

For more information about the components on the Component palette, see online Help. You can press F1 on the Component palette itself, on the component itself when it is selected, or after it has been dropped onto a form to display Help. If a tab of the Component palette is selected, the Help gives a general description for all of the components on that tab. You can also bring up Help on any object by placing the cursor anywhere on its name in the code in the editor and pressing F1.

For more information on the differences between cross-platform and Windows-only components, see Chapter 15, "Developing cross-platform applications."<~JMP Developing cross-platform applications.~!Alink(DevelopingCrossPlatformApplications,1) JMP~>

TOPIC~>

Adding custom components to the Component palette

<~TOPIC<~HEAD
^# IDH_USEOP_addingCustomComponents
^$ Adding custom components to the component palette
^A Adding custom components to the IDE;AddingCustomComponentsToTheIDE
^X installingcomponentpackages;OverviewOfComponentCreation
^K components:installing;custom components;components:custom
^+pcb:000
^T 2_DESIGN
^C 3
HEAD~>Adding custom components to the component palette

You can install custom components—written by yourself or third parties—on the Component palette and use them in your applications. To write a custom component, see Part V, "Creating custom components."Part IV, "Creating custom components."<~JMP Overview of component creation.~!Alink(OverviewOfComponentCreation,1) JMP~> To install an existing component, see "Installing component packages" on page 16-7<~JMP Installing component packages~!Alink(installingcomponentpackages,1) JMP~>.

TOPIC~>

Chapter

7

Working with controls

Controls are visual components that the user can interact with at runtime. This chapter describes a variety of features common to many controls.

Implementing drag and drop in controls

Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view— into another.

Starting a drag operation

Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate* and (optionally in Delphi) an integer parameter called *Threshold*. If you pass true for *Immediate*, dragging begins immediately. If you pass false, dragging does not begin until the user moves the mouse the number of pixels specified by *Threshold*. In C++, if *Threshold* is -1, a default value is used. Calling

```
BeginDrag (False, -1);
```

```
BeginDrag (false, -1);
```

allows the control to accept mouse clicks without beginning a drag operation.

You can place other conditions on whether to begin dragging, such as checking which mouse button the user pressed, by testing the parameters of the mouse-down event before calling *BeginDrag*. The following code, for example, handles a mouse-down event in a file list box by initiating a drag operation only if the left mouse button was pressed.

Delphi example

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then  { drag only if left button pressed }
    with Sender as TFileListBox do  { treat Sender as TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then  { is there an item here? }
          BeginDrag(False);  { if so, drag it }
    end;
end;
```

C++ example

```
void __fastcall TFMForm::FileListBox1MouseDown(TObject *Sender,
  TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if (Button == mbLeft)// drag only if left button pressed
  {
    TFileListBox *pLB = (TFileListBox *)Sender; // cast to TFileListBox
    if (pLB->ItemAtPos(Point(X,Y), true) >= 0) // is there an item here?
      pLB->BeginDrag(false, -1);                  // if so, drag it
  }
}
```

Accepting dragged items

When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event.

The drag-over event has a parameter called *Accept* that the event handler can set to true if it will accept the item. If Accept is true, the application sends a drag-and-drop event to the control.

The drag-over event has other parameters, including the source of the dragging and the current location of the mouse cursor, that the event handler can use to determine whether to accept the drop. In the following example, a directory tree view accepts dragged items only if they come from a file list box.

Delphi example

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True
  else
    Accept := False;
end;
```

C++ example

```
void __fastcall TForm1::TreeView1DragOver(TObject *Sender, TObject *Source,
  int X, int Y, TDragState State, bool &Accept)
{
  if (Source->InheritsFrom(__classid(TFileListBox)))
    Accept = true;
}
```

Dropping items

If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-and-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components as they are passed over.

In the following example, a directory tree view, accepting items dragged from a file list box, responds by moving files to the directory on which they are dropped.

Delphi example

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);
end;
```

C++ example

```
void __fastcall TForm1::TreeView1DragDrop(TObject *Sender, TObject *Source,
int X, int Y){
  if (Source->InheritsFrom(__classid(TFileListBox)))
  {
    TTreeNode *pNode = TreeView1->GetNodeAt(X,Y); // pNode is drop target
    AnsiString NewFile = pNode->Text + AnsiString("//") +
      ExtractFileName(FileListBox1->FileName); // build file name for drop target
    MoveFileEx(FileListBox1->FileName.c_str(), NewFile.c_str(),
      MOVEFILE_REPLACE_EXISTING | MOVEFILE_COPY_ALLOWED); // move the file
  }
}
```

Ending a drag operation

A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the drag was initiated. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event.

The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is nil (Delphi) or NULL (C++), it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates on the receiving control.

In this example, a file list box handles an end-drag event by refreshing its file list.

Delphi example

```
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileListBox1.Update;
end;
```

C++ example

```
void __fastcall TFMForm::FileListBox1EndDrag(TObject *Sender, TObject *Target,   int X, int
Y)
  if (Target)
    FileListBox1->Update();
};
```

Customizing drag and drop with a drag object

You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and drag-and-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-and-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the source needs to support each kind of control. When you use a descendant of *TDragObject*, however, the source is the drag object itself; if each control creates the same kind of drag object in its *OnStartDrag* event, the target needs to handle only one kind of object. The drag-over and drag-and-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

Drag objects let you drag items between a form implemented in the application's main executable file and a form implemented using a shared object or between forms that are implemented using different shared objects.

Working with text in controls

The following sections explain how to use various features of edit and memo controls. Some of these features work with edit controls as well.

- Setting text alignment
- Adding scroll bars at runtime
- Adding the clipboard object
- Selecting text
- Selecting all text
- Cutting, copying, and pasting text
- Deleting selected text
- Disabling menu items
- Providing a pop-up menu
- Handling the OnPopup event

Setting text alignment

In a rich edit or memo component, text can be left- or right-aligned or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is true; if word wrapping is turned off, there is no margin to align to. *WordWrap* turns wordwrapping on and off. When on, the *WrapMode*, *WrapBreak*, and *WrapAtValue* properties allow fine-grain control on how the wrapping is done.

You can also use the *HMargin* property to adjust the left and right margins in a memo control.

D

Delphi example

For example, in Delphi, the following code attaches an *OnClick* event handler to a Character|Left menu item, then attaches the same event handler to both a Character|Right and Character|Center menu items.

```
procedure TForm1.AlignClick(Sender: TObject);
begin
  Left1.Checked := False;  { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True;  { check the item clicked }
  with Editor do  { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
```

```
      Alignment := taCenter;
end;
```

C++ example

In C++, the following code sets the alignment depending on which toolbar button (left aligned, right aligned, or center aligned) is chosen:

```
switch((int)Memo1->Paragraph->Alignment)
{
  case 0: LeftAlign->Down   = true; break;
  case 1: RightAlign->Down  = true; break;
  case 2: CenterAlign->Down = true; break;
}
```

Adding scroll bars at runtime

Rich edit and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.

To add scroll bars at runtime:

1 Determine whether the text might exceed the right margin. In most cases, this means checking whether word wrapping is enabled. You might also check whether any text lines actually exceed the width of the control.

2 Set the rich edit or memo component's *ScrollBars* property to include or exclude scroll bars.

The following example attaches an *OnClick* event handler to a Character | WordWrap menu item.

Delphi example

```
procedure TForm1.WordWrap1Click(Sender: TObject);
begin
  with Editor do
  begin
    WordWrap := not WordWrap;  { toggle word wrapping }
    if WordWrap then
      ScrollBars := ssVertical  { wrapped requires only vertical }
    else
      ScrollBars := ssBoth;  { unwrapped might need both }
      WordWrap1.Checked := WordWrap;  { check menu item to match property }
  end;
end;
```

C++ example

```
void __fastcall TForm1::WordWrap1Click(TObject *Sender)
{
  Editor->WordWrap = !(Editor->WordWrap);   // toggle word wrapping
  if (Editor->WordWrap)
    Editor->ScrollBars = ssVertical;        // wrapped requires only vertical
  else
    Editor->ScrollBars = ssBoth;            // unwrapped can need both
  WordWrap1->Checked = Editor->WordWrap;    // check menu item to match property
}
```

The rich edit and memo components handle their scroll bars in a slightly different way. The rich edit component can hide its scroll bars if the text fits inside the bounds of the component. The memo always shows scroll bars if they are enabled.

Adding the clipboard object

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. *TClipboard* in encapsulates a clipboard and includes methods for cutting, copying, and pasting text (and other formats, including graphics). *TClipboard* is declared in the QClipbrd unit.

To add the *Clipboard* object to an application in Delphi:

1 Select the unit that will use the clipboard.

2 Search for the **implementation** reserved word.

3 Add Q*Clipbrd* to the **uses** clause below **implementation**.

- If there is already a **uses** clause in the **implementation** part, add Q*Clipbrd* to the end of it.
- If there is not already a **uses** clause, add one that says

```
uses QClipbrd;
```

For example, in an application with a child window, the uses clause in the unit's implementation part might look like this:

```
uses
  MDIFrame, QClipbrd;
```

To add the *Clipboard* object to an application in C++:

1 Select the unit that will use the clipboard.

2 In the form's .h file, add:

```
#include <clx\QClipbrd.hpp>
```

Selecting text

For text in an edit control, before you can send any text to the clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted.

Table 7.1 lists properties commonly used to handle selected text.

Table 7.1 Properties of selected text

Property	Description
SelText	Contains a string representing the selected text in the component.
SelLength	Contains the length of a selected string.
SelStart	Contains the starting position of a string relative to the beginning of an edit control's text buffer.

For example, the following *OnFind* event handler searches a Memo component for the text specified in the *FindText* property of a find dialog component. If found, the first occurrence of the text in Memo1 is selected.

D **Delphi example**

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  I, J, PosReturn, SkipChars: Integer;
begin
  for I := 0 to Memo1.Lines.Count do
  begin
    PosReturn := Pos(FindDialog1.FindText,Memo1.Lines[I]);
    if PosReturn <> 0 then {found!}
    begin
      Skipchars := 0;
      for J := 0 to I - 1 do
        Skipchars := Skipchars + Length(Memo1.Lines[J]);
      SkipChars := SkipChars + (I*2);
      SkipChars := SkipChars + PosReturn - 1;
      Memo1.SetFocus;
      Memo1.SelStart := SkipChars;
      Memo1.SelLength := Length(FindDialog1.FindText);
      Break;
    end;
  end;
end;
```

Selecting all text

The *SelectAll* method selects the entire contents of an edit control, such as a rich edit or memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging.

To select the entire contents of a rich edit or memo control, call the *Memo1* control's *SelectAll* method.

D **Delphi example**

```
procedure TMainForm.SelectAll(Sender: TObject);
```

```
begin
  Memo1.SelectAll;  { select all text in Memo }
end;
```

C++ example

```
void __fastcall TMainForm::SelectAll(TObject *Sender)
{
  Memo1->SelectAll();     // select all text in Memo
}
```

Cutting, copying, and pasting text

Applications that use the Q*Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the clipboard. The edit components that encapsulate the standard text-handling controls all have methods built into them for interacting with the clipboard. (See “Using the clipboard with graphics” on page 11-28 for information on using the clipboard with graphics.)

To cut, copy, or paste text with the clipboard, call the edit component’s *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively.

For example, the following code attaches event handlers to the *OnClick* events of the Edit|Cut, Edit|Copy, and Edit|Paste commands, respectively:

Delphi example

```
procedure TForm.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TForm.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TForm.PasteFromClipboard(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;
```

C++ example

```
void __fastcall TMainForm::EditCutClick(TObject* Sender)
{ Memo1->CutToClipboard();
}
void __fastcall TMainForm::EditCopyClick(TObject* Sender)
{ Memo1->CopyToClipboard();
}
void __fastcall TMainForm::EditPasteClick(TObject* Sender)
{ Memo1->PasteFromClipboard();
}
```

Deleting selected text

You can delete the selected text in an edit component without cutting it to the clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

Delphi example

```
procedure TForm1.Delete(Sender: TObject);
begin
  Memo1.ClearSelection;
end;
```

C++ example

```
void __fastcall TMainForm::EditDeleteClick(TObject *Sender)
{
  Memo1->ClearSelection();
}
```

Disabling menu items

It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its *Enabled* property to false.

In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit menu based on whether *Memo1* has selected text. The Paste command is enabled or disabled based on whether any text exists on the clipboard.

Delphi example

```
procedure TForm1.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;  { declare a temporary variable }
begin
  Paste1.Enabled := Clipboard.Provides('text'); {Enable/disable paste menu item}
  ...
end;
```

C++ example

```
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->Provides("text");
  bool HasSelection = (Memo1->SelLength > 0);  // true if text is selected
  Cut1->Enabled = HasSelection;  // enable menu items if HasSelection is true
```

```
    Copy1->Enabled = HasSelection;
    Delete1->Enabled = HasSelection;
  }
```

The *Provides* method of the clipboard returns a Boolean value based on whether the clipboard contains objects, text, or images of a particular format. (In this case, the text is generic. You can specify the type of text using a subtype such as text/plain for plain text or text/html for html.) By calling *Provides* with the parameter text, you can determine whether the clipboard contains any text, and you can enable or disable the Paste item as appropriate.

Chapter 11, "Working with graphics" provides more information about using the clipboard with graphics.

Providing a pop-up menu

Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands.

In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts.

A form's *PopupMenu* property specifies what pop-up menu to display when a user right-clicks any item on the form. Individual controls also have *PopupMenu* properties that can override the form's property, allowing customized menus for particular controls.

To add a pop-up menu to a form:

1. Place a pop-up menu component on the form.
2. Use the Menu Designer to define the items for the pop-up menu.
3. Set the *PopupMenu* property of the form or control that displays the menu to the name of the pop-up menu component.
4. Attach handlers to the *OnClick* events of the pop-up menu items.

Handling the OnPopup event

You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu, as described in "Disabling menu items" on page 7-10.

With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called *OnPopup*.

To adjust menu items on a pop-up menu before displaying them:

1 Select the pop-up menu component.
2 Attach an event handler to its *OnPopup* event.
3 Write code in the event handler to enable, disable, hide, or show menu items.

In the following code, the *Edit1Click* event handler described previously in "Disabling menu items" on page 7-10 is attached to the pop-up menu component's *OnPopup* event. A line of code is added to *Edit1Click* for each item in the pop-up menu.

Delphi example

```
procedure TForm1.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.Provides('text');
  Paste2.Enabled := Paste1.Enabled;{Add this line}
  HasSelection := Editor.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Cut2.Enabled := HasSelection;{Add this line}
  Copy1.Enabled := HasSelection;
  Copy2.Enabled := HasSelection;{Add this line}
  Delete1.Enabled := HasSelection;
end;
```

C++ example

```
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->Provides('text');
  Paste2->Enabled = Paste1->Enabled;           // add this line
  bool HasSelection = (Memo1->SelLength > 0);  // true if text is selected
  Cut1->Enabled = HasSelection;    // enable menu items if HasSelection is true
  Cut2->Enabled = HasSelection;    // add this line
  Copy1->Enabled = HasSelection;
  Copy2->Enabled = HasSelection;   // add this line
  Delete1->Enabled = HasSelection;
}
```

Adding graphics to controls

Several controls let you customize the way the control is rendered. These include list boxes, combo boxes, menus, headers, tab controls, list views, status bars, tree views, and toolbars. Instead of using the standard method of drawing a control or its items, the control's owner (generally, the form) draws them at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see "Adding images to menu items" on page 9-35.

All owner-draw controls contain lists of items. Usually, those lists are lists of strings that are displayed as text, or lists of objects that contain strings that are displayed as text. You can associate an object with each item in the list to make it easy to use that object when drawing items.

In general, creating an owner-draw control in the IDE involves these steps:

1. Indicating that a control is owner-drawn.
2. Adding graphical objects to a string list.
3. Drawing owner-draw items

Indicating that a control is owner-drawn

To customize the drawing of a control, you must supply event handlers that render the control's image when it needs to be painted. Some controls receive these events automatically. For example, list views, tree views, and toolbars all receive events at various stages in the drawing process without your having to set any properties. These events have names such as OnCustomDraw or OnAdvancedCustomDraw.

Other controls, however, require you to set a property before they receive owner-draw events. List boxes, combo boxes, header controls, and status bars have a property called *Style*. *Style* determines whether the control uses the default drawing (called the "standard" style) or owner drawing. Grids use a property called *DefaultDrawing* to enable or disable the default drawing. List views and tab controls have a property called *OwnerDraw* that enables or disabled the default drawing.

List boxes and combo boxes have additional owner-draw styles, called *fixed* and *variable*, as Table 7.2 describes. Other controls are always fixed, although the size of the item that contains the text may vary, the size of each item is determined before drawing the control.

Table 7.2 Fixed vs. variable owner-draw styles

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the *ItemHeight* property.	*lbOwnerDrawFixed*, *csOwnerDrawFixed*
Variable	Each item might have a different height, determined by the data at runtime.	*lbOwnerDrawVariable*, *csOwnerDrawVariable*

Adding graphical objects to a string list

Every string list has the ability to hold a list of objects in addition to its list of strings. You can also add graphical objects of varying sizes to a string list.

For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections.

Note that you can also organize graphical objects using an image list by creating a *TImageList*. However, these images must all be the same size. See "Adding images to menu items" on page 6-27 for an example of setting up an image list.

Adding images to an application

An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example, you can store bitmaps for owner-draw controls in hidden image controls, like this:

1. Add image controls to the main form.
2. Set their *Name* properties.
3. Set the *Visible* property for each image control to false.
4. Set the *Picture* property of each image to the desired bitmap using the Picture editor from the Object Inspector.

The image controls are invisible when you run the application. The image is stored with the form so it doesn't have to be loaded from a file at runtime.

Adding images to a string list

Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available.

Sizing owner-draw items

Before giving your application the chance to draw each item in a variable owner-draw control, a measure-item event, *TMeasureItemEvent*. The measure-item event tells the application where the item appears on the control.

Kylix determines the size of the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle chosen. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be large enough for both.

To change the size of an owner-draw item, attach an event handler to the measure-item event in the owner-draw control. Depending on the control, the name of the event can vary.List boxes and combo boxes use *OnMeasureItem*. Grids have no measure-item event.

The sizing event has two important parameters: the index number of the item and the height of that item. The height is variable: the application can make it either smaller or larger. The positions of subsequent items depend on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is the height of the item. The width of the item is always the width of the control.

Owner-draw grids cannot change the sizes of their cells as they draw. The size of each row and column is set before drawing by the *ColWidths* and *RowHeights* properties.

The following code, attached to the *OnMeasureItem* event of an owner-draw list box, increases the height of each list item to accommodate its associated bitmap.

Delphi example

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer);  { note that TabWidth is a var parameter}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

C++ example

```
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control, int Index,
  int &Height)      // note that Height is passed by reference
{
  int BitmapHeight = ((TBitmap *)ListBox1->Items->Objects[Index])->Height + 2;
  // make sure list item has enough room for bitmap (plus 2)
  if (BitmapHeight > Height)
    Height = BitmapHeight;
}
```

Note You must typecast the items from the *Objects* property in the string list. *Objects* is a property of type *TObject* so that it can hold any kind of object. When you retrieve objects from the array, you need to typecast them back to the actual type of the items.

Drawing owner-draw items

When an application needs to draw or redraw an owner-draw control, the operating system generates draw-item events for each visible item in the control. Depending on the control, the item may also receive draw events for the item as a part of the item.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control.

The names of events for owner drawing typically start with one of the following:

- *OnDraw*, such as *OnDrawItem* or *OnDrawCell*
- *OnCustomDraw*, such as *OnCustomDrawItem*

The draw-item event contains parameters identifying the item to draw, the rectangle in which to draw, and usually some information about the state of the item (such as whether the item has focus). The application handles each event by rendering the appropriate item in the given rectangle.

For example, the following code shows how to draw items in a list box that has bitmaps associated with each string. It attaches this handler to the *OnDrawItem* event for the list box:

Delphi example

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
  Draw(R.Left, R.Top + 4, Bitmap);  { draw bitmap }
  TextOut(R.Left + 2 + Bitmap.Width,  { position text }
    R.Top + 2, DriveTabSet.Tabs[Index]);  { and draw it to the right of the
                                          bitmap }
  end;
end;
```

C++ example

```
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control, int Index,
  TRect &Rect, TOwnerDrawState State)

  TBitmap *Bitmap = (TBitmap *)ListBox1->Items->Objects[Index];
  ListBox1->Canvas->Draw(R.Left, R.Top + 2, Bitmap); // draw the bitmap
  ListBox1->Canvas->TextOut(R.Left + Bitmap->Width + 2, R.Top + 2,
    ListBox1->Items->Strings[Index]);        // and write the text to its right
}
```

Chapter

8

Building applications and shared objects

This chapter provides an overview of how to create applications and shared objects.

Creating applications

The main use of Kylix is designing and building the following types of applications:

- GUI applications
- Console applications

PackagesWorking with shared object librariesGUI applications generally have an easy-to-use interface. Console applications run from a console window.

.GUI applications

A graphical user interface (GUI) application is one that is designed using graphical features such as windows, menus, dialog boxes, and features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an executable file. You can extend the application by calling shared objects, packages, and other support files from the executable.

The IDE offers two application UI models:

- Single document interface (SDI)
- Multiple document interface (MDI)

Any form can be implemented as a single document interface (SDI) or multiple document interface (MDI) form. An SDI application normally contains a single document view. In an MDI application, in contrast, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors.

For more information on developing the UI for an application, see Chapter 9, "Developing the application user interface."

SDI applications

To create a new SDI application, choose File | New | Application.

By default, the *FormStyle* property of your *Form* object is set to *fsNormal*, so the IDE assumes that all new applications are SDI applications.

MDI applications

To create a new MDI application using a wizard:

1 Choose File | New | Other to bring up the New Items dialog.

2 Click on the Projects page and double-click MDI Application.

3 Click OK.

MDI applications require more planning and are somewhat more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. Set the *FormStyle* property of the *TForm* object to specify whether a form is a child (*fsMDIChild*) or main form (*fsMDIForm*). It is a good idea to define a base class for your child forms and derive each child form from this class, to avoid having to reset the child form's properties.

MDI applications often include a main menu that has items such as Cascade and Tile for viewing multiple windows in various styles. When a child window is minimized, its icon is located in the MDI parent form.

To create a new MDI application without using a wizard:

1 Create the main window form or MDI parent window. Set its *FormStyle* property to *fsMDIForm*.

2 Create a menu for the main window that includes File | Open, File | Save, and Window which has Cascade, Tile, and Arrange All items.

3 Create the MDI child forms and set their *FormStyle* properties to *fsMDIChild*.

Setting IDE, project, and compiler options

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE. To specify various options for your project, choose Project | Options

Setting default project options

To change the default options that apply to all future projects, set the options in the Project Options dialog box and check the Default box at the bottom right of the window. All new projects will use the current options selected by default.

For more information, see the online Help.

Programming templates

Programming templates are commonly used *skeleton* structures that you can add to your source code and then fill in. Some standard code templates such as those for array, class, and function declarations, and many statements, are included with Kylix.

You can also write your own templates for coding structures that you often use. For example, if you want to use a **for** loop in your code, you could insert the following template:

Delphi example

```
for :=  to  do
begin

end;
```

C++ example

```
for (; ;)
{

}
```

To insert a code template in the Code editor, press *Ctrl-j* and select the template you want to use. You can also add your own templates to this collection. To add a template:

1. Choose Tools | Editor Options.
2. Click the Code Insight tab.
3. In the Templates section, click Add.
4. Type a name for the template after Shortcut name, enter a brief description of the new template, and click OK.
5. Add the template code to the Code text box.
6. Click OK.

Console applications

Console applications are 32-bit programs that run without a graphical interface, usually in a console window. These applications typically don’t require much user input and perform a limited set of functions. When you create a new console application, the IDE does not create a new form. Only the Code editor appears.

D To create a new console application in Delphi:

1 Choose File | New | Other and double-click Console Application from the New Items dialog box.

C++ To create a new console application in C++:

1 Choose File | New | Other and double-click Console Wizard from the New Items dialog box.
2 In the Console Wizard dialog box, choose the source type (C or C++) for the main module of the project, or check Specify project source and choose a pre-existing file that contains a main function. Check the Console Application option to create a console window; uncheck this option if you want to create a GUI application.
3 To add objects in C++, indicate that you will be using CLX by checking the Use CLX option. If you do not indicate in the wizard that you want to use CLX, you will not be able use any of the CLX classes in this application later. Trying to do so causes linker errors. , and click the OK button.

In both Delphi and C++, the IDE creates a project file for this type of source file and displays the Code editor. You can then add CLX objects to your console application.

Note When you create a new console application, the IDE does not create a new form. Only the Code editor appears.

Console applications should handle all exceptions to prevent windows from displaying a dialog during its execution.

D Delphi example

For example, in Delphi, your application should include exception handling such as shown in the following code:

```
program Project1;
{$APPTYPE CONSOLE}
begin
try
  raise exception.create('hi');
  except
    WriteLn('exception occurred');
  end;
end.
```

Creating packages

Packages are special DLLs used by Galileo applications, the IDE, or both. There are two kinds of packages: runtime packages and design-time packages. Runtime packages provide functionality to a program while that program is running. Design-time packages extend the functionality of the IDE.

For more information on packages, see Chapter 16, “Working with packages and components.”

Working with shared object libraries

Shared object libraries on Linux are similar to Windows DLLs. You can link with third-party shared objects using external function declarations, just as you would with DLL functions under Windows.

The Linux program loader ignores module name bindings when resolving external function references. If an application uses two .so libraries that both export a function named MyLibrary, the loader binds all MyLibrary references to the first MyLibrary function it finds. (The search order is described in the section on “Shared Object Dependencies” in the ELF standard.) When naming conflicts are unavoidable, you can prevent unintended behavior by loading objects dynamically with dlopen().

When you build a library project, the compiler generates a shared object (.so file) instead of a regular executable. By default, the name of the generated file starts with “lib” or “bpl” (for a package). For example, if your project file is called something, the compiler generates a shared object called libsomething.so or bplsomething.so.

D The following Delphi compiler directives can be placed in library project files:

Table 8.1 Delphi compiler directives for libraries

Compiler Directive	Description
{$SONAME 'string'}	Provides a symbolic link to the .so file.
{$SOPREFIX 'string'}	Overrides the default ‘lib’ or ‘bpl’ prefix in the output file name. For example, you could specify {$SOPREFIX 'dcl'} for a design-time package, or use {$SOPREFIX ' '} to eliminate the prefix entirely.
{$SOSUFFIX 'string'}	Adds a specified suffix to the output file name before the .so extension. For example, use {$SOSUFFIX '-2.1.3'} in something.pas to generate libsomething-2.1.3.so.
{$SOVERSION 'string'}	Adds a second extension to the output file name after the .so extension. For example, use {$SOVERSION '2.1.3'} in something.pas to generate libsomething.so.2.1.3.

When to use shared objects and packages

For most applications, packages provide greater flexibility and are easier to create than shared objects. However, there are several situations where shared objects would be better suited to your projects than packages:

- Your code module will be called from non-Kylix applications.
- You are extending the functionality of a Web server.
- You are creating a code module for third-party developers.

However, if your application includes VisualCLX, you must use packages instead of shared objects. Only packages can manage the startup and shut down of the Qt shared libraries.

You cannot pass runtime type information (RTTI) across shared objects or from a shared object to an executable. That's because shared objects all maintain their own symbol information. Packages share symbol information.

D In Delphi, if you need to pass a *TStrings* object from a shared object using an **is** or **as** operator, you need to create a package rather than a shared object.

Using shared objects in C++

A shared object can be used in a Kylix for C++ application just as it would be in any C++ application.

To statically load a shared object when your C++ application is loaded, link the import library file for that shared object into your C++ application at link time. To add an import library to a Kylix for C++ application, choose Project | Add to Project and select the .a file you want to add.

The exported functions of that shared object then become available for use by your application. Prototype the shared object functions your application uses with the **__declspec** `(dllimport)` modifier:

```
__declspec(dllimport) return_type imported_function_name(parameters);
```

To dynamically load a shared object during the run of a Kylix application, include the import library, just as you would for static loading, and set the delay load linker option on the Project | Options | Advanced Linker tab.

Creating C++ shared objects

To create shared objects in C++:

1 Choose File | New | Other to display the New Items dialog box.
2 Double-click the Shared Object Wizard icon.
3 Choose the Source type (C or C++) for the main module.
4 Click Use CLX to create a shared object containing CLX components. This option is only available for C++ source modules.
See "Creating shared objects containing CLX components" on page 8-7.
5 If you want the shared object to be multi-threaded, check the Multi-threaded option.
6 Click OK.

Exported functions in the code should be identified with the __declspec `(dllexport)` modifier. For example, the following code is legal in the C++ compiler:

```
// myso.cpp
double dblValue(double);
double halfValue(double);
extern "C" __declspec(dllexport) double changeValue(double, bool);
```

```
double dblValue(double value)
{
   return value * value;
};

double halfValue(double value)
{
   return value / 2.0;
}

double changeValue(double value, bool whichOp)
{
   return whichOp ? dblValue(value) : halfValue(value);
}
```

In the code above, the function *changeValue* is exported, and therefore made available to calling applications. The functions *dblValue* and *halfValue* are internal, and cannot be called from outside of the shared object.

Creating shared objects containing CLX components

One of the strengths of shared objects is that a shared object created with one development tool can often be used by application written using a different development tool. When your shared object contains CLX components (such as forms) that are to be used by the calling application, you need to provide exported interface routines that use standard calling conventions, avoid C++ name mangling, and do not require the calling application to support the CLX libraries in order to work. To create CLX components that can be exported, use runtime packages. For more information, see Chapter 16, “Working with packages and components.”

Linking shared objects in C++

You can set the linker options for your shared object on the Linker page of the Project Options dialog. The default check box on this page also creates an import library for your shared object. If compiling from the command line, invoke the linker executable, ilinkwith the `-Tpd` switch. For example:

```
ilink /c /aa /Tpd borintso.o.o myso.oo, myso.so, myso.map, libborcrtl.so libborstl.so
libborunwind.so libc.so libm.so
```

For more information about the different options for linking shared objects and using them with other modules that are statically or dynamically linked to the runtime library, see the online Help.

Writing database applications

One of Kylix's strengths is its support for creating advanced database applications. Kylix includes built-in tools that allow you to connect to InterBase, MySQL, or other servers while providing transparent data sharing between applications.

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect to the database information itself. Kylix supports two kinds of datasets:

- dbExpress
- Client

Different kinds of datasets connect to the underlying database information in different ways. *dbExpress* provides fast access to database information, supports cross-platform development, but does not include many data manipulation functions. Client datasets can buffer data in memory but you can't connect a client dataset directly to a database server, because client datasets do not include any built-in database access mechanism. Instead, you need to connect the client dataset to another dataset that can handle data access.

See Part II, "Developing database applications" in this manual for details on design database applications. For an overview of the architecture, see "Database architecture" on page 14-4. See "Deploying database applications" on page 18-4 for deployment information.

Note Not all editions of Kylix include database support.

Distributing database applications

Kylix provides support for creating distributed database applications using a coordinated set of components. Distributed database applications can be built on a variety of communications protocols, including TCP/IP and SOAP.

For more information about building distributed database applications, see Chapter 27, "Using Web Services to create multi-tiered database applications."

Developing applications for the Internet

You can use Kylix to develop different types of applications for use on the Internet. Most developers will want to use technologies such as Web Broker and WebSnap to develop Web server applications that deliver Web content. Another Internet technology called Web Services are applications that are callable by other programs across the Internet in a language-neutral fashion. If you're developing APIs for communicating over the Internet, you would use Web Services.

Web Broker provides the underlying architecture for both Web Snap and Web Services. Web Broker's framework is the basis for responding to HTTP messages.

The next section provides an overview of the latest Web server technologies.

Creating Web server applications

Web server applications are applications that run on servers that deliver Web content such as HTML Web pages or XML documents over the Internet. Examples of Web server applications include those which control access to a Web site, generate purchase orders, or respond to information requests.

You can create several different types of Web server applications using the following Kylix technologies:

- Web Broker
- WebSnap
- Web Services

Creating Web Broker applications

You can use Web Broker to create Web server applications such as CGI applications or dynamic-link libraries (shared objects). These Web server applications can contain any nonvisual component. Components on the Internet page of the Component palette enable you to create event handlers, programmatically construct HTML or XML documents, and transfer them to the client.

To create a new Web server application using the Web Broker architecture, choose File | New | Other and double-click the Web Server Application in the New Items dialog box. Then select the Web server application type:

Table 8.2 Web server applications

Web server application type	Description
CGI Stand-alone executable	CGI Web server applications are console applications that receive requests from clients on standard input, process those requests, and sends back the results to the server on standard output to be sent to the client. In Delphi, selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate $APPTYPE directive to the source.
Apache Shared Module (SO)	Selecting this type of application sets up your project as a shared object. Apache Web server applications are shared objects loaded by the Web server. Information is passed to the shared object, processed, and returned to the client by the Web server.
Web App Debugger Stand-alone executable	Selecting this type of application sets up an environment for developing and testing Web server applications. Web App Debugger applications are executable files loaded by the Web server. This type of application is not intended for deployment.

When writing cross-platform applications, you should select CGI stand-alone or Apache Shared Module (SO) for Web server development. These are also the same options you see when creating WebSnap and Web Service applications.

For more information on building Web server applications, see Chapter 29, "Creating Internet server applications."

Creating WebSnap applications

WebSnap provides a set of components and wizards for building advanced Web servers that interact with Web browsers. WebSnap components generate HTML or other MIME content for Web pages. WebSnap is for server-side development. WebSnap cannot be used in cross-platform applications at this time.

To create a new WebSnap application, select File | New | Other and select the WebSnap tab in the New Items dialog box. Choose WebSnap Application. Then select the Web server application type (CGI, Win-CGI, Apache). See Table 8.2, "Web server applications" for details.

For more information on WebSnap, see Chapter 31, "Creating Web server applications using WebSnap."

Creating Web Services applications

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided. You use Web Services to produce or consume programmable services over the Internet using emerging standards such as XML, XML Schema, SOAP (Simple Object Access Protocol), and WSDL (Web Service Definition Language).

Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

You can use Kylix to build servers to implement Web Services and clients that call on those services. You can write clients for arbitrary servers to implement Web Services that respond to SOAP messages, and Kylix servers to publish Web Services for use by arbitrary clients.

Refer to Chapter 33, "Using Web Services" for more information on Web Services.

Using data modules

A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are several types of data modules, including standard, remote, and Web modules. Each type of data module serves a special purpose.

- Standard data modules are particularly useful for single- and two-tiered database applications, but can be used to organize the nonvisual components in any application. For more information, see "Creating and editing standard data modules" on page 8-11.
- Remote data modules form the basis of an application server in a multi-tiered database application. They are not available in all editions. In addition to holding the nonvisual components in the application server, remote data modules expose the interface that clients use to communicate with the application server. For more information about using them, see "Adding a remote data module to an application server project" on page 8-14.
- Web modules form the basis of Web server applications. In addition to holding the components that create the content of HTTP response messages, they handle the dispatching of HTTP messages from client applications. See Chapter 29, "Creating Internet server applications" for more information about using Web modules.

Creating and editing standard data modules

To create a standard data module for a project, choose File | New | Data Module. The IDE opens a data module container on the desktop, displays the unit file for the new module in the Code editor, and adds the module to the current project.

At design time, a data module looks like a standard form with a white background and no alignment grid. As with forms, you can place nonvisual components from the Component palette onto a module, and edit their properties in the Object Inspector. You can resize a data module to accommodate the components you add to it.

You can also right-click a module to display a context menu for it. The following table summarizes the context menu options for a data module.

Table 8.3 Context menu options for data modules

Menu item	Purpose
Edit	Displays a context menu with which you can cut, copy, paste, delete, and select the components in the data module.
Position	Aligns nonvisual components to the module's invisible grid (*Align To Grid) or* according to criteria you supply in the Alignment dialog box (*Align*).
Tab Order	Enables you to change the order that the focus jumps from component to component when you press the tab key.
Creation Order	Enables you to change the order that data access components are created at start-up.
Revert to Inherited	Discards changes made to a module inherited from another module in the Object Repository, and reverts to the originally inherited module.
Add to Repository	Stores a link to the data module in the Object Repository.
View as Text	Displays the text representation of the data module's properties.
Text DFM	Toggles between the formats (binary or text) in which this particular form file is saved.

For more information about data modules, see the online Help.

Naming a data module and its unit file

The title bar of a data module displays the module's name. The default name for a data module is "DataModule*N*" where *N* is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to "DataModule2." The corresponding unit file for *DataModule2* defaults to "Unit2."

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your modules.

To rename a data module:

1 Select the module.

2 Edit the *Name* property for the module in the Object Inspector.

The new name for the module appears in the title bar when the *Name* property in the Object Inspector no longer has focus.

Changing the name of a data module at design time changes its variable name in the interface section of code. It also changes any use of the type name in procedure declarations. You must manually change any references to the data module in code you write.

To rename a unit file for a data module:

1 Select the unit file.

Placing and naming components

You place nonvisual components in a data module just as you place visual components on a form. Click the desired component on the appropriate page of the Component palette, then click in the data module to place the component. You cannot place visual controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, Kylix assigns it a generic name that identifies what kind of component it is, followed by a *1*. For example, the *TDataSource* component adopts the name *DataSource1*. This makes it easy to select specific components whose properties and methods you want to work with.

You may still want to name a component a different name that reflects the type of component and what it is used for.

To change the name of a component in a data module:

1 Select the component.
2 Edit the component's *Name* property in the Object Inspector.

The new name for the component appears under its icon in the data module as soon as the *Name* property in the Object Inspector no longer has focus.

For example, suppose your database application uses the CUSTOMER table. To access the table, you need a minimum of two data access components: a data source component (*TDataSource)* and a table component (*TClientDataSet)*. When you place these components in your data module, Kylix assigns them the names *DataSource1* and *ClientDataSet1*. To reflect the type of component and the database they access, CUSTOMER, you could change these names to *CustomerSource* and *CustomerTabl*e.

Using component properties and events in a data module

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TClientDataSet*, to control the data available to the data source components that use those datasets. Setting the *ReadOnly* property to true for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset, by double-clicking on *ClientDataSet1,* to restrict the fields within a table or query that are available to a data source and therefore to the data-aware controls on forms. The properties you set for components in a data module apply consistently to all forms in your application that use the module.

In addition to properties, you can write event handlers for components. For example, a *TDataSource* component has three possible events: *OnDataChang*e, *OnStateChang*e, and *OnUpdateDat*a. A *TClientDataSet* component has over 20 potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your application.

Creating business rules in a data module

Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module.

D Delphi example

In Delphi, the prototypes for the procedures and functions you write for a data module should appear in the module's **type** declaration:

```
type
  TCustomerData = class(TDataModule)
    Customers: TClientDataSet;
    Orders: TClientDataSet;
    ⋮
  private
    { Private declarations }
  public
    { Public declarations }
    procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
  end;
```

```
var
  CustomerData: TCustomerData;
```

The procedures and functions you write in Delphi should follow in the implementation section of the code for the module.

Accessing a data module from a form

To associate visual controls on a form with a data module, you must first add the data module to the form's uses clause (Delphi) or the data module's header file to the form's cpp file (C++). You can do this in several ways:

- In the Code editor, open the form's unit file and add the name of the data module to the uses clause in the interface section (Delphi) or include the data module's header file using the *#include* directive (C++).
- Click the form's unit file, choose File | Use Unit (in the Delphi IDE) or File | Include Unit Hdr (in the C++ IDE), and enter the name of the module or pick it from the list box in the Use Unit dialog.
- For database components, in the data module click a dataset or query component to open the Fields editor and drag any existing fields from the editor onto the form. Kylix prompts you to confirm that you want to add the module to the form (in the uses clause in Delphi), then creates controls (such as edit boxes) for the fields.

For example, if you've added the *TClientDataSet* component to your data module, double-click it to open the Fields editor. Select a field and drag it to the form. An edit box component appears.

Because the data source is not yet defined, the IDE adds a new data source component, *DataSource1,* to the form and sets the edit box's *DataSource* property to *DataSource1*. The data source automatically sets its *DataSet* property to the dataset component, *ClientDataSet1,* in the data module.

You can define the data source *before* you drag a field to the form by adding a *TDataSource* component to the data module. Set the data source's *DataSet* property to *ClientDataSet1.* After you drag a field to the form, the edit box appears with its *TDataSource* property already set to *DataSource1*. This method keeps your data access model cleaner.

Adding a remote data module to an application server project

Some editions of Kylix allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks.

To add a remote data module to a project:

1. Choose File | New | Other.
2. Select the Multitier page in the New Items dialog box.

3 Double-click the Remote Data Module icon to open the Remote Data Module wizard.

Once you add a remote data module to a project, use it just like a standard data module.

For more information about multi-tiered database applications, see Chapter 27, "Using Web Services to create multi-tiered database applications."

Using the Object Repository

The Object Repository (Tools | Repository) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The Repository is maintained in the delphi69dro (the Delphi IDE) or bcb69dro (the C++ IDE) file (by default in the .borland directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

Sharing items within a project

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (File | New | Other), you'll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.

Adding items to the Object Repository

You can add your own projects, forms, frames, and data modules to those already available in the Object Repository. To add an item to the Object Repository,

1 If the item is a project or is in a project, open the project.

2 For a project, choose Project | Add To Repository. For a form or data module, right-click the item and choose Add To Repository.

3 Type a description, title, and author.

4 Decide which page you want the item to appear on in the New Items dialog box, then type the name of the page or select it from the Page combo box. If you type the name of a page that doesn't exist, Kylix creates a new page.

5 Choose Browse to select an icon to represent the object in the Object Repository.

6 Choose OK.

Sharing objects in a team environment

You can share objects with your workgroup or development team by making a repository available over a network. To use a shared repository, all team members must select the same Shared Repository directory in the Environment Options dialog:

1 Choose Tools | Environment Options.

2 On the Preferences page, locate the Shared Repository panel. In the Directory edit box, enter the directory where you want to locate the shared repository. Be sure to specify a directory that's accessible to all team members.

The first time an item is added to the repository, Kylix creates a delphi69dro (Delphi) or bcb69dro (C++) file in the Shared Repository directory if one doesn't exist already.

Note It is important that the access permissions on the Object Repository directory (objrepos) are set up correctly on the because if the user does not have write permissions to the directory, they cannot add items to it. Therefore, if you want multiple users to access a common Object Repository, you should create a group and give its members read-write access to the objrepos directory. For example, if you name the group "dev," you would set the permissions on the command line by entering:

```
cd <install directory>
chmod -R 775 objrepos
chgrp -R dev objrepos
```

See the *group(5)* man page for more information.

Using an Object Repository item in a project

To access items in the Object Repository, choose File | New | Other. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project:

- Copy
- Inherit
- Use

Copying an item

Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for project templates.

Inheriting an item

Choose Inherit to derive a new class from the selected item in the Object Repository and add the new class to your project. When you recompile your project, any changes that have been made to the item in the Object Repository will be reflected in your derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.

Using an item

Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution.

The Use option is available for forms, dialog boxes, and data modules.

Using project templates

Templates are predesigned projects that you can use as starting points for your own work. To create a new project from a template:

1. Choose File | New | Other to display the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want and choose OK.
4. In the Select Directory dialog, specify a directory for the new project's files.

Kylix copies the template files to the specified directory, where you can modify them. The original project template is unaffected by your changes.

Modifying shared items

If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives:

- Copy the item and modify it in your current project only.
- Copy the item to the current project, modify it, then add it to the Repository under a different name.
- Create a component, shared object, component template, or frame from the item. If you create a component or shared object, you can share it with other developers.

Specifying a default project, new form, and main form

By default, when you choose File | New | Application or File | New | Form, Kylix displays a blank form. You can change this behavior by reconfiguring the Repository:

1. Choose Tools | Repository.
2. If you want to specify a default project, select the Projects page and choose an item under Objects. Then select the New Project check box.

3 If you want to specify a default form, select a Repository page (such as Forms), them choose a form under Objects. To specify the default new form (File|New|Form), select the New Form check box. To specify the default main form for new projects, select the Main Form check box.

4 Click OK.

Enabling Help in applications

CLX supports displaying Help from applications using an object-based mechanism that allows Help requests to be passed on to one of multiple external Help viewers (such as Man, Info, and HyperHelp). To support this, an application must include a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and registers itself with the global Help Manager.

The Help Manager maintains a list of registered viewers and passes requests to them in a two-phase process: it first asks each viewer if it can provide support for a particular Help keyword or context, and then it passes the Help request on to the viewer which says it can provide such support.

If more than one viewer supports the keyword, as would be the case in an application that had registered viewers for both WinHelp and HyperHelp on Windows or Man and Info on Linux, the Help Manager can display a selection box through which the user of the application can determine which Help viewer to invoke. Otherwise, it displays the first responding Help system encountered.

Help system interfaces

The Help system allows communication between your application and Help viewers through a series of interfaces. These interfaces are all defined in the HelpIntfs unit, which also contains the implementation of the Help Manager.

ICustomHelpViewer provides support for displaying Help based upon a provided keyword and for displaying a table of contents listing all Help available in a particular viewer.

IExtendedHelpViewer provides support for displaying Help based upon a numeric Help context and for displaying topics; in most Help systems, topics function as high-level keywords (for example, “IntToStr” might be a keyword in the Help system, but “String manipulation routines” could be the name of a topic).

*ISpecialWinHelpViewe*r provides support for responding to specialized WinHelp messages that an application running under Windows may receive and which are not easily generalizable. In general, only applications operating in the Windows environment need to implement this interface, and even then it is only required for applications that make extensive use of non-standard WinHelp messages.

IHelpManager provides a mechanism for the Help viewer to communicate back to the application's Help Manager and request additional information. *IHelpManager* is obtained at the time the Help viewer registers itself.

IHelpSystem provides a mechanism through which *TApplication* passes Help requests on to the Help system. *TApplication* obtains an instance of an object which implements both *IHelpSystem* and *IHelpManager* at application load time and exports that instance as a property; this allows other code within the application to file Help requests directly when appropriate.

IHelpSelector provides a mechanism through which the Help system can invoke the user interface to ask which Help viewer should be used in cases where more than one viewer is capable of handling a Help request, and to display a Table of Contents. This display capability is not built into the Help Manager directly to allow the Help Manager code to be identical regardless of which widget set or class library is in use.

Implementing ICustomHelpViewer

The *ICustomHelpViewer* interface contains three types of methods: methods used to communicate system-level information (for example, information not related to a particular Help request) with the Help Manager; methods related to showing Help based upon a keyword provided by the Help Manager; and methods for displaying a table of contents.

Communicating with the Help Manager

ICustomHelpViewer provides four functions that can be used to communicate system information with the Help Manager:

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- *SoftShutDown*

The Help Manager calls through these functions in the following circumstances:

- *GetViewerName* is called when the Help Manager wants to know the name of the viewer (for example, if the application is asked to display a list of all registered viewers). This information is returned via a string, and is required to be logically static (that is, it cannot change during the operation of the application). Multibyte character sets are not supported.
- *NotifyID* is called **immediately** following registration to provide the viewer with a unique cookie that identifies it. This information must be stored off for later use; if the viewer shuts down on its own (as opposed to in response to a notification from the Help Manager), it must provide the Help Manager with the identifying cookie so that the Help Manager can release all references to the viewer. (Failing to provide the cookie, or providing the wrong one, causes the Help Manager to potentially release references to the wrong viewer.)

- *ShutDown* is called by the Help Manager to notify the Help viewer that the Manager is shutting down and that any resources the Help viewer has allocated should be freed. It is recommended that all resource freeing be delegated to this method.
- *SoftShutDown* is called by the Help Manager to ask the Help viewer to close any externally visible manifestations of the Help system (for example, windows displaying Help information) without unloading the viewer.

Asking the Help Manager for information

Help viewers communicate with the Help Manager through the *IHelpManager* interface, an instance of which is returned to them when they register with the Help Manager. *IHelpManager* allows the Help viewer to communicate four things:

- A request for the window handle of the currently active control.
- A request for the name of the Help file which the Help Manager believes should contain help for the currently active control.
- A request for the path to that Help file.
- A notification that the Help viewer is shutting itself down in response to something other than a request from the Help Manager that it do so.

GetHandle is called by the Help viewer if it needs to know the handle of the currently active control; the result is a window handle.

GetHandle is called by the Help viewer if it wishes to know the name of the Help file which the currently active control believes contains its help.

Release is called to notify the Help Manager when a Help viewer is disconnecting. It should never be called in response to a request through *ShutDown*; it is only used to notify the Help Manager of unexpected disconnects.

Displaying keyword-based Help

Help requests typically come through to the Help viewer as either *keyword-based* Help, in which case the viewer is asked to provide help based upon a particular string, or as *context-based* Help, in which case the viewer is asked to provide help based upon a particular numeric identifier.

Numeric help contexts are the default form of Help requests in applications running under Windows, which use the WinHelp system; while CLX supports them, they are not recommended for use in Linux applications because most Linux Help systems do not understand them.

ICustomHelpViewer implementations are required to provide support for keyword-based Help requests, while *IExtendedHelpViewer* implementations are required to support context-based Help requests.

ICustomHelpViewer provides three methods for handling keyword-based Help:

- *UnderstandsKeyword*
- *GetHelpStrings*

- *ShowHelp*

```
ICustomHelpViewer.UnderstandsKeyword(const HelpString: String): Integer
int__fastcall ICustomHelpViewer::UnderstandsKeyword(const AnsiString HelpString)
```

is the first of the three methods called by the Help Manager, which will call *each* registered Help viewer with the same string to ask if the viewer provides help for that string; the viewer is expected to respond with an integer indicating how many different Help pages it can display in response to that Help request. The viewer can use any method it wants to determine this — inside the IDE, the HyperHelp viewer maintains its own index and searches it. If the viewer does not support help on this keyword, it should return zero. Negative numbers are currently interpreted as meaning zero, but this behavior is not guaranteed in future releases.

```
ICustomHelpViewer.GetHelpStrings(const HelpString: String): TStringList
```

```
Classes::TStringList*__fastcall ICustomHelpViewer::GetHelpStrings(const AnsiString
HelpString)
```

is called by the Help Manager if more than one viewer can provide Help on a topic. The viewer is expected to return a *TStringList*. The strings in the returned list should map to the pages available for that keyword, but the characteristics of that mapping can be determined by the viewer. In the case of the HyperHelp viewer on Linux, the string list always contains exactly one entry. HyperHelp provides its own indexing, and duplicating that elsewhere would be pointless duplication. In the case of the Man page viewer (Linux), the string list consists of multiple strings, one for each section of the manual which contains a page for that keyword.

```
ICustomHelpViewer.ShowHelp(const HelpString: String)
```

```
void__fastcall ICustomHelpViewer::ShowHelp(const AnsiString HelpString)
```

is called by the Help Manager if it needs the Help viewer to display help for a particular keyword. This is the last method call in the operation; it is guaranteed to never be called unless the *UnderstandsKeyword* method is invoked first.

Displaying tables of contents

ICustomHelpViewer provides two methods relating to displaying tables of contents:

- *CanShowTableOfContents*
- *ShowTableOfContents*

The theory behind their operation is similar to the operation of the keyword Help request functions: the Help Manager first queries all Help viewers by calling the *CanShowTableOfContents* method and then invokes a particular Help viewer by calling the *ShowTableOfContents* method.

It is reasonable for a particular viewer to refuse to allow requests to support a table of contents. The Man page viewer does this, for example, because the concept of a table of contents does not map well to the way Man pages work; the HyperHelp viewer supports a table of contents, on the other hand, by passing the request to display a table of contents directly to HyperHelp. It is *not* reasonable, however, for an implementation of *ICustomHelpViewer* to respond to queries through

CanShowTableOfContents with the answer true, and then ignore requests through *ShowTableOfContents*.

Implementing IExtendedHelpViewer

ICustomHelpViewer only provides direct support for keyword-based Help. Some Help systems (especially WinHelp) work by associating numbers (known as *context IDs*) with keywords in a fashion which is internal to the Help system and therefore not visible to the application. Such systems require that the application support context-based Help in which the application invokes the Help system with that context, rather than with a string, and the Help system translates the number itself.

Applications written in CLX can talk to systems requiring context-based Help by extending the object that implements *ICustomHelpViewer* to also implement *IExtendedHelpViewer*. *IExtendedHelpViewer* also provides support for talking to Help systems that allow you to jump directly to high-level topics instead of using keyword searches.

IExtendedHelpViewer exposes four functions. Two of them—*UnderstandsContext* and *DisplayHelpByContext*—are used to support context-based Help; the other two—*UnderstandsTopic* and *DisplayTopic*—are used to support topics.

When an application user presses F1, the Help Manager calls

```
IExtendedHelpViewer.UnderstandsContext(const ContextID: Integer;
const HelpFileName: String): Boolean
```

```
int__fastcall IExtendedHelpViewer::UnderstandsContext(const int ContextID, AnsiString
HelpFileName)
```

and the currently activated control supports context-based, rather than keyword-based Help. As with the *ICustomHelpViewer*'s *UnderstandsKeyword* method, the Help Manager queries all registered Help viewers iteratively. Unlike the case with *UnderstandsKeyword*, however, if more than one viewer supports a specified context, the *first* registered viewer with support for a given context is invoked.

The Help Manager calls

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID: Integer;
const HelpFileName: String)
void__fastcall IExtendedHelpViewer::DisplayHelpByContext(const int ContextID, AnsiString
HelpFileName)
```

after it has polled the registered Help viewers.

The topic support functions work the same way:

```
IExtendedHelpViewer.UnderstandsTopic(const Topic: String): Boolean
```

```
bool__fastcall IExtendedHelpViewer::UnderstandsTopic(const AnsiString Topic)
```

is used to poll the Help viewers asking if they support a topic;

```
IExtendedHelpViewer.DisplayTopic(const Topic: String)
```

```
void__fastcall IExtendedHelpViewer::DisplayTopic(const AnsiString Topic)
```

is used to invoke the first registered viewer which reports that it is able to provide help for that topic.

Implementing IHelpSelector

IHelpSelector is a companion to *ICustomHelpViewer*. When more than one registered viewer claims to provide support for a given keyword, context, or topic, or provides a table of contents, the Help Manager must choose between them. In the case of contexts or topics, the Help Manager *always* selects the first Help viewer that claims to provide support. In the case of keywords or the table of context, the Help Manager will, by default, select the first Help viewer. This behavior can be overridden by an application.

To override the decision of the Help Manager in such cases, an application must register a class that provides an implementation of the *IHelpSelector* interface. *IHelpSelector* exports two functions: *SelectKeyword*, and *TableOfContents*. Both take as arguments a *TStrings* containing, one by one, either the possible keyword matches or the names of the viewers claiming to provide a table of contents. The implementor is required to return the index (in the *TStringList*) that represents the selected string.

Note The Help Manager may get confused if the strings are rearranged; it is recommended that implementors of *IHelpSelector* refrain from doing this. The Help system only supports *one* HelpSelector; when new selectors are registered, any previously existing selectors are disconnected.

Registering Help system objects

For the Help Manager to communicate with them, objects that implement *ICustomHelpViewer, IExtendedHelpViewer, ISpecialWinHelpViewer,* and *IHelpSelector* must register with the Help Manager.

To register Help system objects with the Help Manager, you need to:

- Register the Help viewer.
- Register the Help Selector.

Registering Help viewers

The unit that contains the object implementation must use HelpIntfs. An instance of the object must be declared in the var section (Delphi) header file (C++) of the implementing unit.

In Delphi, the initialization section of the implementing unit must assign the instance variable and pass it to the function *RegisterViewer.* In C++, the implementing unit must include a pragma startup directive that calls a method that assigns the instance variable and passes it to the function *RegisterViewer*. *RegisterViewer* is a flat function exported by the HelpIntfs.unit, which takes as an argument an *ICustomHelpViewer* and returns an *IHelpManager*. The *IHelpManager* should be stored for future use.

C++ example

In C++, the corresponding .cpp file contains the code to register the interface. For the interface described above, this registration code looks like the following:

```
void InitServices()
{
  THelpImplementor GlobalClass;
  Global = dynamic_cast<ICustomHelpViewer*>(GlobalClass);
  Global->AddRef;
  HelpIntfs::RegisterViewer(Global, GlobalClass->Manager);
}
#pragma startup InitServices
```

Note In C++, the Help Manager object must be freed in the destructor for the GlobalClass object if it has not already been freed.

Registering Help selectors

The unit that contains the object implementation must use QForms. An instance of the object must be declared in the var section (Delphi) or .cpp file (C++) of the implementing unit.

The implementing unit (in Delphi, the initialization section) must register the Help selector through the *HelpSystem* property of the global Application object:

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

```
Application->HelpSystem->AssignHelpSelector(myHelpSelectorInstance)
```

This procedure does not return a value.

Using Help in a cross-platform application

The following sections explain how to use Help within a cross-platform application.

- How TApplication processes cross-platform Help
- How cross-platform controls process Help
- Calling a Help system directly
- Using IHelpSystem

How TApplication processes cross-platform Help

TApplication in CLX provides two methods that are accessible from application code:

- *ContextHelp*, which invokes the Help system with a request for context-based Help
- *KeywordHelp*, which invokes the Help system with a request for keyword-based Help

Both functions take as an argument the context or keyword being passed and forward the request on through a data member of *TApplication,* which represents the Help system. That data member is directly accessible through the read-only property *HelpSystem*.

How cross-platform controls process Help

All controls that derive from *TControl* expose four properties which are used by the Help system: *HelpType*, *HelpFile*, *HelpContext*, and *HelpKeyword*. *HelpFile* is supposed to contain the name of the file in which the control's help is located; if the help is located in an external Help system that does not care about file names (say, for example, the Man page system), then the property should be left blank.

The *HelpType* property contains an instance of an enumerated type which determines if the control's designer expects help to be provided via keyword-based Help or context-based Help; the other two properties are linked to it. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, which can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWidgetControl* calls *InvokeHelp*.

Calling a Help system directly

For additional Help system functionality not provided by CLX, *TApplication* provides a read-only property that allows direct access to the Help system. This property is an instance of an implementation of the interface *IHelpSystem*. *IHelpSystem* and *IHelpManager* are implemented by the same object, but one interface is used to allow the application to talk to the Help Manager, and one is used to allow the Help viewers to talk to the Help Manager.

Using IHelpSystem

IHelpSystem allows a cross-platform application to do three things:

- Provides path information to the Help Manager.
- Provides a new Help selector.
- Asks the Help Manager to display Help.

Providing path information is important because the Help Manager is platform-independent and Help system-independent and so is not able to ascertain the location of Help files. If an application expects help to be provided by an external Help system that is not able to ascertain file locations itself, it must provide this information through the *IHelpSystem's ProvideHelpPath* method, which allows the information to become available through the *IHelpManager's GetHelpPath* method. (This information propagates outward only if the Help viewer asks for it.)

Assigning a Help selector allows the Help Manager to delegate decision-making in cases where multiple external Help systems can provide Help for the same keyword. For more information, see "Implementing IHelpSelector" on page 8-23.

IHelpSystem exports four procedures and one function to request the Help Manager to display Help:

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

Hook is intended entirely for WinHelp compatibility and should not be used in a Linux-only application; it allows processing of WM_HELP messages that cannot be mapped directly onto requests for keyword-based, context-based, or topic-based Help. The other methods each take two arguments: the keyword, context ID, or topic for which help is being requested, and the Help file in which it is expected that help can be found.

In general, unless you are asking for topic-based help, it is equally effective and more clear to pass help requests to the Help Manager through the *InvokeHelp* method of your control.

Customizing the IDE Help system

The IDE supports multiple Help viewers in exactly the same way that a CLX application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE makes use of the same WinHelpViewer that CLX uses.

The IDE comes with two Help viewers installed: the HyperHelp viewer, which allows Help requests to be forwarded to HyperHelp, an external WinHelp emulator under which the Kylix Help files are viewed, and the Man page viewer, which allows you to access the Man system installed on most Unix machines. Because it is necessary for Kylix Help to work, the HyperHelp viewer may not be removed; the Man page viewer ships in a separate package whose source is available in the examples directory.

To install a new Help viewer in the IDE, you do exactly what you would do in a CLX application, with one difference. You write an object that implements *ICustomHelpViewer* (and, if desired, *IExtendedHelpViewer*) to forward Help requests to the external viewer of your choice, and you register the *ICustomHelpViewer* with the IDE.

To register a custom Help viewer with the IDE:

1 Make sure that the unit implementing the Help viewer contains the HelpIntfs unit.

2 Build the unit into a design-time package registered with the IDE, and build the package with runtime packages turned on. (This is necessary to ensure that the Help Manager instance used by the unit is the same as the Help Manager instance used by the IDE.)

3 Make sure that the Help viewer exists as a global instance within the unit.

4 In the initialization section of the unit (Delphi) or function blocked by #pragma startup (C++) , make sure that the instance is passed to the *RegisterHelpViewer* function.

Chapter

9

Developing the application user interface

When you open the IDE or create a new project, a blank form is displayed on the screen. You design your application's user interface (UI) by placing and arranging visual components, such as windows, menus, and dialog boxes, from the Component palette onto the form.

Once a visual component is on the form, you can adjust its position, size, and other design-time properties, and code its event handlers. The Form Designer takes care of the underlying programming details.

The following sections describe some of the major interface tasks, such as working with forms, creating component templates, adding dialog boxes, and organizing actions for menus and toolbars.

Controlling application behavior

TApplication, *TScreen*, and *TForm* are the classes that form the backbone of all Kylix applications by controlling the behavior of your project. The *TApplication* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard Linux program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as maintaining system-specific information such as screen resolution and available display fonts. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm*.

D In Delphi, you can set the form's *Visible* property to *False* using the Object Inspector at design time rather than setting it at runtime as shown above.

Working at the application level

The global variable *Application*, of type *TApplication*, is in every CLX-based application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* handles how you call a Help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the Project|Options Application page when you create a project.

In addition, *Application* receives many events that apply to the application as a whole. For example, the *OnActivate* event lets you perform actions when the application first starts up, the *OnIdle* event lets you perform background processes when the application is not busy, the *OnEvent* event lets you intercept events, and so on. Although you can't use the IDE to examine the properties and events of the global *Application* variable, another component, *TApplicationEvents*, intercepts the events and lets you supply event-handlers using the IDE.

Handling the screen

A global variable of type *TScreen* called *Screen* is created when you create a project. Screen encapsulates the state of the screen on which your application is running. Common tasks performed by *Screen* include specifying:

- The look of the cursor.
- The size of the window in which your application is running.
- A list of fonts available to the screen device.

For cross-platform programs, the default behavior is that applications create a screen component based on information about the current screen device and assign it to *Screen*.

Setting up forms

TForm is the key class for creating GUI applications. When you open Kylix displaying a default project or when you create a new project, a form appears on which you can begin your UI design.

Using the main form

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form:

1 Choose Project | Options and select the Forms page.

2 In the Main Form combo box, select the form you want to use as the project's main form and choose OK.

Now if you run the application, the form you selected as the main form is displayed.

Hiding the main form

You can prevent the main form from displaying when your application starts. To do so, you must use the global *Application* variable (described in "Working at the application level" on page 9-2).

To hide the main form at startup:

1 Choose Project | View Source to display the main project file.

2 Add the following code after the call to *TApplication's CreateForm* method and before the call to the *Run* method.

```
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
```

```
Application->ShowMainForm = false;
```

3 In C++, use the Object Inspector to set the *Visible* property of your main form to false.

Adding forms

To add a form to your project, select File | New | Form. You can see all your project's forms and their associated units listed in the Project Manager (View | Project Manager) and you can display a list of the forms alone by choosing View | Forms.

Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking*.

A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to enable a form that contains data-aware components to connect to the data-access components in a data module.

To link a form to another form,

1 Select the form that needs to refer to another.
2 Choose File | Use Unit (the Delphi IDE) or File | Include Unit Hdr (the C++ IDE).
3 Select the name of the form unit for the form to be referenced.
4 Choose OK.

Linking a form to another just means that one form unit contains either a reference in its uses clause to the other's form unit (Delphi) or the header for the other's form unit

(C++), meaning that the linked form and its components are now in scope for the linking form.

Avoiding circular unit references in Delphi

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both uses clauses, with the unit identifiers, in the implementation parts of the respective unit files. (This is what the File|Use Unit command does.)
- Place one uses clause in an interface part and the other in an implementation part. (You rarely need to place another form's unit identifier in this unit's interface part.)

Do not place both uses clauses in the interface parts of their respective unit files. This will generate the "Circular reference" error at compile time.

Managing layout

At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms.

Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

Two properties affect how a control is positioned and sized in relation to its parent. The *Align* property lets you force a control to fit perfectly within its parent along a specific edge or filling up the entire client area after any other controls have been aligned. When the parent is resized, the controls aligned to it are automatically resized and remain positioned so that they fit against a particular edge.

If you want to keep a control positioned relative to a particular edge of its parent, but don't want it to necessarily touch that edge or be resized so that it always runs along the entire edge, you can use the *Anchors* property.

If you want to ensure that a control does not grow too big or too small, you can use the *Constraints* property. *Constraints* lets you specify the control's maximum height, minimum height, maximum width, and minimum width. Set these to limit the size (in pixels) of the control's height and width. For example, by setting the *MinWidth* and *MinHeight* of the constraints on a container object, you can ensure that child objects are always visible.

The value of *Constraints* propagates through the parent/child hierarchy so that an object's size can be constrained because it contains aligned children that have size constraints. *Constraints* can also prevent a control from being scaled in a particular dimension when its *ChangeScale* method is called.

TControl introduces a protected event, *OnConstrainedResize*, of type *TConstrainedResizeEvent*:

D
```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
MaxHeight: Integer) of object;
```

C++
```
void __fastcall (__closure *TConstrainedResizeEvent)(System::TObject* Sender, int &MinWidth,
int &MinHeight, int &MaxWidth, int &MaxHeight);
```

This event allows you to override the size constraints when an attempt is made to resize the control. The values of the constraints are passed as `var` parameters which can be changed inside the event handler. *OnConstrainedResize* is published for container objects (*TForm*, *TScrollBox*, *TControlBar*, and *TPanel*). In addition, component writers can use or publish this event for any descendant of *TControl*.

Controls that have contents that can change in size have an *AutoSize* property that causes the control to adjust its size to its font or contained objects.

Using forms

When you create a form from the IDE, Kylix automatically creates the form in memory by including code in the main entry point of your application function. Usually, this is the desired behavior and you don’t have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default Kylix behavior when creating the form for your main window.

However, you may not want all your application’s forms in memory for the duration of the program execution. That is, if you do not want all your application’s dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

Controlling when forms reside in memory

By default, Kylix automatically creates the application’s main form in memory by including the following code in the application’s project source unit (Delphi) or main entry point (C++):

D
```
Application.CreateForm(TForm1, Form1);
```

C++
```
Application ->CreateForm(__classid(TForm1), &Form1);
```

This function creates a global variable with the same name as the form. So, every form in an application has an associated global variable. This variable is a pointer to an instance of the form’s class and is used to reference the form while the application is running. Any unit that in Delphi includes the form’s unit in its **uses** clause or in C++ includes the source code (.cpp) file that includes the form’s header (.h) file can access the form using this variable.

In Delphi, all forms created in this way in the project unit appear when the program is invoked and exist in memory for the duration of the application.

In C++, because the form is added to the application's main entry point, the form appears when the program is invoked and it exists in memory for the duration of the application.

Displaying an auto-created form

If you choose to create a form at startup, and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:

Delphi example

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
   ResultsForm.ShowModal;
end;
```

C++ example

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
   ResultsForm->ShowModal();
}
```

In this case, since the form is already in memory, there is no need to create another instance or destroy that instance.

Creating forms dynamically

You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.

To create a form at a different stage during execution using the IDE, you:

1. Select the File | New | Form from the main menu to display the new form.
2. Remove the form from the Auto-create forms list of the Project | Options | Forms page.

   This removes the form's invocation at startup. As an alternative, you can manually remove the following line from program's main entry point:

   ```
   Application.CreateForm(TResultsForm, ResultsForm);
   ```

   ```
   Application->CreateForm(__classid(TResultsForm), &ResultsForm);
   ```
3. Invoke the form when desired by using the form's *Show* method, if the form is modeless, or *ShowModal* method, if the form is modal.

An event handler for the main form must create an instance of the result form and destroy it. One way to invoke the result form is to use the global variable as follows. Note that *ResultsForm* is a modal form so the handler uses the *ShowModal* method.

Delphi example

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm := TResultForm.Create(self);
try
  ResultsForm.ShowModal;
finally
  ResultsForm.Free;
end;
```

In the above Delphi example, note the use of **try..finally**. Putting in the line `ResultsForm.Free;` in the **finally** clause ensures that the memory for the form is freed even if the form raises an exception.

C++ example

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
   ResultsForm = new TResultsForm(this);
 ResultsForm->ShowModal();
 delete ResultsForm;
}
```

The event handler in the example deletes the form after it is closed, so the form would need to be recreated (using the **new** operator in C++) if you needed to use *ResultsForm* elsewhere in the application. If the form were displayed using *Show* you could not delete the form within the event handler because *Show* returns while the form is still open.

Note If you create a form using its constructor (Delphi) or the **new** operator (C++), be sure to check that the form is not in the Auto-create forms list on the Project Options | Forms page. Specifically, if you create the new form without deleting the form of the same name from the list, Kylix creates the form at startup and this event-handler creates a new instance of the form, overwriting the reference to the auto-created instance. The auto-created instance still exists, but the application can no longer access it. After the event-handler terminates, the global variable no longer points to a valid form. Any attempt to use (Delphi) or dereference (C++) the global variable will likely crash the application.

Creating modeless forms such as windows

You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables (in C++, of type pointer to the form class) for each instance.

Creating a form instance using a local variable

A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:

D **Delphi example**

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

C++ example

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
   TResultsForm *rf = new TResultsForm(this);// rf is local form instance
 rf->ShowModal();
 delete rf;       // form safely destroyed
}
```

Notice how the global instance of the form is never used in this version of the event handler.

Typically, applications use the global instances of forms. However, if you need a new instance of a modal form, and you use that form in a limited, discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Of course, you cannot use local variables in event handlers for modeless forms because they must have global scope to ensure that the forms exist for as long as the form is in use. *Show* returns as soon as the form opens, so if you used a local variable, the local variable would go out of scope immediately.

Passing additional arguments to forms

Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is (a pointer to in C++) the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be nil (Delphi) or NULL (C++).

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor (Delphi) or the **new** operator (C++). The example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.

Delphi example

```
TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
```

C++ example

```
class TResultsForm : public TForm
{
__published:    // IDE-managed Components
   TLabel *ResultsLabel;
   TButton *OKButton;
   void __fastcall OKButtonClick(TObject *Sender);
private:        // User declarations
public:         // User declarations
   virtual __fastcall TResultsForm(TComponent* Owner);
   virtual __fastcall TResultsForm(int whichButton, TComponent* Owner);

};
```

Here's the manually coded constructor that passes the additional argument, *whichButton*. This constructor uses the *whichButton* parameter to set the *Caption* property of a *Label* control on the form.

Delphi example

```
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
  inherited Create(Owner);
  case whichButton of
    1: ResultsLabel.Caption := 'You picked the first button.';
    2: ResultsLabel.Caption := 'You picked the second button.';
    3: ResultsLabel.Caption := 'You picked the third button.';
  end;
end;
```

C++ example

```
void__fastcall TResultsForm::TResultsForm(int whichButton, TComponent* Owner)
 : TForm(Owner)
{
   switch (whichButton) {
      case 1:
         ResultsLabel->Caption = "You picked the first button!";
         break;
      case 2:
         ResultsLabel->Caption = "You picked the second button!";
         break;
      case 3:
            ResultsLabel->Caption = "You picked the third button!";
        }
    }
```

When creating an instance of a form with multiple constructors, you can select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* that uses the extra parameter:

Delphi example

```
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;
```

C++ example

```
void __fastcall TMainMForm::SecondButtonClick(TObject *Sender)
{
   TResultsForm *rf = new TResultsForm(2, this);
   rf->ShowModal();
   delete rf;
}
```

Retrieving data from forms

Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.

Retrieving data from modeless forms

You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red," "Green," "Blue," and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:

Delphi example

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
```

```
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
```

C++ example

```
class TColorForm : public TForm
{
   __published:    // IDE-managed Components
      TListBox *ColorListBox;
      void __fastcall ColorListBoxClick(TObject *Sender);
   private:        // User declarations
      String getColor();
      void   setColor(String);
      String curColor;
   public:         // User declarations
      virtual __fastcall TColorForm(TComponent* Owner);
      __property String CurrentColor = {read=getColor, write=setColor};
};
```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The *CurrentColor* property uses the setter function, *SetColor* (Delphi) or *setColor* (C++), to store the actual value for the property in the private data member *FColor* (Delphi) or *curColor* (C++):

Delphi example

```
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;
```

C++ example

```
void __fastcall TColorForm::ColorListBoxClick(TObject *Sender)
{
   int index = ColorListBox->ItemIndex;
   if (index >= 0) {// make sure a color is selected
      CurrentColor = ColorListBox->Items->Strings[index];
   }
   else          // no color selected
      CurrentColor = "";
}
//---------------------------------------------------------------------
```

```
void TColorForm::setColor(String s)
{
   curColor = s;
}
```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button (called *UpdateButton)* on *ResultsForm* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

Delphi example

```
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {do something with the string MainColor}
  end;
end;
```

C++ example

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
   if (ColorForm) {// verify ColorForm exists
      String s = ColorForm->CurrentColor;
      // do something with the color name string
   }
}
```

The event handler first verifies that *ColorForm* exists; in Delphi, by using the *Assigned* function and in C++, by by checking whether the point is NULL. It then gets the value of *ColorForm's CurrentColor* property.

In C++, the query of *CurrentColor* calls its getter function *getColor* which is shown here:

```
String TColorForm::getColor()
{
   return curColor;
}
```

Alternatively, if in Delphi, *ColorForm* had a public function named *GetColor* and in C++, *ColorForm's getColor* function were public, another form could get the current color without using the *CurrentColor* property. For example:

```
MainColor := ColorForm.GetColor;
```

```
String s = ColorForm->getColor();
```

In fact, there's nothing to prevent another form from getting the *ColorForm's* currently selected color by checking the listbox selection directly:

```
with ColorForm.ColorListBox do
```

```
  MainColor := Items[ItemIndex];
```

```
String s = ColorListBox->Items->Strings[ColorListBox->ItemIndex];
```

However, using a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

Retrieving data from modal forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is when form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit return value, you must preserve important information from the form before it is destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

Delphi example

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

C++ example

```
class TColorForm : public TForm
{
   __published:    // IDE-managed Components
      TListBox *ColorListBox;
      TButton *SelectButton;
      TButton *CancelButton;
      void __fastcall CancelButtonClick(TObject *Sender);
      void __fastcall SelectButtonClick(TObject *Sender);
   private:        // User declarations
      String* curColor;
   public:         // User declarations
      virtual __fastcall TColorForm(TComponent* Owner);
      virtual __fastcall TColorForm(String* s, TComponent* Owner);
};
```

The form has a listbox called *ColorListBox* with a list of names of colors. When pressed, the button called *SelectButton* makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button that simply closes the form.

Note that a user-defined constructor was added to the class that takes a *Pointer* (Delphi) or *String** (C++) argument. Presumably, this *Pointer* (Delphi) or *String** (C++) points to a string that the form launching *ColorForm* knows about. The implementation of this constructor is as follows:

Delphi example

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

C++ example

```
void__fastcall TColorForm::TColorForm(String* s, TComponent* Owner)
 : TForm(Owner)
{
   curColor = s;
   *curColor = "";
}
```

The constructor saves the pointer to a private data member *FColor* (Delphi) or *curColor* (C++) and initializes the string to an empty string.

Note To use the above user-defined constructor, the form must be explicitly created. It cannot be auto-created when the application is started. For details, see “Controlling when forms reside in memory” on page 9-5.

In the application, the user selects a color from the listbox and presses *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* might look like this:

Delphi example

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
```

C++ example

```
void __fastcall TColorForm::SelectButtonClick(TObject *Sender)
{
   int index = ColorListBox->ItemIndex;
```

```
    if (index >= 0)
       *curColor = ColorListBox->Items->Strings[index];
    Close();
 }
```

Notice that the event handler stores the selected color name in the string referenced by the pointer (Delphi) or string address (C++) that was passed to the constructor.

To use *ColorForm* effectively, the calling form must pass the constructor a pointer to an existing string. For example, assume *ColorForm* was instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked. The event handler would look as follows:

Delphi example

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {do something with the MainColor string}
  else
    {do something else because no color was picked}
end;

procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
```

C++ example

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
   String s;
   GetColor(&s);
   if (s != "") {
      // do something with the color name string
   }
   else {
      // do something else because no color was picked
   }
}
//----------------------------------------------------------------------
void TResultsForm::GetColor(String *s)
{
   ColorForm = new TColorForm(s, this);
   ColorForm->ShowModal();
   delete ColorForm;
   ColorForm = 0; // NULL the pointer
}
```

UpdateButtonClick creates a String called MainColor (Delphi) or *s* (C++). The address of MainColor (Delphi) or *s* (C++) is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to MainColor (Delphi) or *s* (C++) as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in MainColor (Delphi) or *s* (C++), assuming that a color was selected. Otherwise, MainColor (Delphi) or *s* (C++) contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This example uses one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. Keep in mind that you should always provide a way to let the calling form know if the modal form was closed without making any changes or selections (such as having MainColor (Delphi) or *s* (C++) default to an empty string).

Reusing components and groups of components

The IDE offers several ways to save and reuse work you've done with components:

- *Component templates* provide a simple, quick way of configuring and saving groups of components. See "Creating and using component templates" on page 9-16.
- You can save forms, data modules, and projects in the *Repository*. This gives you a central database of reusable elements and lets you use form inheritance to propagate changes. See "Using the Object Repository" on page 8-15.
- You can save *frames* on the Component palette or in the repository. Frames use form inheritance and can be embedded into forms or other frames. See "Working with frames" on page 9-17.
- Creating a *custom component* is the most complicated way of reusing code, but it offers the greatest flexibility. See Chapter 35, "Overview of component creation."

Creating and using component templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the Component palette, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time.

Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

To create a component template,

1. Place and arrange components on a form. In the Object Inspector, set their properties and events as desired.

2 Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.

3 Choose Component | Create Component Template.

4 Specify a name for the component template in the Component Template Information edit box. The default proposal is the component type of the first component selected in step 2 followed by the word "Template." For example, if you select a label and then an edit box, the proposed name will be "TLabelTemplate." You can change this name, but be careful not to duplicate existing component names.

5 In the Palette page edit box, specify the Component palette page where you want the template to reside. If you specify a page that does not exist, a new page is created when you save the template.

6 Next to Palette Icon, select a bitmap to represent the template on the palette. The default proposal will be the bitmap used by the component type of the first component selected in step 2. To browse for other bitmaps, click Change. The bitmap you choose must be no larger than 24 pixels by 24 pixels.

7 Click OK.

To remove templates from the Component palette, choose Component | Configure Palette.

Working with frames

A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties.

In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the Component palette for easy reuse, and they can be nested within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from the components (including other frames) it contains. When a frame is embedded in another frame or form, it continues to inherit changes made to the frame from which it derives.

Frames are useful to organize groups of controls that are used in multiple places in your application. For example, if you have a bitmap that is used on multiple forms, you can put it in a frame and only one copy of that bitmap is included in the resources of your application. You could also describe a set of edit fields that are intended to edit a table with a frame and use that whenever you want to enter data into the table.

Creating frames

To create an empty frame, choose File | New | Frame, or choose File | New | Other and double-click Frame. You can then drop components (including other frames) onto your new frame.

It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose File | New | Application, close the new form and unit without saving them, then choose File | New | Frame and save the project.

Note When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these are likely to cause conflicts when you try to use the frames later.

At design time, you can display any frame included in the current project by choosing View | Forms and selecting a frame. As with forms and data modules, you can toggle between the Form Designer and the frame's form file by right-clicking and choosing View as Form or View as Text.

Adding frames to the Component palette

Frames are added to the Component palette as component templates. To add a frame to the Component palette, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click the frame, and choose Add to Palette. When the Component Template Information dialog opens, select a name, palette page, and icon for the new template.

Using and modifying frames

To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes.

The Form Designer provides two ways to add a frame to an application:

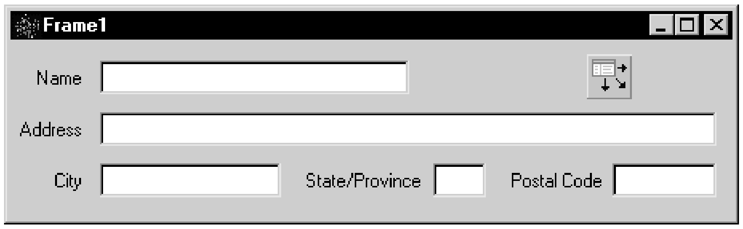
- Select a frame from the Component palette and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project.
- Select *Frames* from the Standard page of the Component palette and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

When you drop a frame onto a form or other container, Kylix declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Kylix declares a new class that descends from *TForm*.) This means that changes made later to the original (ancestor) frame propagate to the embedded frame, but changes to the embedded frame do not propagate backward to the ancestor.

Suppose, for example, that you wanted to assemble a group of data-access components and data-aware controls for repeated use, perhaps in more than one application. One way to accomplish this would be to collect the components into a component template; but if you started to use the template and later changed your mind about the arrangement of the controls, you would have to go back and manually alter each project where the template was placed.

If, on the other hand, you put your database components into a frame, later changes would need to be made in only one place; changes to an original frame automatically propagate to its embedded descendants when your projects are recompiled. At the same time, you are free to modify any embedded frame without affecting the original frame or other embedded descendants of it. The only limitation on modifying embedded frames is that you cannot add components to them.

Figure 9.1 A frame with data-aware controls and a data source component



In addition to simplifying maintenance, frames can help you to use resources more efficiently. For example, to use a bitmap or other graphic in an application, you might load the graphic into the *Picture* property of a *TImage* control. If, however, you use the same graphic repeatedly in one application, each *Imag*e object you place on a form will result in another copy of the graphic being added to the form's resource file. (This is true even if you set *Picture* property once and save the *Image* control as a component template.) A better solution is to drop the *Image* object onto a frame, load your graphic into it, then use the frame where you want the graphic to appear. This results in smaller form files and has the added advantage of letting you change the graphic everywhere it occurs simply by modifying the *Image* on the original frame.

Sharing frames

You can share a frame with other developers in two ways:

- Add the frame to the Object Repository.
- Distribute the frame's unit (.pas. in Delphi and cpp and .h in C++) and form (.xfm) files.

To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository. For more information, see "Using the Object Repository" on page 8-15.

If you send a frame's unit and form files to other developers, they can open them and add them to the Component palette. If the frame has other frames embedded in it, the frame must be opened as part of a project to add it to the Component palette.

Developing dialog boxes

The dialog box components on the Dialogs page of the Component palette make various dialog boxes available to your applications. These dialog boxes provide applications with a familiar, consistent interface that enables the user to perform common file operations such as opening, saving, and printing files. Dialog boxes display and/or obtain data.

Each dialog box opens when its *Execute* method is called. *Execute* returns a Boolean value: if the user chooses OK to accept any changes made in the dialog box, *Execute* returns true; if the user chooses Cancel to escape from the dialog box without making or saving changes, *Execute* returns false.

Note You can use the dialogs provided with CLX in the QDialogs unit. For operating systems that have native dialog box types for common tasks, such as for opening or saving a file or for changing font or color, you can use the *UseNativeDialog* property. Set *UseNativeDialog* to true if your application will run in such an environment, and if you want it to use the native dialogs instead of the Qt dialogs.

Using open dialog boxes

One of the commonly used dialog box components is *TOpenDialog*. This component is usually invoked by a New or Open menu item under the File option on the main menu bar of a form. The dialog box contains controls that let you select groups of files using a wildcard character and navigate through directories.

The *TOpenDialog* component makes an Open dialog box available to your application. The purpose of this dialog box is to let a user specify a file to open. You use the *Execute* method to display the dialog box.

When the user chooses OK in the dialog box, the user's file is stored in the *TOpenDialog FileName* property, which you can then process as you want.

The following code can be placed in an *Action* and linked to the *Action* property of a *TMainMenu* subitem or be placed in the subitem's *OnClick* event:

Delphi example

```
if OpenDialog1.Execute then
   filename := OpenDialog1.FileName;
```

C++ example

```
if(OpenDialog1->Execute())
{
   filename = OpenDialog1->FileName;
};
```

This code will show the dialog box and if the user presses the OK button, it will copy the name of the file into a previously declared *AnsiString* variable named `filename`.

Organizing actions for toolbars and menus

Kylix provides several features that simplify the work of creating, customizing, and maintaining menus and toolbars. These features allow you to organize lists of actions that users of your application can initiate by pressing a button on a toolbar, choosing a command on a menu, or pointing and clicking on an icon.

Often a set of actions is used in more than one user interface element. For example, the Cut, Copy, and Paste commands often appear on both an Edit menu and on a toolbar. You only need to add the action once to use it in multiple UI elements in your application.

The following table defines the terminology related to setting up menus and toolbars:

Table 9.1 Action setup terminology

Term	Definition
Action	A response to something a user does, such as clicking a menu item. Many standard actions that are frequently required are provided for you to use in your applications as is. For example, file operations such as File Open, File SaveAs, File Run, and File Exit are included along with many others for editing, formatting, searches, help, dialogs, and window actions. You can also program custom actions and access them using action lists.
Action category	Lets you group actions and drop them as a group onto a menu or toolbar. For example, one of the standard action categories is Search which includes Find, FindFirst, FindNext, and Replace actions all at once.
Action classes	Classes that perform the actions used in your application. All of the standard actions are defined in action classes such as *TEditCopy*, *TEditCut*, and *TEditUndo*. You can use these classes by dragging and dropping them from the Customize dialog onto an action band.
Action client	Most often represents a menu item or a button that receives a notification to initiate an action. When the client receives a user command (such as a mouse click), it initiates an associated action.
Action list	Maintains a list of actions that your application can take in response to something a user does.
Menu	Lists commands that the user of the application can execute by clicking on them. You can create menus by using cross-platform components such as *TMainMenu* or *TPopupMenu*.
Target	Represents the item an action does something to. The target is usually a control, such as a memo or a data control. Not all actions require a target. For example, the standard help actions ignore the target and simply launch the Help system.
Toolbar	Displays a visible row of button icons which, when clicked, cause the program to perform some action, such as printing the current document. You can create toolbars by using the cross-platform component *TToolBar*.

See to .

What is an action?

As you are developing your application, you can create a set of actions that you can use on various UI elements. You can organize them into categories that can be dropped onto a menu as a set (for example, Cut, Copy, and Paste) or one at a time (for example, Tools|Customize).

An action corresponds to one or more elements of the user interface, such as menu commands or toolbar buttons. Actions serve two functions: (1) they represent properties common to the user interface elements, such as whether a control is enabled or checked, and (2) they respond when a control fires, for example, when the application user clicks a button or chooses a menu item. You can create a repertoire of actions that are available to your application through menus, through buttons, through toolbars, context menus, and so on.

Actions are associated with other components:

- **Clients:** One or more clients use the action. The client most often represents a menu item or a button (for example, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton*, and so on). When the client receives a user command (such as a mouse click), it initiates an associated action. Typically, a client's *OnClick* event is associated with its action's *OnExecute* event.
- **Target:** The action acts on the target. The target is usually a control, such as a memo or a data control. Component writers can create actions specific to the needs of the controls they design and use, and then package those units to create more modular applications. Not all actions use a target. For example, the standard help actions ignore the target and simply launch the help system.

  A target can also be a component. For example, data controls change the target to an associated dataset.

The client influences the action—the action responds when a client fires the action. The action also influences the client—action properties dynamically update the client properties. For example, if at runtime an action is disabled (by setting its *Enabled* property to false), every client of that action is disabled, appearing grayed.

You can add, delete, and rearrange actions using the Action List editor (displayed by double-clicking an action list object, *TActionList*). These actions are later connected to client controls.

Using action lists

Action lists maintain a list of actions that your application can take in response to something a user does. By using action objects, you centralize the functions performed by your application from the user interface. This lets you share common code for performing actions (for example, when a toolbar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

Setting up action lists

Setting up action lists is fairly easy once you understand the basic steps involved:

- Create the action list.
- Add actions to the action list.
- Set properties on the actions.
- Attach clients to the action.

Here are the steps in more detail:

1. Drop a *TActionList* object onto your form or data module. (ActionList is on the Standard page of the Component palette.)
2. Double-click the *TActionList* object to display the Action List editor.
   a. Use one of the predefined actions listed in the editor: right-click and choose New Standard Action.
   b. The predefined actions are organized into categories (such as Dataset, Edit, Help, and Window) in the Standard Action Classes dialog box. Select all the standard actions you want to add to the action list and click OK.

      or
   c. Create a new action of your own: right-click and choose New Action.
3. Set the properties of each action in the Object Inspector. (The properties you set affect every client of the action.)

   The *Name* property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible,* and *Execute*) correspond to the properties and events of its client controls. The client's corresponding properties are typically, but not necessarily, the same name as the corresponding client property. For example, an action's *Enabled* property corresponds to a *TToolButton*'s *Enabled* property. However, an action's *Checked* property corresponds to a *TToolButton*'s *Down* property.
4. If you use the predefined actions, the action includes a standard response that occurs automatically. If creating your own action, you need to write an event handler that defines how the action responds when fired. See "What happens when an action fires" on page 9-24 for details.
5. Attach the actions in the action list to the clients that require them:
   - Click on the control (such as the button or menu item) on the form or data module. In the Object Inspector, the *Action* property lists the available actions.
   - Select the one you want.

The standard actions, such as *TEditDelete* or *TDataSetPost*, all perform the action you would expect. You can look at the online reference Help for details on how all of the standard actions work if you need to. If writing your own actions, you'll need to understand more about what happens when the action is fired.

What happens when an action fires

When an event fires, a series of events intended primarily for generic actions occurs. Then if the event doesn't handle the action, another sequence of events occurs.

Responding with events

When a client component or control is clicked or otherwise acted on, a series of events occurs to which you can respond. For example, the following code illustrates the event handler for an action that toggles the visibility of a toolbar when the action is executed:

Delphi example

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
   { Toggle Toolbar1's visibility }
   ToolBar1.Visible := not ToolBar1.Visible;
end;
```

C++ example

```
void __fastcall TForm1::Action1Execute(TObject *Sender)
{
   // Toggle Toolbar1's visibility
   ToolBar1->Visible = !ToolBar1->Visible;
}
```

Note For general information about events and event handlers, see "Working with events and event handlers" on page 5.

You can supply an event handler that responds at one of three different levels: the action, the action list, or the application. This is only a concern if you are using a new generic action rather than a predefined standard action. You do not have to worry about this if using the standard actions because standard actions have built-in behavior that executes when these events occur.

The order in which the event handlers will respond to events is as follows:

- Action list
- Application
- Action

When the user clicks on a client control, Kylix calls the action's Execute method which defers first to the action list, then the Application object, then the action itself if neither action list nor Application handles it. To explain this in more detail, Kylix follows this dispatching sequence when looking for a way to respond to the user action:

1 If you supply an *OnExecute* event handler for the action list and it handles the action, the application proceeds.

The action list's event handler has a parameter called *Handled*, that returns false by default. If the handler is assigned and it handles the event, it returns true, and the processing sequence ends here. For example:

D

Delphi example

```
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
   Handled := True;
end;
```

C++

C++ example

```
void __fastcall TForm1::ActionList1ExecuteAction(TBasicAction *Action, bool &Handled)
{
   Handled = true;
}
```

If you don't set *Handled* to true in the action list event handler, then processing continues.

2 If you did not write an *OnExecute* event handler for the action list or if the event handler doesn't handle the action, the application's *OnActionExecute* event handler fires. If it handles the action, the application proceeds.

The global *Application* object receives an *OnActionExecute* event if any action list in the application fails to handle an event. Like the action list's *OnExecute* event handler, the *OnActionExecute* handler has a parameter *Handled* that returns false by default. If an event handler is assigned and handles the event, it returns true, and the processing sequence ends here. For example:

D

Delphi example

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
  { Prevent execution of all actions in Application }
  Handled := True;
end;
```

C++

C++ example

```
void __fastcall TForm1::ApplicationExecuteAction(TBasicAction *Action, bool &Handled)
{
  // Prevent execution of all actions in Application
  Handled = true;
}
```

3 If the application's *OnExecute* event handler doesn't handle the action, the action's *OnExecute* event handler fires.

You can use built-in actions or create your own action classes that know how to operate on specific target classes (such as edit controls). When no event handler is found at any level, the application next tries to find a target on which to execute the action. When the application locates a target that the action knows how to address, it invokes the action. See the next section for details on how the application locates a target that can respond to a predefined action class.

How actions find their targets

"What happens when an action fires" on page 9-24 describes the execution cycle that occurs when a user invokes an action. If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

The application looks for the target using the following sequence:

1. Active control: The application looks first for an active control as a potential target.
2. Active form: If the application does not find an active control or if the active control can't act as a target, it looks at the screen's *ActiveForm*.
3. Controls on the form: If the active form is not an appropriate target, the application looks at the other controls on the active form for a target.

If no target is located, nothing happens when the event is fired.

Some controls can expand the search to defer the target to an associated component; for example, data-aware controls defer to the associated dataset component. Also, some predefined actions do not use a target; for example, the File Open dialog.

Updating actions

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is "checked" when the toolbar is visible:

Delphi example

```
procedure TForm1.Action1Update(Sender: TObject);
begin
   { Indicate whether ToolBar1 is currently visible }
   (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

C++ example

```
void __fastcall TForm1::Action1Update(TObject *Sender)
{
   // Indicate whether ToolBar1 is currently visible
   ((TAction *)Sender)->Checked = ToolBar1->Visible;
}
```

Warning Do not add time-intensive code to the *OnUpdate* event handler. This executes whenever the application is idle. If the event handler takes too much time, it will adversely affect performance of the entire application.

Predefined action classes

The Action List editor lets you use predefined action classes that automatically perform actions. The predefined actions fall into the following categories:

Table 9.2 Action classes

Class	Description
Edit	Used with an edit control target. *TEditAction* is the base class for descendants that each override the *ExecuteTarget* method to implement copy, cut, and paste tasks by using the clipboard.
Help	Used with any target. *THelpAction* is the base class for descendants that each override the *ExecuteTarget* method to pass the command onto a Help system.
Window	Used with forms as targets in an MDI application. *TWindowAction* is the base class for descendants that each override the *ExecuteTarget* method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.
DataSet	Used with a dataset component target. *TDataSetAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement navigation and editing of the target. *TDataSetAction* introduces a *DataSource* property that ensures actions are performed on that dataset. If *DataSource* is nil (Delphi) or NULL (C++), the currently focused data-aware control is used.

All of the action objects are described under the action object names in the online reference Help. Refer to the Help for details on how they work.

Writing action components

You can also create your own predefined action classes. When you write your own action classes, you can build in the ability to execute on certain target classes of object. Then, you can use your custom actions in the same way you use pre-defined action classes. That is, when the action can recognize and apply itself to a target class, you can simply assign the action to a client control, and it acts on the target with no need to write an event handler.

Component writers can use the classes in the QStdActns and DBActns units as examples for deriving their own action classes to implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction*, *TWindowAction*, and so on) generally override *HandlesTarget*, *UpdateTarget*, and other methods to limit the target for the action to a specific class of objects. The descendant classes typically override *ExecuteTarget* to perform a specialized task. These methods are described here:

Method	Description
HandlesTarget	Called automatically when the user invokes an object (such as a tool button or menu item) that is linked to the action. The *HandlesTarget* method lets the action object indicate whether it is appropriate to execute at this time with the object specified by the *Target* parameter as a target. See "How actions find their targets" on page 9-26 for details.
UpdateTarget	Called automatically when the application is idle so that actions can update themselves according to current conditions. Use in place of *OnUpdateAction*. See "Updating actions" on page 9-26 for details.
ExecuteTarget	Called automatically when the action fires in response to a user action in place of *OnExecute* (for example, when the user selects a menu item or presses a tool button that is linked to this action). See "What happens when an action fires" on page 9-24 for details.

Registering actions

When you write your own actions, you can register actions to enable them to appear in the Action List editor. You register and unregister actions by using the global routines in the Actnlist unit:

Delphi example

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);

procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

C++ example

```
extern PACKAGE void __fastcall RegisterActions(const AnsiString CategoryName, TMetaClass*
const * AClasses, const int AClasses_Size, TMetaClass* Resource);

extern PACKAGE void __fastcall UnRegisterActions(TMetaClass* const * AClasses, const int
AClasses_Size);
```

When you call *RegisterActions*, the actions you register appear in the Action List editor for use by your applications. You can supply a category name to organize your actions, as well as a *Resource* parameter that lets you supply default property values.

Delphi example

The following code registers the standard actions with the IDE:

```
{ Standard action registration }

RegisterActions('', [TAction], nil);

RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);

RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

C++ example

The following code registers actions with the IDE in the *MyAction* unit:

```
namespace MyAction
{
  void __fastcall PACKAGE Register()
  {
     // code goes here to register any components and editors
      TMetaClass classes[2] = {__classid(TMyAction1), __classid(TMyAction2)};
      RegisterActions("MySpecialActions", classes, 1, NULL);
  }
}
```

When you call *UnRegisterActions*, the actions no longer appear in the Action List editor.

Creating and managing menus

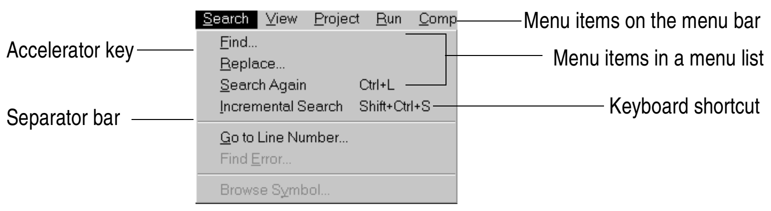
Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also change menus at runtime, to provide more information or options to the user.

This chapter explains how to use the Menu Designer to design menu bars and pop-up (local) menus. It discusses the following ways to work with menus at design time and runtime:

- Opening the Menu Designer.
- Building menus.
- Editing menu items in the Object Inspector.
- Using the Menu Designer context menu.
- Using menu templates.
- Saving a menu as a template.
- Adding images to menu items.

Figure 9.2 Menu terminology

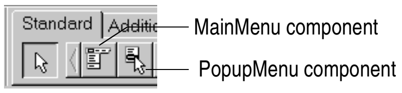


For information about hooking up menu items to the code that executes when they are selected, see "Associating menu events with event handlers" on page 6-9.

Opening the Menu Designer

You design menus for your application using the Menu Designer. Before you can start using the Menu Designer, first add either a *TMainMenu* or *TPopupMenu* component to your form. Both menu components are located on the Standard page of the Component palette.

Figure 9.3 MainMenu and PopupMenu components



A MainMenu component creates a menu that's attached to the form's title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar.

To open the Menu Designer, select a menu component on the form, and then either:

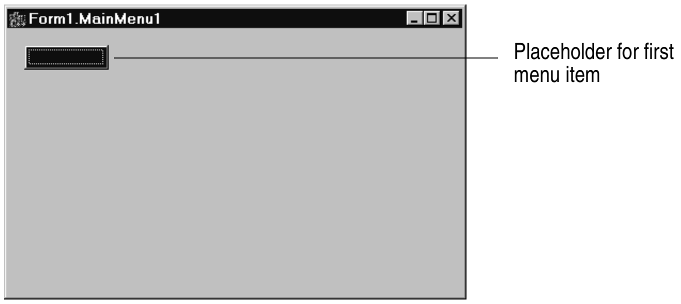
- Double-click the menu component.

  or

- From the Properties page of the Object Inspector, select the *Items* property, and then either double-click [Menu] in the Value column, or click the ellipsis (...) button.

  The Menu Designer appears, with the first (blank) menu item highlighted in the Designer, and the *Caption* property selected in the Object Inspector.

Figure 9.4 Menu Designer for a main menu



Building menus

You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates.

This section discusses the basics of creating a menu at design time. For more information about menu templates, see "Using menu templates" on page 9-38.

Naming menus

As with all components, when you add a menu component to the form, Kylix gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows language naming conventions.

Kylix adds the menu name to the form's type declaration, and the menu name then appears in the Component list.

Naming the menu items

In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways:

- Directly type the value for the *Name* property.
- Type the value for the *Caption* property first, and let Kylix derive the *Name* property from the caption.

  For example, if you give a menu item a *Caption* property value of *File*, Kylix assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Kylix leaves the *Caption* property blank until you type a value.

Note If you enter characters in the *Caption* property that are not valid for Delphi or C++ identifiers, Kylix modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Kylix precedes the number with a character to derive the *Name* property.

The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

Table 9.3 Sample captions and their derived names

Component caption	Derived name	Explanation
&File	File1	Removes ampersand
&File (2nd occurrence)	File2	Numerically orders duplicate items
1234	N12341	Adds a preceding letter and numerical order
1234 (2nd occurrence)	N12342	Adds a number to disambiguate the derived name
$@@@#	N1	Removes all non-standard characters, adding preceding letter and numerical order
- (hyphen)	N2	Numerical ordering of second occurrence of caption with no standard characters

As with the menu component, Kylix adds any menu item names to the form's type declaration, and those names then appear in the Component list.

Adding, inserting, and deleting menu items

The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.

To add menu items at design time,

1. Select the position where you want to create the menu item.

   If you've just opened the Menu Designer, the first position on the menu bar is already selected.

2. Begin typing to enter the caption. Or enter the *Name* property first by specifically placing your cursor in the Object Inspector and entering a value. In this case, you then need to reselect the *Caption* property and enter a value.

3. Press *Enter*.

   The next placeholder for a menu item is selected.

   If you entered the *Caption* property first, use the arrow keys to return to the menu item you just entered. You'll see that Kylix has filled in the *Name* property based on the value you entered for the caption. (See "Naming the menu items" on page 9-31.)

4. Continue entering values for the *Name* and *Caption* properties for each new item you want to create, or press *Esc* to return to the menu bar.

   Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press *Enter* to complete an action. To return to the menu bar, press *Esc*.

To insert a new, blank menu item,

1. Place the cursor on a menu item.
2. Press *Ins*.

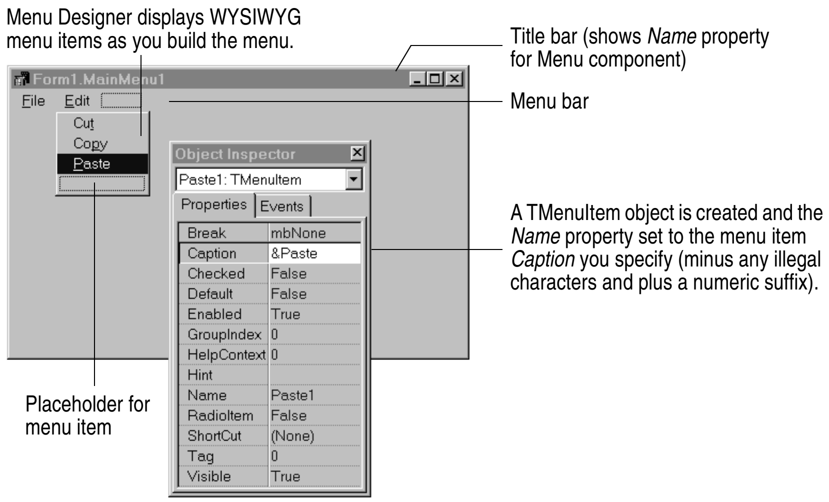
   Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

To delete a menu item or command,

1. Place the cursor on the menu item you want to delete.
2. Press *Del*.

Note You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at runtime.

Figure 9.5 Adding menu items to a main menu



Menu Designer displays WYSIWYG menu items as you build the menu.

Title bar (shows *Name* property for Menu component)

Menu bar

A TMenuItem object is created and the *Name* property set to the menu item *Caption* you specify (minus any illegal characters and plus a numeric suffix).

Placeholder for menu item

Break	mbNone
Caption	&Paste
Checked	False
Default	False
Enabled	True
GroupIndex	0
HelpContext	0
Hint	
Name	Paste1
RadioItem	False
ShortCut	(None)
Tag	0
Visible	True

Adding separator bars

Separator bars insert a line between menu items. You can use separator bars to indicate groupings within the menu list, or simply to provide a visual break in a list.

To make the menu item a separator bar, type a hyphen (-) for the caption.

Specifying accelerator keys and keyboard shortcuts

Accelerator keys enable the user to access a menu command from the keyboard by pressing *Alt*+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Kylix automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual***.**

To specify an accelerator,

- Add an ampersand in front of the appropriate letter.

  For example, to add a Save menu command with the *S* as an accelerator key, type `&Save`.

Keyboard shortcuts enable the user to perform the action without using the menu directly, by typing in the shortcut key combination.

To specify a keyboard shortcut,

- Use the Object Inspector to enter a value for the *ShortCut* property, or select a key combination from the drop-down list.

  This list is only a subset of the valid combinations you can type in.

When you add a shortcut, it appears next to the menu item caption.

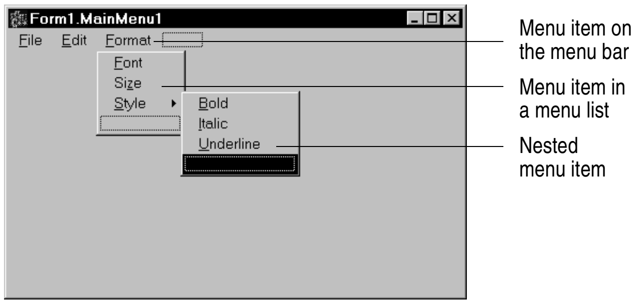
Caution Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

Creating submenus

Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Kylix supports as many levels of such submenus as you want to build into your menu.

Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)

Figure 9.6 Nested menu structures



To create a submenu,

1. Select the menu item under which you want to create a submenu.
2. Press *Ctrl→* to create the first placeholder, or right-click and choose Create Submenu.
3. Type a name for the submenu item, or drag an existing menu item into this placeholder.
4. Press *Enter*, or ↓, to create the next placeholder.
5. Repeat steps 3 and 4 for each item you want to create in the submenu.
6. Press *Esc* to return to the previous menu level.

Creating submenus by demoting existing menus

You can create a submenu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact submenu. This pertains to submenus as well. Moving a menu item into an existing submenu just creates one more level of nesting.

Moving menu items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a different menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a *different* menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar,

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.
2. Release the mouse button to drop the menu item at the new location.

To move a menu item into a menu list,

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

   This causes the menu to open, enabling you to drag the item to its new location.
2. Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

Adding images to menu items

Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. You can add single bitmaps to menu items, or you can organize images for your application into an image list and add them to a menu from the image list. If you're using several bitmaps of the same size in your application, it's useful to put them into an image list.

To add a single image to a menu or menu item, set its *Bitmap* property to reference the name of the bitmap to use on the menu or menu item.

To add an image to a menu item using an image list:

1. Drop a *TMainMenu* or *TPopupMenu* object on a form.
2. Drop a *TImageList* object on the form.
3. Open the ImageList editor by double clicking on the *TImageList* object.
4. Click Add to select the bitmap or bitmap group you want to use in the menu. Click OK.
5. Set the *TMainMenu* or *TPopupMenu* object's *Images* property to the ImageList you just created.
6. Create your menu items and submenu items as described previously.

7. Select the menu item you want to have an image in the Object Inspector and set the *ImageIndex* property to the corresponding number of the image in the *ImageList* (the default value for *ImageIndex* is -1, which doesn't display an image).

Note Use images that are 16 by 16 pixels for proper display in the menu. Although you can use other sizes for the menu images, alignment and consistency problems may result when using images greater than or smaller than 16 by 16 pixels.

Viewing the menu

You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)

To view the menu,

1. If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.
2. If the form has more than one menu, select the menu you want to view from the form's *Menu* property drop-down list.

   The menu appears in the form exactly as it will when you run the program.

Editing menu items in the Object Inspector

This section has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer.

The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the Object Inspector, just as you would for any component selected in the form.

When you edit a menu item by using the Menu Designer, its properties are still displayed in the Object Inspector. You can switch focus to the Object Inspector and continue editing the menu item properties there. Or you can select the menu item from the Component list in the Object Inspector and edit its properties without ever opening the Menu Designer.

To close the Menu Designer window and continue editing menu items,

1. Switch focus from the Menu Designer window to the Object Inspector by clicking the properties page of the Object Inspector.
2. Close the Menu Designer as you normally would.

   The focus remains in the Object Inspector, where you can continue editing properties for the selected menu item. To edit another menu item, select it from the Component list.

Using the Menu Designer context menu

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to "Using menu templates" on page 9-38.)

To display the context menu, right-click the Menu Designer window, or press *Alt+F10* when the cursor is in the Menu Designer window.

Commands on the context menu

The following table summarizes the commands on the Menu Designer context menu.

Table 9.4 Menu Designer context menu commands

Menu command	Action
Insert	Inserts a placeholder above or to the left of the cursor.
Delete	Deletes the selected menu item (and all its sub-items, if any).
Create Submenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item.
Select Menu	Opens a list of menus in the current form. Double-clicking a menu name opens the designer window for the menu.
Save As Template	Opens the Save Template dialog box, where you can save a menu for future reuse.
Insert From Template	Opens the Insert Template dialog box, where you can select a template to reuse.
Delete Templates	Opens the Delete Templates dialog box, where you can choose to delete any existing templates.
Insert From Resource	Opens the Insert Menu from Resource file dialog box, where you can choose an .rc or .mnu file to open in the current form.

Switching between menus at design time

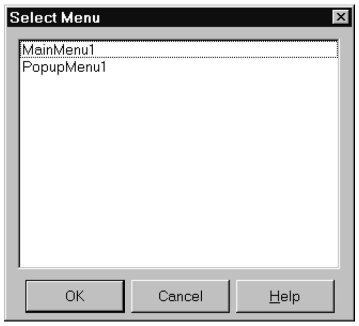
If you're designing several menus for your form, you can use the Menu Designer context menu or the Object Inspector to easily select and move among them.

To use the context menu to switch between menus in a form,

1 Right-click in the Menu Designer and choose Select Menu.

The Select Menu dialog box appears.

Figure 9.7 Select Menu dialog box



This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.

2 From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch between menus in a form,

1 Give focus to the form whose menus you want to choose from.
2 From the Component list, select the menu you want to edit.
3 On the Properties page of the Object Inspector, select the *Items* property for this menu, and then either click the ellipsis button, or double-click [Menu].

Using menu templates

Kylix provides several predesigned menus, or menu templates, that contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

The menu templates shipped with Kylix are stored in the .borland directory in the delphi69dmt (Delphi) and bcb69dmt (C++) filesin a default installation.

You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

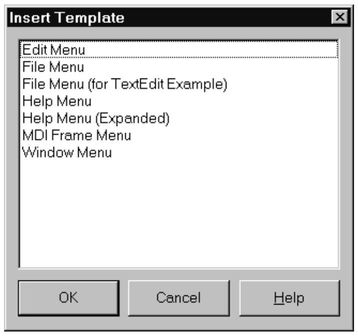
To add a menu template to your application,

1 Right-click the Menu Designer and choose Insert From Template.

(If there are no templates, the Insert From Template option appears dimmed in the context menu.)

The Insert Template dialog box opens, displaying a list of available menu templates.

Figure 9.8 Sample Insert Template dialog box for menus



2 Select the menu template you want to insert, then press *Enter* or choose OK.

This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

1 Right-click the Menu Designer and choose Delete Templates.

(If there are no templates, the Delete Templates option appears dimmed in the context menu.)

The Delete Templates dialog box opens, displaying a list of available templates.

2 Select the menu template you want to delete, and press *Del*.

Kylix deletes the template from the templates list and from your hard disk.

Saving a menu as a template

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in the .borland directory in the delphi69dmt (Delphi) and bcb69dmt (C++) files.

To save a menu as a template,

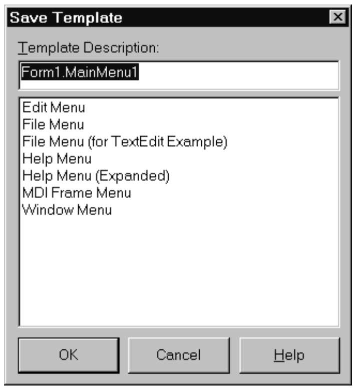
1 Design the menu you want to be able to reuse.

This menu can contain as many items, commands, and submenus as you like; everything in the active Menu Designer window will be saved as one reusable menu.

2 Right-click in the Menu Designer and choose Save As Template.

The Save Template dialog box appears.

Figure 9.9 Save Template dialog box for menus



3 In the Template Description edit box, type a brief description for this menu, and then choose OK.

The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

Note The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the *Name* or *Caption* property for the menu.

Naming conventions for template menu items and event handlers

When you save a menu as a template, Kylix does not save its *Name* property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Kylix then generates new names for it and all of its items.

For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Kylix names it *File1*. If you insert it into a menu with an existing menu item named *File1*, Kylix names it *File2*.

Kylix also does not save any *OnClick* event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, Kylix still generates the event handler name. You can easily associate items in the menu template with existing *OnClick* event handlers in the form.

For more information, see "Associating an event with an existing event handler" on page 6-7.

Manipulating menu items at runtime

Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can insert a menu item by using the menu item's *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the *Enabled* property.

For examples that use the menu item's *Visible* and *Enabled* properties, see "Disabling menu items" on page 7-10.

In multiple document interface (MDI) applications, you can also merge menu items into an existing menu bar. The following section discusses this in more detail.

Merging menus

For MDI applications, such as the text editor sample application, your application's main menu needs to be able to receive menu items either from another form. This is often called *merging menus*.

You prepare menus for merging by specifying values for two properties:

- *Menu*, a property of the form
- *GroupIndex*, a property of menu items in the menu

Specifying the active menu: Menu property

The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,

```
Form1.Menu := SecondMenu;
```

```
Form1->Menu = SecondMenu;
```

Determining the order of merged menu items: GroupIndex property

The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted.

The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:

- Lower numbers appear first (farther left) in the menu.

  For instance, set the *GroupIndex* property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.

- To replace items in the main menu, give items on the child menu the same *GroupIndex* value.

  This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a *GroupIndex* value of 1, you can replace it with one or more items from the child form's menu by giving them a *GroupIndex* value of 1 as well.

Giving multiple items in the child menu the same *GroupIndex* value keeps their order intact when they merge into the main menu.

- To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.

  For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

Designing toolbars

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on.

You can add a toolbar to a form in several ways:

- Place a panel (*TPanel*) on the form and add controls (typically speed buttons) to it.
- Use a toolbar component (*TToolBar*) instead of *TPanel*, and add controls to it. *TToolBar* manages buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. If you use tool button (*TToolButton*) controls on the toolbar, *TToolBar* makes it easy to group the buttons functionally and provides other display options.

How you implement your toolbar depends on your application. The advantage of using the Panel component is that you have total control over the look and feel of the toolbar.

By using the toolbar component, you are ensuring that your application has a consistent look and feel. If these operating system controls change in the future, your application could change as well.

The following sections describe how to:

- Add a toolbar and corresponding speed button controls using the panel component.
- Add a toolbar and corresponding tool button controls using the Toolbar component.
- Respond to clicks.
- Add hidden toolbars.
- Hide and show toolbars.

Adding a toolbar using a panel component

To add a toolbar to a form using the panel component, 1 Add a panel component to the form (from the Standard page of the Component palette).

2 Set the panel's *Align* property to *alTop*. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.

3 Add speed buttons or other controls to the panel.

Speed buttons are designed to work on toolbar panels. A speed button usually has no caption, only a small graphic (called a *glyph*), which represents the button's function.

Speed buttons have three possible modes of operation. They can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement speed buttons on toolbars, do the following:

- Add a speed button to a toolbar panel.
- Assign a speed button's glyph.
- Set the initial condition of a speed button.
- Create a group of speed buttons.
- Allow toggle buttons.

Adding a speed button to a panel

To add a speed button to a toolbar panel, place the speed button component (from the Additional page of the Component palette) on the panel.

The panel, rather than the form, "owns" the speed button, so moving or hiding the panel also moves or hides the speed button.

The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.

Assigning a speed button's glyph

Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer.

You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.

To assign a glyph to a speed button at design time,

1 Select the speed button.

2 In the Object Inspector, select the *Glyph* property.

3 Double-click the Value column beside *Glyph* to open the Picture Editor and select the desired bitmap.

Setting the initial condition of a speed button

Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users.

Table 9.5 lists some actions you can set to change a speed button's appearance:

Table 9.5 Setting speed buttons' appearance

To make a speed button:	Set the toolbar's:
Appear pressed	*GroupIndex* property to a value other than zero and its *Down* property to true.
Appear disabled	*Enabled* property to false.
Have a left margin	*Indent* property to a value greater than 0.

If your application has a default drawing tool, ensure that its button on the toolbar is pressed when the application starts. To do so, set its *GroupIndex* property to a value other than zero and its *Down* property to true.

Creating a group of speed buttons

A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up.

To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property.

The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.

Allowing toggle buttons

Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a *toggle*. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up.

To make a grouped speed button a toggle, set its *AllowAllUp* property to true.

Setting *AllowAllUp* to true for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group to act as a normal group, with only one button pressed at a time, but also allows every button to be up at the same time.

Adding a toolbar using the toolbar component

The toolbar component (*TToolBar*) offers button management and display features that panel components do not. To add a toolbar to a form using the toolbar component, 1 Add a toolbar component to the form (from the Common Controls page of the Component palette). The toolbar automatically aligns to the top of the form.

2 Add tool buttons or other controls to the bar.

Tool buttons are designed to work on toolbar components. Like speed buttons, tool buttons can:

- Act like regular pushbuttons.
- Toggle on and off when clicked.
- Act like a set of radio buttons.

To implement tool buttons on a toolbar, do the following:

- Add a tool button
- Assign images to tool buttons
- Set the tool buttons' appearance
- Create a group of tool buttons
- Allow toggled tool buttons

Adding a tool button

To add a tool button to a toolbar, right-click on the toolbar and choose New Button.

The toolbar "owns" the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the Component palette onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.

For example, the following code shows how to add tool buttons in a loop, adding them in the order 10-1 from left to right.

D

Delphi example

```
Toolbar1:= TToolbar.create(self);
  Toolbar1.Parent := Form1;
  Toolbar1.ShowCaptions := True;
  for i := 0 to 10 do
  begin
    Toolbutton := TToolbutton.Create(Self);
    {$IFDEF LINUX}
    ToolButton.Toolbar := Toolbar1;
    {$ENDIF}
    {$IFDEF WINDOWS}
    ToolButton.Parent := Toolbar1;
    {$ENDIF}
      ToolButton.Caption := '#' + inttostr(i);
    end;
  end;
```

C++ example

Assigning images to tool buttons

Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled. To assign images to tool buttons at design time,

1 Select the toolbar on which the buttons appear.

2 In the Object Inspector, assign a *TImageList* object to the toolbar's *Images* property. An image list is a collection of same-sized icons or bitmaps.

3 Select a tool button.

4 In the Object Inspector, assign an integer to the tool button's *ImageIndex* property that corresponds to the image in the image list that you want to assign to the button.

You can also specify separate images to appear on the tool buttons when they are disabled and when they are under the mouse pointer. To do so, assign separate image lists to the toolbar's *DisabledImages* and *HotImages* properties.

Setting tool button appearance and initial conditions

Table 9.6 lists some actions you can set to change a tool button's appearance:

Table 9.6 Setting tool buttons' appearance

To make a tool button:	Set the toolbar's:
Appear pressed	(on tool button) *Style* property to *tbsCheck* and *Down* property to true.
Appear disabled	*Enabled* property to false.
Have a left margin	*Indent* property to a value greater than 0.
Appear to have "pop-up" borders, thus making the toolbar appear transparent	*Flat* property to true.

To force a new row of controls after a specific tool button, Select the tool button that you want to appear last in the row and set its *Wrap* property to true.

To turn off the auto-wrap feature of the toolbar, set the toolbar's *Wrapable* property to false.

Creating groups of tool buttons

To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to true. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices.

Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to true forms a single group. To break up a group of tool buttons, separate the buttons with any of the following:

- A tool button whose *Grouped* property is false.
- A tool button whose *Style* property is not set to *tbsCheck*. To create spaces or dividers on the toolbar, add a tool button whose *Style* is *tbsSeparator* or *tbsDivider*.
- Another control besides a tool button.

Allowing toggled tool buttons

Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to true.

As with speed buttons, setting *AllowAllUp* to true for any tool button in a group automatically sets the same property value for all buttons in the group.

Responding to clicks

When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For more information, see “Working with events and event handlers” on page 6-5 and “Generating a handler for a component’s default event” on page 6-6.

Assigning a menu to a tool button

If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

1 Select the tool button.
2 In the Object Inspector, assign a pop-up menu (*TPopupMenu*) to the tool button’s *DropDownMenu* property.

If the menu’s *AutoPopup* property is set to true, it will appear automatically when the button is pressed.

Adding hidden toolbars

Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar:

1 Add a toolbar or panel component to the form.
2 Set the component’s *Visible* property to false.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

Hiding and showing toolbars

Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed.

To show or hide a toolbar at runtime, set its *Visible* property to true or false, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the user clicks that button, the application hides the corresponding toolbar.

You can also provide a means of toggling the toolbar. In the following example, a toolbar of pens is toggled from a button on the main toolbar. Since each click presses or releases the button, an *OnClick* event handler can show or hide the Pen toolbar depending on whether the button is up or down.

Delphi example

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
  PenBar.Visible := PenButton.Down;
end;
```

C++ example

```
void __fastcall TForm1::PenButtonClick(TObject *Sender)
{
  PenBar->Visible = PenButton->Down;
}
```

Sample programs

For examples of applications that use actions, action lists, menus, and toolbars, refer to ...\kylix\samples.

Chapter

10

Types of controls

Controls are visual components that help you design your user interface.

This chapter describes the different controls you can use, including text controls, input controls, buttons, list controls, grouping controls, display controls, grids, value list editors, and graphics controls.

To create a graphic control, see Chapter 44, "Creating a graphic control." To learn how to implement drag and drop in these controls, see Chapter 7, "Working with controls."

Text controls

Many applications use text controls to display text to the user. You can use:

- Edit controls, which allow the user to add text.
- Text viewing controls and labels, which do not allow user to add text.

Edit controls

Edit controls display text to the user and allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.

Use this component:	When you want users to do this:
TEdit	Edit a single line of text.
TMemo	Edit multiple lines of text.
TMaskEdit	Adhere to a particular format, such as a postal code or phone number.

TEdit and *TMaskEdit* are simple edit controls that include a single line text edit box in which you can type information. When the edit box has focus, a blinking insertion point appears.

You can include text in the edit box by assigning a string value to its *Text* property. You control the appearance of the text in the edit box by assigning values to its *Font* property. You can specify the typeface, size, color, and attributes of the font. The attributes affect all of the text in the edit box and cannot be applied to individual characters.

An edit box can be designed to change its size depending on the size of the font it contains. You do this by setting the *AutoSize* property to true. You can limit the number of characters an edit box can contain by assigning a value to the *MaxLength* property.

TMaskEdit is a special edit control that validates the text entered against a mask that encodes the valid forms the text can take. The mask can also format the text that is displayed to the user.

TMemo allow the user to add several lines of text.

Edit control properties

Following are some of the important properties of edit controls:

Table 10.1 Edit control properties

Property	Description
Text	Determines the text that appears in the edit box or memo control.
Font	Controls the attributes of text written in the edit box or memo control.
AutoSize	Enables the edit box to dynamically change its height depending on the currently selected font.
ReadOnly	Specifies whether the user is allowed to change the text.
MaxLength	Limits the number of characters in simple edit controls.
SelText	Contains the currently selected (highlighted) part of the text.
SelStart, *SelLength*	Indicate the position and length of the selected part of the text.

Memo controls

The *TMemo* control handles multiple lines of text.

TMemo is another type of edit box that handles multiple lines of text. The lines in a memo control can extend beyond the right boundary of the edit box, or they can wrap onto the next line. You control whether the lines wrap using the *WordWrap* property.

In addition to the properties that all edit controls have, memo controls include other properties, such as the following:

- *Alignment* specifies how text is aligned (left, right, or center) in the component.
- The *Text* property contains the text in the control. Your application can tell if the text changes by checking the *Modified* property.
- *Lines* contains the text as a list of strings.
- *WordWrap* determines whether the text will wrap at the right margin.

- *WantReturns* determines whether the user can insert hard returns in the text.
- *WantTabs* determines whether the user can insert tabs in the text.
- *AutoSelect* determines whether the text is automatically selected (highlighted) when the control becomes active.

At runtime, you can select all the text in the memo with the *SelectAll* method.

Text viewing controls

The text viewing controls display text but are read-only.

Use this component:	When you want users to do this:
TTextBrowser	Display a text file or simple HTML page that users can scroll through.
TTextViewer	Display a text file or simple HTML page. Users can scroll through the page or click links to view other pages and images.
TLCDNumber	Display numeric information in a digital display form.

TTextViewer acts as a simple viewer so that users can read and scroll through documents. With *TTextBrowser*, users can also click links to navigate to other documents and other parts of the same document. Documents visited are stored in a history list, which can be navigated using the *Backward*, *Forward*, and *Home* methods. *TTextViewer* and *TTextBrowser* are best used to display HTML-based text or to implement an HTML-based Help system.

TTextBrowser has the same properties as *TTextViewer* plus *Factory*. *Factory* determines the MIME factory object used to determine file types for embedded images. For example, you can associate filename extensions—such as .txt, .html, and .xml—with MIME types and have the factory load this data into the control.

Use the *FileName* property to add a text file, such as .html, to appear in the control at runtime.

Labels

Labels display text and are usually placed next to other controls.

Use this component:	When you want users to do this:
TLabel	Display text on a nonwindowed control.
TStaticText	Display text on a windowed control.

You place a label on a form when you need to identify or annotate another component such as an edit box or when you want to include text on a form. The standard label component, *TLabel*, is a widget-based control in CLX, so it cannot receive focus.

Label properties include the following:

- *Caption* contains the text string for the label.

- *Font*, *Color*, and other properties determine the appearance of the label. Each label can use only one typeface, size, and color.
- *FocusControl* links the label to another control on the form. If *Caption* includes an accelerator key, the control specified by *FocusControl* receives focus when the user presses the accelerator key.
- *ShowAccelChar* determines whether the label can display an underlined accelerator character. If *ShowAccelChar* is true, any character preceded by an ampersand (&) appears underlined and enables an accelerator key.
- *Transparent* determines whether items under the label (such as graphics) are visible.

Labels usually display read-only static text that cannot be changed by the application user. The application can change the text while it is executing by assigning a new value to the *Caption* property. To add a text object to a form that a user can scroll or edit, use *TEdit*.

Specialized input controls

The following components provide additional ways of capturing input.

Use this component:	When you want users to do this:
TScrollBar	Select values on a continuous range
TTrackBar	Select values on a continuous range (more visually effective than a scroll bar)
TSpinEdit	Select a value from a spinner widget

Scroll bars

The scroll bar component creates a scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar.

The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don't require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBox*.

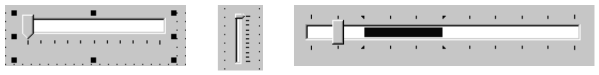
Track bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like color, volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar.

- Use the *Max* and *Min* properties to set the upper and lower range of the track bar.

- Use *SelEnd* and *SelStart* to highlight a selection range. See Figure 10.1.
- The *Orientation* property determines whether the track bar is vertical or horizontal.
- By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To control the intervals between ticks, use the *TickStyle* property and *SetTick* method.

Figure 10.1 Three views of the track bar component



- *Position* sets a default position for the track bar and tracks the position at runtime.
- By default, users can move one tick up or down by pressing the up and down arrow keys. Set *LineSize* to change that increment.
- Set *PageSize* to determine the number of ticks moved when the user presses *Page Up* and *Page Down*.

Spin edit controls

A spin edit control (*TSpinEdit*) is also called an up-down widget, little arrows widget, or spin button. This control lets the application user change an integer value in fixed increments, either by clicking the up or down arrow buttons to increase or decrease the value currently displayed, or by typing the value directly into the spin box.

The current value is given by the *Value* property; the increment, which defaults to 1, is specified by the *Increment* property.

Splitter controls

A splitter (*TSplitter*) placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane.

After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment* to *alLeft*, then place a splitter (also aligned to *alLeft*) to the right of the panel, and finally place another panel (aligned to *alLeft* or *alClient*) to the right of the splitter.

Set *MinSize* to specify a minimum size the splitter must leave when resizing its neighboring control. Set *Beveled* to true to give the splitter's edge a 3D look.

Buttons and similar controls

Aside from menus, buttons provide the most common way to initiate an action or command in an application. Kylix offers several button-like controls:

Use this component:	To do this:
TButton	Present command choices on buttons with text
TBitBtn	Present command choices on buttons with text and glyphs
TSpeedButton	Create grouped toolbar buttons
TCheckBox	Present on/off options
TRadioButton	Present a set of mutually exclusive choices
TToolBar	Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions

Action lists let you centralize responses to user commands (actions) for objects such as menus and buttons that respond to those commands. See "Using action lists" on page 6-14 for details on how to use action lists with buttons, toolbars, and menus.

CLX allows you to custom draw buttons individually or application wide. See Chapter 6, "Developing the application user interface."

Button controls

Users click button controls with the mouse to initiate actions. Buttons are labeled with text that represent the action. The text is specified by assigning a string value to the *Caption* property. Most buttons can also be selected by pressing a key on the keyboard as a keyboard shortcut. The shortcut is shown as an underlined letter on the button.

Users click button controls to initiate actions. You can assign an action to a *TButto*n component by creating an *OnClick* event handler for it. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor.

- Set *Cancel* to true if you want the button to trigger its *OnClick* event when the user presses Esc.
- Set *Default* to true if you want the Enter key to trigger the button's *OnClick* event.

Bitmap buttons

A bitmap button (*TBitBtn*) is a button control that presents a bitmap image on its face.

- To choose a bitmap for your button, set the *Glyph* property.
- Use *Kind* to automatically configure a button with a glyph and default behavior.
- By default, the glyph appears to the left of any text. To move it, use the *Layout* property.

- The glyph and text are automatically centered on the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button.
- By default, the image and the text are separated by 4 pixels. Use *Spacing* to increase or decrease the distance.
- Bitmap buttons can have 3 states: up, down, and held down. Set the *NumGlyphs* property to 3 to show a different bitmap for each state.

Speed buttons

Speed buttons (*TSpeedButton*), which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars.

- To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value.
- By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to true.
- If *AllowAllUp* is true, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to false if you want a group of buttons to act like a radio group.

For more information on speed buttons, refer to the section "Adding a toolbar using a panel component" on page 9-42 and "Organizing actions for toolbars and menus" on page 9-21.

Check boxes

A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank. You create check boxes using *TCheckBox*.

- Set *Checked* to true to make the box appear checked by default.
- Set *AllowGrayed* to true to give the check box three possible states: checked, unchecked, and grayed.
- The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*).

Note Check box controls display one of two binary states. The indeterminate state is used when other settings make it impossible to determine the current value for the check box.

Radio buttons

Radio buttons, also called option buttons, present a set of mutually exclusive choices. You can create individual radio buttons using *TRadioButton* or use the *radio group* component (*TRadioGroup*) to arrange radio buttons into groups automatically. You can group radio buttons to let the user select one from a limited set of choices. See "Grouping controls" on page 10-10 for more information.

A selected radio button is displayed as a circle filled in the middle. When not selected, the radio button shows an empty circle. Assign the value true or false to the Checked property to change the radio button's visual state.

Toolbars

Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the *TToolBar* component, then right-click and choose New Button to add buttons to the toolbar.

Using the *TToolBar* component has several advantages: buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and *TToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

You can use a centralized set of actions on toolbars and menus, by using *action lists*. See "Using action lists" on page 9-22 for details on how to use action lists with buttons and toolbars.

Toolbars can also parent other controls such as edit boxes, combo boxes, and so on.

List controls

Lists present the user with a collection of items to select from. Several components display lists:

Use this component:	To display:
TListBox	A list of text strings
TCheckListBox	A list with a check box in front of each item
TComboBox	An edit box with a scrollable drop-down list
TTreeView	A hierarchical list
TListView	A list of (draggable) items with optional icons, columns, and headings
TIconView	A list of items or data in rows and columns displayed as either small or large icons

Use the nonvisual *TStringList* and *TImageList* components to manage sets of strings and images. For more information about string lists, see "Working with string lists" on page 5-15.

List boxes and check-list boxes

List boxes (*TListBox*) and check-list boxes display lists from which users can select one or more choices from a list of possible options. The choices are represented using text, graphics, or both.

- *Items* uses a *TStrings* object to fill the control with values.
- *ItemIndex* indicates which item in the list is selected.

- *MultiSelect* specifies whether a user can select more than one item at a time.
- *Sorted* determines whether the list is arranged alphabetically.
- *Columns* specifies the number of columns in the list control.
- *ItemHeight* specifies the height of each item in pixels. The *Style* property can cause *ItemHeight* to be ignored.
- The *Style* property determines how a list control displays its items. By default, items are displayed as strings. By changing the value of *Style*, you can create *owner-draw* list boxes that display items graphically or in varying heights. For information on owner-draw controls, see "Adding graphics to controls" on page 7-12.

To create a simple list box,

1. Within your project, drop a list box component from the Component palette onto a form.
2. Size the list box and set its alignment as needed.
3. Double-click the right side of the *Items* property or choose the ellipsis button to display the String List Editor.
4. Use the editor to enter free form text arranged in lines for the contents of the list box.
5. Then choose OK.

To let users select multiple items in the list box, you can use the *ExtendedSelect* and *MultiSelect* properties.

Combo boxes

A combo box (*TComboBox*) combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes. If *AutoComplete* is enabled, the application looks for and displays the closest match in the list as the user types the data.

Three types of combo boxes are: standard, drop-down (the default), and drop-down list.

- Use the *Style* property to select the type of combo box you need.
- Use *csDropDown* to create an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list). Set the *DropDownCount* property to change the number of items displayed in the list.
- Use *csOwnerDrawFixed* or *csOwnerDrawVariable* to create *owner-draw* combo boxes that display items graphically or in varying heights. For information on owner-draw controls, see "Adding graphics to controls" on page 7-12.

In CLXcombo boxes , you can add an item to a drop-down list by entering text and pressing Enter in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to ciNone. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

Tree views

A tree view *(TTreeView)* displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. You can include icons with items' text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as check boxes, that reflect state information about the items.

- *Indent* sets the number of pixels horizontally separating items from their parents.
- *ShowButtons* enables the display of '+' and '–' buttons to indicate whether an item can be expanded.

To add items to a tree view control at design time, double-click on the control to display the TreeView Items editor. The items you add become the value of the *Items* property. You can change the items at runtime by using the methods of the *Items* property, which is an object of type *TTreeNodes*. *TTreeNodes* has methods for adding, deleting, and navigating the items in the tree view.

Tree views can display columns and subitems similar to list views in vsReport mode.

List views

List views, created using *TListView*, display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want:

- *vsList* displays items as labeled icons that cannot be dragged.
- *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

Icon views

The icon view, created using *TIconView*, displays a list of items or data in rows and columns as either small or large icons.

Grouping controls

A graphical interface is easier to use when related controls and information are presented in groups. Kylix provides several components for grouping components:

Use this component:	When you want this:
TGroupBox	A standard group box with a title
TRadioGroup	A simple group of radio buttons
TPanel	A more visually flexible group of controls
TScrollBox	A scrollable region containing controls
TTabControl	A set of mutually exclusive notebook-style tabs

Use this component:	When you want this:
TPageControl	A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls
THeaderControl	Resizable column headers

Group boxes and radio groups

A group box (*TGroupBox*) arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the Component palette and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component (*TRadioGroup*) simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the Object Inspector; each string in *Items* makes a radio button appear in the group box with the string as its caption. The value of the *ItemIndex* property determines which radio button is currently selected. Display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property. To respace the buttons, resize the radio group component.

Panels

The *TPanel* component provides a generic container for other controls. Panels are typically used to visually group components together on a form. Panels can be aligned with the form to maintain the same relative position when the form is resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel.

You can also place other controls onto a panel and use the *Align* property to ensure proper positioning of all the controls in the group on the form. You can make a panel alTop aligned so that its position will remain in place even if the form is resized.

The look of the panel can be changed to a raised or lowered look by using the *BevelOuter* and *BevelInner* properties. You can vary the values of these properties to create different visual 3-D effects. Note that if you merely want a raised or lowered bevel, you can use the less resource intensive *TBevel* control instead.

You can also use one or more panels to build various status bars or information display areas.

Scroll boxes

Scroll boxes (*TScrollBox*) create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents.

Another use of scroll boxes is to create multiple scrolling areas (views) in a window. Views are common in commercial word-processor, spreadsheet, and project management applications. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form.

Like panels and group boxes, scroll boxes contain other controls, such as *TButton* and *TCheckBox* objects. But a scroll box is normally invisible. If the controls in the scroll box cannot fit in its visible area, the scroll box automatically displays scroll bars.

Another use of a scroll box is to restrict scrolling in areas of a window, such as a toolbar or status bar (*TPanel* components). To prevent a toolbar and status bar from scrolling, hide the scroll bars, and then position a scroll box in the client area of the window between the toolbar and status bar. The scroll bars associated with the scroll box will appear to belong to the window, but will scroll only the area inside the scroll box.

Tab controls

The tab control component (*TTabControl*) creates a set of tabs that look like notebook dividers. You can create tabs by editing the *Tabs* property in the Object Inspector; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

Page controls

The page control component (*TPageControl*) is a page set suitable for multipage dialog boxes. A page control displays multiple overlapping pages that are *TTabSheet* objects. A page is selected in the user interface by clicking a tab on top of the control.

To create a new page in a page control at design time, right-click the control and choose New Page. At runtime, you add new pages by creating the object for the page and setting its *PageControl* property:

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

```
TTabSheet *pTabSheet = new TTabSheet(PageControl1);
pTabSheet->PageControl = PageControl1;
```

To access the active page, use the *ActivePage* property. To change the active page, you can set either the *ActivePage* or the *ActivePageIndex* property.

Header controls

A header control (*THeaderControl*) is a is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers. You can place the header sections above columns or fields. For example, header sections might be placed over a list box (*TListBox*).

Display controls

There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime to identify the object.

Use this component or property:	To do this:
TStatusBar	Display a status region (usually at the bottom of a window)
TProgressBar	Show the amount of work completed for a particular task
Hint and *ShowHint*	Activate fly-by or "tooltip" Help
HelpContext and *HelpFile*	Link context-sensitive online Help

Status bars

Although you can use a panel to make a status bar, it is simpler to use the *TStatusBar* component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

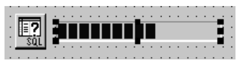
If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to true and use the *SimpleText* property to control the text displayed in the status bar.

You can also divide a status bar into several text areas, called panels. To create panels, edit the *Panels* property in the Object Inspector, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. Each panel's *Text* property contains the text displayed in the panel.

Progress bars

When your application performs a time-consuming operation, you can use a progress bar to show how much of the task is completed. A progress bar (*TProgressBar*) displays a dotted line that grows from left to right.

Figure 10.2 A progress bar



The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.

Help and hint properties

Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control.

The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to true; setting *ParentShowHint* to true causes the control's *ShowHint* property to have the same value as its parent's.

Grids

Grids display information in rows and columns. If you're writing a database application, use the *TDBGrid* or *TDBCtrlGrid* component described in "Using data controls." Otherwise, use a standard draw grid or string grid.

Draw grids

A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid.

- The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells.
- The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in the grid.
- You can change the width or height of a column or row with the *ColWidths* and *RowHeights* properties. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.
- You can choose to have fixed or non-scrolling columns and rows with the *FixedCols* and *FixedRows* properties. Assign a color to the fixed columns and rows with the *FixedColor* property.
- The *Options*, *DefaultColWidth*, and *DefaultRowHeight* properties also affect the appearance and behavior of the grid.

String grids

The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.

Graphic controls

The following components make it easy to incorporate graphics into an application.

Use this component:	To display:
TImage	Graphics files
TShape	Geometric shapes
TBevel	3-D lines and frames
TPaintBox	Graphics drawn by your program at runtime
TAnimate	GIF files

Notice that these include common paint routines (*Repaint*, *Invalidate*, and so on) that never need to receive focus.

Images

The image component (*TImage*) displays a graphical image, like a bitmap, icon, or drawing. The *Picture* property determines the graphic to be displayed. Use *Center*, *AutoSize*, *Stretch*, and *Transparent* to set display options. For more information, see "Overview of graphics programming" on page 11-1.

Shapes

The shape component displays a geometric shape. It is a widget-based control in CLX and therefore, cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape's color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.

Bevels

The bevel component (*TBevel*) is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.

Paint boxes

The paint box component (*TPaintBox*) allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see "Overview of graphics programming" on page 11-1.

Animation control

The animation component is a window that silently displays aGIF clip. An clip is a series of GIF frames, like a movie. Although the clips can have sound, animation controls work only with silent clips.

Following are some of the properties of an animation component:

- Set *AutoSize* to true to have the animation control adjust its size to the size of the frames in the AVI clip.
- *StartFrame* and *StopFrame* specify in which frames to start and stop the clip.
- Specify when to start and interrupt the animation by setting the *Active* property to true and false, respectively, and how many repetitions to play by setting the *Repetitions* property.

Chapter

11

Working with graphics

Graphics and multimedia elements can add polish to your applications. You can introduce these features into your application in a variety of ways. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime. To add multimedia capabilities, special components can play video clips.

Overview of graphics programming

CLX graphics components defined in the QGraphics unit encapsulate the Qt graphics widgets for adding graphics to cross-platform applications.

To draw graphics in an application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it manages the device context for you, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or drawings. Canvases are available only at runtime, so you do all your work with canvases by writing code.

TCanvas is a wrapper resource manager around a Qt painter. The *Handle* property of the canvas is a typed pointer to an instance of a Qt painter object. Having this instance pointer exposed allows you to use low-level Qt graphics library functions that require an instance pointer to a painter object QPainterH.

How graphic images appear in your application depends on the type of object whose canvas you draw on. If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its *OnPaint* event.

Using *TBitmap* for a basic command line program only works if XWindows is running. You will see the message “ProjectName: cannot connect to X server.”

When working with graphics, you often encounter the terms *drawing* and *painting*:

- Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.
- Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to *OnPaint* events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

The examples in the beginning of this chapter demonstrate how to draw various graphics, but they do so in response to *OnPaint* events. Later sections show how to do the same kind of drawing in response to other events.

Refreshing the screen

At certain times, the operating system determines that objects onscreen need to refresh their appearance. This is the case when a form or control is temporarily obscured, such as during window dragging, the form or control must repaint the obscured area when it is reposed.When this occurs,a paint event is generated, which CLX routes to *OnPaint* events. If you have written an *OnPaint* event handler for that object, it is called when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form’s *OnResize* event handler to redisplay any graphics.

If you use the *TImage* control to display a graphical image on a form, the painting and refreshing of the graphic contained in the *TImage* is handled automatically. The *Picture* property specifies the actual bitmap, drawing, or other graphic object that *TImage* displays. Drawing on a *TImage* creates a persistent image. Consequently, you do not need to do anything to redraw the contained image. In contrast, *TPaintBox*’s canvas maps directly onto the painter, so that anything drawn to the *PaintBox*’s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a *TPaintBox* in its constructor, you will need to add that code to your *OnPaint* event handler in order for the image to be repainted each time the client area is invalidated.

Types of graphic objects

CLX provides the graphic objects shown in Table 11.1. These objects have methods to draw on the canvas, which are described in “Using Canvas methods to draw graphic objects" on page 11-10 and to load and save to graphics files, as described in "Loading and saving graphics files" on page 11-25.

Table 11.1 Graphic object types

Object	Description
Picture	Used to hold any graphic image. To add additional graphic file formats, use the Picture *Register* method. Use this to handle arbitrary files such as displaying images in an image control.
Bitmap	A powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. Creating copies of a bitmap is fast since the *handle* is copied, not the image.
Clipboard	Represents the container for any text or graphics that are cut, copied, or pasted from or to an application. With the clipboard, you can get and retrieve data according to the appropriate format; handle reference counting, and opening and closing the clipboard; manage and manipulate formats for objects in the clipboard.
Icon	Represents the value loaded from an icon file.
Drawing	Contains a file that records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Drawings are scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, drawings do not display as fast as bitmaps. Use a drawing when versatility or precision is more important than performance.

Common properties and methods of Canvas

Table 11.2 lists the commonly used properties of the Canvas object. For a complete list of properties and methods, see the *TCanvas* component in online Help.

Table 11.2 Common properties of the Canvas object

Properties	Descriptions
Font	Specifies the font to use when writing text on the image. Set the properties of the TFont object to specify the font face, color, size, and style of the font.
Brush	Determines the color and pattern the canvas uses for filling graphical shapes and backgrounds. Set the properties of the TBrush object to specify the color and pattern or bitmap to use when filling in spaces on the canvas.
Pen	Specifies the kind of pen the canvas uses for drawing lines and outlining shapes. Set the properties of the TPen object to specify the color, style, width, and mode of the pen.
PenPos	Specifies the current drawing position of the pen.
Pixels	Specifies the color of the area of pixels within the current ClipRect.

These properties are described in more detail in "Using the properties of the Canvas object" on page 11-4.

Table 11.3 is a list of several methods you can use:

Table 11.3 Common methods of the Canvas object

Method	Descriptions
Arc	Draws an arc on the image along the perimeter of the ellipse bounded by the specified rectangle.
Chord	Draws a closed figure represented by the intersection of a line and an ellipse.
CopyRect	Copies part of an image from another canvas into the canvas.
Draw	Renders the graphic object specified by the Graphic parameter on the canvas at the location given by the coordinates (X, Y).
Ellipse	Draws the ellipse defined by a bounding rectangle on the canvas.
FillRect	Fills the specified rectangle on the canvas using the current brush.
FrameRect	Draws a rectangle using the Brush of the canvas to draw the border.
LineTo	Draws a line on the canvas from PenPos to the point specified by X and Y, and sets the pen position to (X, Y).
MoveTo	Changes the current drawing position to the point (X,Y).
Pie	Draws a pie-shaped the section of the ellipse bounded by the rectangle (X1, Y1) and (X2, Y2) on the canvas.
Polygon	Draws a series of lines on the canvas connecting the points passed in and closing the shape by drawing a line from the last point to the first point.
Polyline	Draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in Points.
Rectangle	Draws a rectangle on the canvas with its upper left corner at the point (X1, Y1) and its lower right corner at the point (X2, Y2). Use *Rectangle* to draw a box using Pen and fill it using Brush.
RoundRect	Draws a rectangle with rounded corners on the canvas.
StretchDraw	Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit.
TextHeight, TextWidth	Returns the height and width, respectively, of a string in the current font. Height includes leading between lines.
TextOut	Writes a string on the canvas, starting at the point (X,Y), and then updates the PenPos to the end of the string.
TextRect	Writes a string inside a region; any portions of the string that fall outside the region do not appear.

These methods are described in more detail in "Using Canvas methods to draw graphic objects" on page 11-10.

Using the properties of the Canvas object

With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image.

This section describes:

- Using pens.

• Using brushes.

Using pens

The *Pen* property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points.

The pen itself has four properties you can change:

- *Color* property changes the pen color.
- *Width* property changes the pen width.
- *Style* property changes the pen style.
- *Mode* property changes the pen mode.

The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style, and a mode called copy that overwrites anything already on the canvas.

You can use *TPenRecall* for quick saving off and restoring the properties of pens.

Changing the pen color

You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen.

To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which is drawn in the gaps between line segments. Background color comes from the Brush color property.

Since the user chooses a new color by clicking the grid, this code changes the pen's color in response to the *OnClick* event:

Delphi example

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

C++ example

```
void __fastcall TForm1::PenColorClick(TObject *Sender)
{
  Canvas->Pen->Color = PenColor->ForegroundColor;
}
```

Changing the pen width

A pen's width determines the thickness, in pixels, of the lines it draws.

Note If you are developing a cross-platform application to deploy on Windows 95/98 and the thickness is greater than 1, Windows always draw solid lines, regardless of the value of the pen's *Style* property.

To change the pen width, assign a numeric value to the pen's *Width* property.

Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width every time the position changes.

This is how to handle the scroll bar's *OnChange* event:

Delphi example

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
  Canvas.Pen.Width := PenWidth.Position;{ set the pen width directly }
 PenSize.Caption := IntToStr(PenWidth.Position);{ convert to string for caption }
end;
```

C++ example

```
void __fastcall TForm1::PenWidthChange(TObject *Sender)
{
  Canvas->Pen->Width = PenWidth->Position;         // set the pen width directly
  PenSize->Caption = IntToStr(PenWidth->Position); // convert to string
}
```

Changing the pen style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

Note For cross-platform applications deployed under Windows, Windows does not support dashed or dotted line styles for pens wider than one pixel and makes all larger pens solid, no matter what style you specify.

The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

To create one click-event handler for six pen-style buttons on a pen's toolbar, do the following:

1. Select all six pen-style buttons and select the Object Inspector | Events | *OnClick* event and in the Handler column, type `SetPenStyle`.

   The IDE generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.

2. Fill in the click-event handler by setting the pen's style depending on the value of *Sender*, which is the control that sent the click event:

Delphi example

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
   else if Sender = DashPen then Style := psDash
   else if Sender = DotPen then Style := psDot
   else if Sender = DashDotPen then Style := psDashDot
   else if Sender = DashDotDotPen then Style := psDashDotDot
   else if Sender = ClearPen then Style := psClear;
  end;
end;
```

C++ example

```
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender == SolidPen)
    Canvas->Pen->Style = psSolid;
  else if (Sender == DashPen)
    Canvas->Pen->Style = psDash;
  else if (Sender == DotPen)
    Canvas->Pen->Style = psDot;
  else if (Sender == DashDotPen)
    Canvas->Pen->Style = psDashDot;
  else if (Sender == DashDotDotPen)
    Canvas->Pen->Style = psDashDotDot;
'  else if (Sender == ClearPen)
    Canvas->Pen->Style = psClear;
}
```

The above event handler code could be further reduced by putting the pen style constants into the *Tag* properties of the pen style buttons. Then this event code would be something like:

```
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Pen->Style = (TPenStyle) ((TSpeedButton *)Sender)->Tag;
}
```

Changing the pen mode

A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on. See *TPen* in online Help for details.

Getting the pen position

The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its *PenPos* property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need.

To set the pen position, call the *MoveTo* method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:

```
Canvas.MoveTo(0, 0);
```

```
Canvas->MoveTo(0, 0);
```

Note Drawing a line with the *LineTo* method also moves the current position to the endpoint of the line.

Using brushes

The *Brush* property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way.

The brush has three properties you can manipulate:

- *Color* property changes the fill color.
- *Style* property changes the brush style.
- *Bitmap* property uses a bitmap as a brush pattern.

The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap.

You can use *TBrushRecall* for quick saving off and restoring the properties of brushes.

Changing the brush color

A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property.

You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar (see "Changing the pen color" on page 11-5):

Delphi example

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
  Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

C++ example

```
void __fastcall TForm1::BrushColorClick(TObject *Sender)
{
  Canvas->Brush->Color = BrushColor->BackgroundColor;
}
```

Changing the brush style

A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns.

To change the style of a brush, set its *Style* property to one of the predefined values: *bsBDiagonal*, *bsClear*, *bsCross*, *bsDense1 through bsDense7, bsDiagCross, bsFDiagonal*, *bsHorizontal*, *bsSolid*, or *bsVertical*.

This example sets brush styles by sharing a click-event handler for a set of eight of the brush-style buttons. All eight buttons are selected, the Object Inspector | Events | *OnClick* is set, and the *OnClick* handler is named *SetBrushStyle*. Here is the handler code:

D

Delphi example

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
   else if Sender = HorizontalBrush then Style := bsHorizontal
   else if Sender = VerticalBrush then Style := bsVertical
   else if Sender = FDiagonalBrush then Style := bsFDiagonal
   else if Sender = BDiagonalBrush then Style := bsBDiagonal
   else if Sender = CrossBrush then Style := bsCross
   else if Sender = DiagCrossBrush then Style := bsDiagCross;
  end;
end;
```

C++ example

```
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
  if (Sender == SolidBrush)
    Canvas->Brush->Style = bsSolid;
  else if (Sender == ClearBrush)
    Canvas->Brush->Style = bsClear;
  else if (Sender == HorizontalBrush)
    Canvas->Brush->Style = bsHorizontal;
  else if (Sender == VerticalBrush)
    Canvas->Brush->Style = bsVertical;
  else if (Sender == FDiagonalBrush)
    Canvas->Brush->Style = bsFDiagonal;
  else if (Sender == BDiagonalBrush)
    Canvas->Brush->Style = bsBDiagonal;
  else if (Sender == CrossBrush)
    Canvas->Brush->Style = bsCross;
  else if (Sender == DiagCrossBrush)
    Canvas->Brush->Style = bsDiagCross;
}
```

The above event handler code could be further reduced by putting the brush style constants into the *Tag* properties of the brush style buttons. Then this event code would be something like:

```
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
   if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Brush->Style = (TBrushStyle) ((TSpeedButton *)Sender)->Tag;
}
```

Setting the Brush Bitmap property

A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.

The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:

Delphi example

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;
```

C++ example

```
BrushBmp->LoadFromFile("MyBitmap.bmp");
  Form1->Canvas->Brush->Bitmap = BrushBmp;
  Form1->Canvas->FillRect(Rect(0,0,100,100));
```

Note The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remains valid for the lifetime of the Brush, and you must free the Bitmap object yourself afterwards.

Using Canvas methods to draw graphic objects

This section shows how to use some common methods to draw graphic objects. It covers:

- Drawing lines and polylines.
- Drawing shapes.
- Drawing rounded rectangles.

• Drawing polygons.

Drawing lines and polylines

A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.

Drawing lines

To draw a straight line on a canvas, use the *LineTo* method of the canvas.

LineTo draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.

For example, the following method draws crossed diagonal lines across a form whenever the form is painted:

D

Delphi example

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
 begin
   MoveTo(0, 0);
   LineTo(ClientWidth, ClientHeight);
   MoveTo(0, ClientHeight);
   LineTo(ClientWidth, 0);
 end;
end;
```

C++ example

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->MoveTo(0,0);
  Canvas->LineTo(ClientWidth, ClientHeight);
  Canvas->MoveTo(0, ClientHeight);
  Canvas->LineTo(ClientWidth, 0);
}
```

Drawing polylines

In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments.

To draw a polyline on a canvas, call the *Polyline* method of the canvas.

The parameter passed to the *Polyline* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the *MoveTo* method and the *LineTo* method because it eliminates a lot of call overhead.

The following method, for example, draws a rhombus in a form:

Delphi example

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
   Polyline([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
end;
```

This example takes advantage of Delphi's ability to create an open-array parameter on-the-fly. You can pass any array of points, but an easy way to construct an array quickly is to put its elements in brackets and pass the whole thing as a parameter.

C++ example

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  TPoint vertices[5];
  vertices[0] = Point(0, 0);
  vertices[1] = Point(50, 0);
  vertices[2] = Point(75, 50);
  vertices[3] = Point(25, 50);
  vertices[4] = Point(0, 0);
  Canvas->Polyline(vertices, 4);
}
```

Note that the last parameter to *Polyline* is the index of the last point, not the number of points.

For more information, see online Help.

Drawing shapes

Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object.

This section covers:

- Drawing rectangles and ellipses.
- Drawing rounded rectangles.
- Drawing polygons.

Drawing rectangles and ellipses

To draw a rectangle or ellipse on a canvas, call the canvas's *Rectangle* method or *Ellipse* method, passing the coordinates of a bounding rectangle.

The *Rectangle* method draws the bounding rectangle; *Ellipse* draws an ellipse that touches all sides of the rectangle.

The following method draws a rectangle filling a form's upper left quadrant, then draws an ellipse in the same area:

Delphi example

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
 Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

C++ example

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->Rectangle(0, 0, ClientWidth/2, ClientHeight/2);
  Canvas->Ellipse(0, 0, ClientWidth/2, ClientHeight/2);
}
```

Drawing rounded rectangles

To draw a rounded rectangle on a canvas, call the canvas's *RoundRect* method.

The first four parameters passed to *RoundRect* are a bounding rectangle, just as for the *Rectangle* method or the *Ellipse* method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.

The following method, for example, draws a rounded rectangle in a form's upper left quadrant, rounding the corners as sections of a circle with a diameter of 10 pixels:

Delphi example

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

C++ example

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->RoundRect(0, 0, ClientWidth/2, ClientHeight/2, 10, 10);
}
```

Drawing polygons

To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas.

Polygon takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.

For example, the following code draws a right triangle in the lower left half of a form:

Delphi example

```
procedure TForm1.FormPaint(Sender: TObject);
```

```
begin
 Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
   Point(ClientWidth, ClientHeight)]);
end;
```

C++ example

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  TPoint vertices[3];
  vertices[0] = Point(0, 0);
  vertices[1] = Point(0, ClientHeight);
  vertices[2] = Point(ClientWidth,ClientHeight);
  Canvas->Polygon(vertices,2);
}
```

Handling multiple drawing objects in your application

Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to:

- Keep track of which drawing tool to use.
- Change the tool with speed buttons.
- Use drawing tools.

Keeping track of which drawing tool to use

A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time.

D

In Delphi, you could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won't be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Delphi provides a means to handle both of these shortcomings. You can declare an enumerated type.

In Delphi, an enumerated type is really just a shorthand way of assigning sequential values to constants. Since it's also a type declaration, you can use Delphi's type-checking to ensure that you assign only those specific values.

To declare an enumerated type in Delphi, use the reserved work type, followed by an identifier for the type, then an equal sign, and the identifiers for the values in the type in parentheses, separated by commas.

In C++, you would typically use the enumerated type to list the available tools. Since an enumerated type is also a type declaration, you can use C++'s type-checking to ensure that you assign *only* those specific values.

The following code declares an enumerated type for each drawing tool available in a graphics application:

D

```
type
```

```
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

```
typedef enum {dtLine, dtRectangle, dtEllipse, dtRoundRect} TDrawingTool;
```

In C++, a variable of type *TDrawingTool* can be assigned only one of the constants *dtLine, dtRectangle, dtEllipse,* or *dtRoundRect*.

By convention, type identifiers begin with the letter *T*, and groups of similar constants (such as those making up an enumerated type) begin with a 2-letter prefix (such as *dt* for "drawing tool").

Delphi example

The declaration of the TDrawingTool type is equivalent to declaring a group of constants:

```
const
  dtLine = 0;
  dtRectangle = 1;
  dtEllipse = 2;
  dtRoundRect = 3;
```

In Delphi, the main difference is that by declaring the enumerated type, you give the constants not just a value, but also a type, which enables you to use Delphi's type-checking to prevent many errors. A variable of type TDrawingTool can be assigned only one of the constants dtLine..dtRoundRect. Attempting to assign some other number (even one in the range 0..3) generates a compile-time error.

In the following code, a field added to a form keeps track of the form's drawing tool:

Delphi example

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
 TForm1 = class(TForm)
    ...{ method declarations }
 public
    Drawing: Boolean;
   Origin, MovePt: TPoint;
   DrawingTool: TDrawingTool;{ field to hold current tool }
  end;
```

C++ example

```
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};

class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X,
  int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
private:// User declarations
public:// User declarations
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  //field to track whether button was pressed
  TPoint Origin, MovePt;  // fields to store points
  TDrawingTool DrawingTool;  // field to hold current tool
};
```

Changing the tool with speed buttons

Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:

Delphi example

```
procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
  DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
  DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
  DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
  DrawingTool := dtRoundRect;
end;
```

C++ example

```
void __fastcall TForm1::LineButtonClick(TObject *Sender)  // LineButton
{
  DrawingTool = dtLine;
}

void __fastcall TForm1::RectangleButtonClick(TObject *Sender) // RectangleButton
{
  DrawingTool = dtRectangle;
}

void __fastcall TForm1::EllipseButtonClick(TObject *Sender)  // EllipseButton
{
  DrawingTool = dtEllipse;
}

void __fastcall TForm1::RoundedRectButtonClick(TObject *Sender) // RoundRectBtn
{
  DrawingTool = dtRoundRect;
}
```

Using drawing tools

Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected.

To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler.

This section describes:

- Drawing shapes.
- Sharing code among event handlers.

Drawing shapes

Drawing shapes is just as easy as drawing lines: Each one takes a single statement; you just need the coordinates.

Here's a rewrite of the *OnMouseUp* event handler that draws shapes for all four tools:

Delphi example

```
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift: TShiftState;
                             X,Y: Integer);
begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
       Canvas.LineTo(X, Y)
      end;
   dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
   dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
   dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                   (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
 Drawing := False;
end;
```

C++ example

```
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
                                    TShiftState Shift, int X, int Y){
  switch (DrawingTool)
  {
    case dtLine:
      Canvas->MoveTo(Origin.x, Origin.y);
      Canvas->LineTo(X, Y);
      break;
    case dtRectangle:
      Canvas->Rectangle(Origin.x, Origin.y, X, Y);
```

```
        break;
      case dtEllipse:
        Canvas->Ellipse(Origin.x, Origin.y, X, Y);
        break;
      case dtRoundRect:
        Canvas->Rectangle(Origin.x, Origin.y, X, Y, (Origin.x - X)/2,
                          (Origin.y - Y)/2);
        break;
    }
    Drawing = false;
}
```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

Delphi example

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;
   case DrawingTool of
      dtLine: begin
                Canvas.MoveTo(Origin.X, Origin.Y);
               Canvas.LineTo(MovePt.X, MovePt.Y);
               Canvas.MoveTo(Origin.X, Origin.Y);
               Canvas.LineTo(X, Y);
              end;
     dtRectangle: begin
                     Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
                    Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
                  end;
     dtEllipse: begin
                   Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                  Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                 end;
     dtRoundRect: begin
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                      (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                   Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                      (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                  end;
    end;
   MovePt := Point(X, Y);
  end;
 Canvas.Pen.Mode := pmCopy;
end;
```

C++ example

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    switch (DrawingTool)
    {
      case dtLine:
        Canvas->MoveTo(Origin.x, Origin.y);
        Canvas->LineTo(MovePt.x, MovePt.y);
        Canvas->MoveTo(Origin.x, Origin.y);
        Canvas->LineTo(X, Y);
        break;
      case dtRectangle:
        Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y);
        Canvas->Rectangle(Origin.x, Origin.y, X, Y);
        break;
      case dtEllipse:
        Canvas->Ellipse(Origin.x, Origin.y, MovePt.x, MovePt.y);
        Canvas->Ellipse(Origin.x, Origin.y, X, Y);
        break;
      case dtRoundRect:
        Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y,
                         (Origin.x - MovePt.x)/2,(Origin.y - MovePt.y)/2);
        Canvas->Rectangle(Origin.x, Origin.y, X, Y,
                         (Origin.x - X)/2, (Origin.y - Y)/2);
        break;
    }
    MovePt = Point(X, Y);
  }
  Canvas->Pen->Mode = pmCopy;
}
```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next section shows all the shape-drawing code in a single routine that all mouse-event handlers can call.

Sharing code among event handlers

Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.

To add a method to a form:

1 Add the method declaration to the form object.

You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it's probably safest to make the shared method **private**.

2 Write the method implementation in the implementation part of the form unit (Delphi) or the .cpp file for the form's unit (C++).

The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

The following code adds a method to the form called *DrawShape* and calls it from each of the handlers. First, the declaration of *DrawShape* is added to the form object's declaration:

Delphi example

```
type
  TForm1 = class(TForm)
    ...{ fields and methods declared here}
 public
    { Public declarations }
   procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;
```

C++ example

```
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};

class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X,
  int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
private:// User declarations
  void __fastcall DrawShape(TPoint TopLeft, TPoint BottomRight, TPenMode AMode);
public:// User declarations
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  //field to track whether button was pressed
  TPoint Origin, MovePt;  // fields to store points
  TDrawingTool DrawingTool;  // field to hold current tool

};
```

Then, the implementation of *DrawShape* is written in the implementation part of the unit (Delphi) or the .cpp file for the unit (C++):

Delphi example

```
implementation
{$R *.FRM}
...{ other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
  begin
    Pen.Mode := AMode;
   case DrawingTool of
      dtLine:
        begin
          MoveTo(TopLeft.X, TopLeft.Y);
         LineTo(BottomRight.X, BottomRight.Y);
```

```
          end;
        dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
        dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
        dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
          (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
      end;
    end;
end;
```

C++ example

```
void __fastcall TForm1::DrawShape(TPoint TopLeft, TPoint BottomRight,
  TPenMode AMode)
{
  Canvas->Pen->Mode = AMode;
  switch (DrawingTool)
  {
    case dtLine:
      Canvas->MoveTo(TopLeft.x, TopLeft.y);
      Canvas->LineTo(BottomRight.x, BottomRight.y);
      break;
    case dtRectangle:
      Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y);
      break;
    case dtEllipse:
      Canvas->Ellipse(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y);
      break;
    case dtRoundRect:
      Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y,
                (TopLeft.x - BottomRight.x)/2,(TopLeft.y - BottomRight.y)/2);
      break;
  }
}
```

The other event handlers are modified to call *DrawShape*.

Delphi example

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy);{ draw the final shape }
 Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor);{ erase the previous shape }
   MovePt := Point(X, Y);{ record the current point }
   DrawShape(Origin, MovePt, pmNotXor);{ draw the current shape }
  end;
end;
```

C++ example

```
void __fastcall TForm1::FormMouseUp(TObject *Sender)
{
  DrawShape(Origin, Point(X,Y), pmCopy); // draw the final shape
  Drawing = false;
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    DrawShape(Origin, MovePt, pmNotXor); // erase previous shape
    MovePt = Point(X, Y);
    DrawShape(Origin, MovePt, pmNotXor); // draw current shape
  }
}
```

Drawing on a graphic

You don't need any components to manipulate your application's graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses an image control component to display the graphic on a form.

Once you move the application's drawing to the graphic in the image control, it is easy to add printing, clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, drawings, icons or whatever other graphics classes that have been installed such as jpeg graphics.

Note Because you are drawing on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from a bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its paint message. But if you are drawing directly onto the canvas property of a control, the picture object is displayed immediately.

Making scrollable graphics

The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it, you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.

Adding an image control

An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

Note "Adding graphics to controls" on page 7-12 shows how to use graphics in controls.

Placing the control

You can place an image control anywhere on a form. If you take advantage of the image control's ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component.

If you drop the image control on a scroll box already aligned to the form's client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image's picture. Then set the image control's properties.

Setting the initial bitmap size

When you place an image control, it is simply a container. However, you can set the image control's *Picture* property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in "Loading and saving graphics files" on page 11-25.

To create a blank bitmap when the application starts,

1 Attach a handler to the *OnCreate* event for the form that contains the image.

2 Create a bitmap object, and assign it to the *Graphic* property of the *TPicture* object that is the value of the image control's *Picture* property.

In this example, the image is in the application's main form, *Form1*, so the code attaches a handler to *Form1*'s *OnCreate* event:

Delphi example

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;{ temporary variable to hold the bitmap }
begin
  Bitmap := TBitmap.Create;{ construct the bitmap object }
  Bitmap.Width := 200;{ assign the initial width... }
  Bitmap.Height := 200;{ ...and the initial height }
  Image.Picture.Graphic := Bitmap;{ assign the bitmap to the image control }
  Bitmap.Free; {We are done with the bitmap, so free it }
end;
```

C++ example

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Graphics::TBitmap *Bitmap = new Graphics::TBitmap();   // create the bitmap object
  Bitmap->Width = 200;              // assign the initial width...
  Bitmap->Height = 200;             // ...and the initial height
  Image->Picture->Graphic = Bitmap;  // assign the bitmap to the image control
  delete Bitmap;                    // free the bitmap object
}
```

Assigning the bitmap to the picture's *Graphic* property copies the bitmap to the picture object. However, the picture object does not take ownership of the bitmap, so after making the assignment, you must free it.

If you run the application now, you see that client area of the form has a white region, representing the bitmap. If you size the window so that the client area cannot display the entire image, you'll see that the scroll box automatically shows scroll bars to allow display of the rest of the image. But if you try to draw on the image, you don't get any graphics, because the application is still drawing on the form, which is now behind the image and the scroll box.

Drawing on the bitmap

To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically, you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing.

An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of the *ScanLine* property. This example shows how to use *ScanLine* to get pixels one line at a time.

D Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
  x,y : integer;
  Bitmap : TBitmap;
  P : PByteArray;
begin
  Bitmap := TBitmap.create;
  try
   if OpenDialog1.Execute then
   begin
    Bitmap.LoadFromFile(OpenDialog1.FileName);
    for y := 0 to Bitmap.height -1 do
    begin
      P := Bitmap.ScanLine[y];
      for x := 0 to Bitmap.width -1 do
        P[x] := y;
    end;
   end;
canvas.draw(0,0,Bitmap);
  finally
    Bitmap.free;
  end;
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Graphics::TBitmap *pBitmap = new Graphics::TBitmap();
// This example shows drawing directly to the Bitmap
  Byte *ptr;
  try
  {
    pBitmap->LoadFromFile("C:\\Program Files\\Borland\\CBuilder\\Images\\Splash\\256color\\
factory.bmp ");
    for (int y = 0; y < pBitmap->Height; y++)
    {
      ptr = pBitmap->ScanLine[y];
      for (int x = 0; x < pBitmap->Width; x++)
        ptr[x] = (Byte)y;
    }
    Canvas->Draw(0,0,pBitmap);
  }
  catch (...)
  {
    ShowMessage("Could not load or alter bitmap");
  }
  delete pBitmap;
}
```

Loading and saving graphics files

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The image component makes it easy to load pictures from a file and save them again.

The components you use to load, save, and replace graphic images support many graphic formats including bitmap files, .pngs, .xpms, and so on. They also support installable graphic classes.

The way to load and save graphics files is the similar to any other files and is described in the following sections:

- Loading a picture from a file.
- Saving a picture to a file.
- Replacing the picture.

Loading a picture from a file

Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture.

To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object.

The following code gets a file name from an open picture file dialog box, and then loads that file into an image control named *Image*:

Delphi example

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
  begin
    CurrentFile := OpenPictureDialog1.FileName;
   Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

C++ example

```
void __fastcall TForm1::Open1Click(TObject *Sender)
{
  if (OpenPictureDialog1->Execute())
  {
    CurrentFile = OpenPictureDialog1->FileName;
    Image->Picture->LoadFromFile(CurrentFile);
  }
}
```

Saving a picture to a file

The picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well.

To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object.

The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the next section.

The following pair of event handlers, attached to the File|Save and File|Save As menu items, respectively, handle the resaving of named files, saving of unnamed files, and saving existing files under new names.

Delphi example

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile){ save if already named }
 else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then{ get a file name }
  begin
    CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
   Save1Click(Sender);{ then save normally }
  end;
end;
```

C++ example

```
void __fastcall TForm1::Save1Click(TObject *Sender)
{
  if (!CurrentFile.IsEmpty())
    Image->Picture->SaveToFile(CurrentFile);   // save if already named
else SaveAs1Click(Sender);                     // otherwise get a name
}

void __fastcall TForm1::SaveAs1Click(TObject *Sender)
{
  if (SaveDialog1->Execute())             // get a file name
  {
    CurrentFile = SaveDialog1->FileName;  // save user-specified name
    Save1Click(Sender);                  // then save normally
  }
}
```

Replacing the picture

You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an image control, assign a new graphic to the image control's *Picture* object.

Creating the new graphic is the same process you used to create the initial graphic (see "Setting the initial bitmap size" on page 11-23), but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to present a dialog box.

With a dialog box in your project, add it to the uses clause in the unit (Delphi) or an include statement for bmpdlg.hpp in the .cpp file (C++) for your main form. You can then attach an event handler to the File|New menu item's *OnClick* event. Here's an example:

Delphi example

```
procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap;{ temporary variable for the new bitmap }
begin
  with NewBMPForm do
  begin
    ActiveControl := WidthEdit;{ make sure focus is on width field }
   WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
   HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
   if ShowModal <> idCancel then{ continue if user doesn't cancel dialog box }
```

```
      begin
        Bitmap := TBitmap.Create;{ create fresh bitmap object }
       Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
       Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
       Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
       CurrentFile := '';{ indicate unnamed file }
       Bitmap.Free;
      end;
    end;
  end;
```

C++ example

```
void __fastcall TForm1::New1Click(TObject *Sender)
{
     Graphics::TBitmap *Bitmap;
  // make sure focus is on width field
     NewBMPForm->ActiveControl = NewBMPForm->WidthEdit;
  // initialize to current dimensions as default ...
     NewBMPForm->WidthEdit->Text = IntToStr(Image->Picture->Graphic->Width);
     NewBMPForm->HeightEdit->Text = IntToStr(Image->Picture->Graphic->Height);
     if (NewBMPForm->ShowModal() != IDCANCEL){        // if user does not cancel dialog...
        Bitmap = new Graphics::TBitmap();             // create a new bitmap object
        // use specified dimensions
        Bitmap->Width = StrToInt(NewBMPForm->WidthEdit->Text);
        Bitmap->Height = StrToInt(NewBMPForm->HeightEdit->Text);
        Image->Picture->Graphic = Bitmap;            // replace graphic with new bitmap
        CurrentFile = EmptyStr;                      //indicate unnamed file
        delete Bitmap;
     }
}
```

Note Assigning a new bitmap to the picture object's *Graphic* property causes the picture object to copy the new graphic, but it does not take ownership of it. The picture object maintains its own internal graphic object. Because of this, the previous code frees the bitmap object after making the assignment.

Using the clipboard with graphics

You can use the clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The CLX clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the clipboard object in your application, in Delphi you must add the QClipbrd unit to the uses clause of any unit or in C++, add an include statement for QClipbrd.hpp to any .cpp file that needs to access clipboard data.

Data that is stored on the clipboard when using CLX is stored as a MIME type with an associated *TStream* object. CLX provides predefined constants for the following MIME types.

Table 11.4 CLX MIME types and constants

MIME type	CLX constant
'image/delphi.bitmap'	SDelphiBitmap
'image/delphi.component'	SDelphiComponent
'image/delphi.picture'	SDelphiPicture
'image/delphi.drawing'	SDelphiDrawing

Copying graphics to the clipboard

You can copy any picture, including the contents of image controls, to the clipboard. Once on the clipboard, the picture is available to all applications.

To copy a picture to the clipboard, assign the picture to the clipboard object using the *Assign* method.

This code shows how to copy the picture from an image control named *Image* to the clipboard in response to a click on an Edit|Copy menu item:

Delphi example

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

C++ example

```
void __fastcall TForm1::Copy1Click(TObject *Sender)
{
  Clipboard()->Assign(Image->Picture);
}
```

Cutting graphics to the clipboard

Cutting a graphic to the clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the clipboard, first copy it to the clipboard, then erase the original.

In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the Edit|Cut menu item:

Delphi example

```
procedure TForm1.Cut1Click(Sender: TObject);
var
```

```
  ARect: TRect;
begin
  Copy1Click(Sender);{ copy picture to clipboard }
 with Image.Canvas do
  begin
    CopyMode := cmWhiteness;{ copy everything as white }
   ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
   CopyRect(ARect, Image.Canvas, ARect);{ copy bitmap over itself }
   CopyMode := cmSrcCopy;{ restore normal mode }
  end;
end;
```

C++ example

```
void __fastcall TForm1::Cut1Click(TObject *Sender)
{
  TRect ARect;
  Copy1Click(Sender);          // copy picture to clipboard
  Image->Canvas->CopyMode = cmWhiteness; // copy everything as white
  ARect = Rect(0, 0, Image->Width, Image->Height); // get dimensions of image
  Image->Canvas->CopyRect(ARect, Image->Canvas, ARect); // copy bitmap over self
  Image->Canvas->CopyMode = cmSrcCopy; // restore default mode
}
```

Pasting graphics from the clipboard

If the clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

To paste a graphic from the clipboard:

1 Call the clipboard's *Provides* method to see whether the clipboard contains a graphic.

  Provides is a Boolean function. It returns true if the clipboard contains an item of the type specified in the parameter. To test for graphics on cross-platform applications, you pass *SDelphiBitmap*.

2 Assign the clipboard to the destination.

This CLX code shows how to paste a picture from the clipboard into an image control in response to a click on an Edit | Paste menu item:

Delphi example

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
 if Clipboard.Provides(SDelphiBitmap) then { is there a bitmap on the clipboard? )
 begin
   Image1.Picture.Bitmap.Assign(Clipboard);
 end;
end;
```

C++ example

```
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
  QGraphics::TBitmap *Bitmap;
  if (Clipboard()->Provides(SDelphiBitmap)){
   Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

The graphic on the clipboard could come from this application, or it could have been copied from another application. You do not need to check the clipboard format in this case because the paste menu should be disabled when the clipboard does not contain a supported format.

Rubber banding example

This example describes the details of implementing the "rubber banding" effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The application draws lines and shapes on a window's canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing.

To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap.

The following topics describe the example:

- Responding to the mouse.
- Adding a field to a form object to track mouse actions.
- Refining line drawing.

Responding to the mouse

Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box).

This section covers:

- What's in a mouse event.
- Responding to a mouse-down action.
- Responding to a mouse-up action.
- Responding to a mouse move.

What's in a mouse event?

A mouse event occurs when a user moves the mouse in the user interface of an application. CLX has three mouse events.

Table 11.5 Mouse events

Event	Description
OnMouseDown event	Occurs when the user presses a mouse button with the mouse pointer over a control.
OnMouseMove event	Occurs when the user moves the mouse while the mouse pointer is over a control.
OnMouseUp event	Occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

When an application detects a mouse action, it calls whatever event handler you've defined for the corresponding event, passing five parameters. Use the information in those parameters to customize your responses to the events. The five parameters are as follows:

Table 11.6 Mouse-event parameters

Parameter	Meaning
Sender	The object that detected the mouse action
Button	Indicates which mouse button was involved: *mbLeft*, *mbMiddle*, or *mbRight*
Shift	Indicates the state of the *Alt*, *Ctrl*, and *Shift* keys at the time of the mouse action
X, *Y*	The coordinates where the event occurred

Most of the time, you need the coordinates returned in a mouse-event handler, but sometimes you also need to check *Button* to determine which mouse button caused the event.

KylixResponding to a mouse-down action

Whenever the user presses a button on the mouse, an *OnMouseDown* event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event.

The IDE generates an empty handler for a mouse-down event on the form:

Delphi example

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
end;
```

C++ example

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
```

```
{

}
```

Responding to a mouse-down action

The following code displays the string 'Here!' at the location on a form clicked with the mouse:

Delphi example

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.TextOut(X, Y, 'Here!');{ write text at (X, Y) }
end;
```

C++ example

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->TextOut(X, Y, "Here!");// write text at (X, Y)
}
```

When the application runs, you can press the mouse button down with the mouse cursor on the form and have the string, “Here!” appear at the point clicked. This code sets the current drawing position to the coordinates where the user presses the button:

Delphi example

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.MoveTo(X, Y);{ set pen position }
end;
```

C++ example

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(X, Y);// set pen position
}
```

Pressing the mouse button now sets the pen position, setting the line’s starting point. To draw a line to the point where the user releases the button, you need to respond to a mouse-up event.

Responding to a mouse-up action

An *OnMouseUp* event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions, define a handler for the *OnMouseUp* event.

Here's a simple *OnMouseUp* event handler that draws a line to the point of the mouse-button release:

Delphi example

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line from PenPos to (X, Y) }
end;
```

C++ example

```
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);// draw line from PenPos to (X, Y)
}
```

This code lets a user draw lines by clicking, dragging, and releasing. In this case, the user cannot see the line until the mouse button is released.

Responding to a mouse move

An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves.

To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location of the *OnMouseMove* event:

Delphi example

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line to current position }
end;
```

C++ example

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);// draw line to current position
}
```

With this code, moving the mouse over the form causes drawing to follow the mouse, even before the mouse button is pressed.

Mouse-move events occur even when you haven't pressed the mouse button.

If you want to track whether there is a mouse button pressed, you need to add an object field to the form object.

Adding a field to a form object to track mouse actions

To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Kylix adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file.

In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its value when the user presses the mouse button.

To add a field to an object, edit the object's type definition, specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Kylix "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

The following code gives a form a field called *Drawing* of the boolean type in the form object's declaration. It also adds two fields to store points *Origin* and *MovePt* of type TPoint.

D Delphi example

```
type
  TForm1 = class(TForm)
   procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
 public
    Drawing: Boolean;{ field to track whether button was pressed }
    Origin, MovePt: TPoint;{ fields to store points }
  end;
```

C++ example

```
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
```

```
    void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X,
    int Y);
    void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);
  private:// User declarations
  public:// User declarations
    __fastcall TForm1(TComponent* Owner);
    bool Drawing;  //field to track whether button was pressed
     TPoint Origin, MovePt;  // fields to store points
  };
```

When you have a *Drawing* field to track whether to draw, set it to true when the user presses the mouse button, and false when the user releases it:

Delphi example

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;{ set the Drawing flag }
 Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
 Drawing := False;{ clear the Drawing flag }
end;
```

C++ example

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
}

void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);      // draw line from PenPos to (X, Y)
  Drawing = false;           // clear the Drawing flag
}
```

Then you can modify the *OnMouseMove* event handler to draw only when *Drawing* is true:

Delphi example

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then{ only draw if Drawing flag is set }
   Canvas.LineTo(X, Y);
```

```
end;
```

C++ example

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
    Canvas->LineTo(X, Y);// only draw if mouse is down
}
```

This results in drawing only between the mouse-down and mouse-up events, but you still get a scribbled line that tracks the mouse movements instead of a straight line.

The problem is that each time you move the mouse, the mouse-move event handler calls *LineTo*, which moves the pen position, so by the time you release the button, you've lost the point where the straight line was supposed to start.

Refining line drawing

With fields in place to track various points, you can refine an application's line drawing.

Tracking the origin point

When drawing lines, track the point where the line starts with the *Origin* field. *Origin* must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:

Delphi example

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
 Canvas.LineTo(X, Y);
 Drawing := False;
end;
```

C++ example

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
}

void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
  Canvas->LineTo(X, Y);                // draw line from PenPos to (X, Y)
  Drawing = false;                     // clear the Drawing flag
}
```

Those changes get the application to draw the final line again, but they do not draw any intermediate actions--the application does not yet support “rubber banding.”

Tracking movement

The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position,* not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:

Delphi example

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
   Canvas.LineTo(X, Y);
  end;
end;
```

C++ example

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(X, Y);
  }
}
```

The above tracks the current mouse position, but the intermediate lines do not go away, so you can hardly see the final line. The example needs to erase each line before drawing the next one, by keeping track of where the previous one was. The *MovePt* field allows you to do this.

MovePt must be set to the endpoint of each intermediate line, so you can use *MovePt* and *Origin* to erase that line the next time a line is drawn:

Delphi example

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);
 MovePt := Point(X, Y);{ keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;{ use XOR mode to draw/erase }
   Canvas.MoveTo(Origin.X, Origin.Y);{ move pen back to origin }
   Canvas.LineTo(MovePt.X, MovePt.Y);{ erase the old line }
   Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
   Canvas.LineTo(X, Y);{ draw the new line }
  end;
 MovePt := Point(X, Y);{ record point for next move }
 Canvas.Pen.Mode := pmCopy;
end;
```

C++ example

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
  MovePt = Point(X, Y);      // record last endpoint
}

void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(MovePt.x, MovePt.y);  // erase old line
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point again
    Canvas->LineTo(X, Y);                // draw new line
  }
  MovePt = Point(X, Y);      // record new endpoint
  Canvas->Pen->Mode = pmCopy;
}
```

Now you get a “rubber band” effect when you draw the line. By changing the pen’s mode to *pmNotXor*, you have it combine your line with the background pixels. When you go to erase the line, you’re actually setting the pixels back to the way they were. By changing the pen mode back to *pmCopy* (its default value) after drawing the lines, you ensure that the pen is ready to do its final drawing when you release the mouse button.

Working with multimedia

You can add multimedia components to your applications. To do this, you can use the *TAnimate* Common Controlspage of the Component palette. Use the animate component when you want to add silent video clips to your application.For more information on the *TAnimate* component, see the online Help.

The following topic is discussed in this section:

- Adding silent video clips to an application

Adding silent video clips to an application

With the animation control, you can add silent video clips to your application.

To add a silent video clip to an application:

1. Double-click the *TAnimate* icon on the Common Controls page of the Component palette. This automatically puts an animation control on the form window in which you want to display the video clip.
2. Using the Object Inspector, select the *Name* property and enter a new *nam*e for your animation control. You will use this name when you call the animation control. (Follow the standard rules for naming Kylix identifiers).

   Always work directly with the Object Inspector when setting design time properties and creating event handlers.
3. Do the following:
   - Select the *FileName* property and click the ellipsis (…) button, choose a GIF file from any available local or network directories and click Open in the Open GIF dialog.

   Cross-platform applications use animated GIFs only.

   This loads the GIF file into memory. If you want to display the first frame of the GIF clip on-screen until it is played using the *Active* property or the *Play* method, then set the *Open* property to true.
4. Set the *Repetitions* property to the number of times you want to the GIF clip to play. If this value is 0, then the sequence is repeated until the *Stop* method is called.
5. Make any other changes to the animation control settings. For example, if you want to change the first frame displayed when animation control opens, then set the *StartFrame* property to the desired frame value.
6. Set the *Active* property to true using the drop-down list or write an event handler to run the GIF clip when a specific event takes place at runtime. For example, to

activate the GIF clip when a button object is clicked, write the button’s *OnClick* event specifying that.

Note If you make any changes to the form or any of the components on the form after setting *Active* to true, the *Active* property becomes false and you have to reset it to true. Do this either just before runtime or at runtime.

Chapter

12

Writing multi-threaded applications

CLX provides several objects that make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by:

- **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- **Organizing program behavior.** Often, a program's behavior can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases. Use threads to assign priorities to various program tasks so that you can give more CPU time to more critical tasks.
- **Multiprocessing.** If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

Note Linux is a multiprocessing operating system with Intel MP architecture. Processes are separate tasks each with their own rights and responsibilities. Each individual process runs in its own virtual address space and is not capable of interacting with another process except through secure, kernel-managed mechanisms.

Defining thread objects

For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

Note Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the *BeginThread* function. Even when using *BeginThread*, you can still benefit from some of the thread synchronization objects and methods described in "Coordinating threads" on page 12-8.

To use a thread object in your application, you must create a new descendant of *TThread*. To create a descendant of *TThread*, choose File | New | Other from the main menu. In the New Items dialog box, double-click Thread Object and enter a class name, such as *TMyThread*. After you click OK, Kylix creates a new unit (Delphi) or .cpp and header (C++) file to implement the thread.

Note Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog box does not automatically prepend a 'T' to the front of the class name you provide.

The automatically generated unit (Delphi) or .cpp (C++) file contains the skeleton code for your new thread class. If you named your thread *TMyThread*, it would look like the following:

Delphi example

```
unit Unit2;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.
```

C++ example

```
//---------------------------------------------------------------------------
#include <clx.h>
#pragma hdrstop

#include "Unit2.h"
#pragma package(smart_init)
#pragma resource "*.xfm"
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
}
```

```
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
  // ---- Place thread code here ----
}
//---------------------------------------------------------------------------
```

You must write the code for the *Execute* method and in C++, the constructor. These steps are described in the following sections.

Initializing the thread

In Delphi, if you want to write initialization code for your new thread class, you must override the Create method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation.

In C++, use the constructor to initialize your new thread class.

This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.

Assigning a default priority

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the *Priority* property.

Note *Priority* values are integers in Linux where a lower number indicates a higher priority. Priority values can be set to 0 through 99 and they determine how much time the thread gets to use the CPU. The higher the priority, the more time the thread gets. On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See *TThread* and *Priority* online Help for details.

Warning Boosting the thread priority of a CPU intensive operation may “starve” other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application’s performance:

Delphi example

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
  Priority := tpIdle;
end;
```

C++ example

```
//---------------------------------------------------------------------------
```

```
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
  Priority = tpIdle;
}
//---------------------------------------------------------------------------
```

Indicating when threads are freed

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the *FreeOnTerminate* property to true.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to false and then explicitly freeing the first thread from the second.

Writing the thread function

The *Execute* method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

Using the main CLX thread

When you use objects from the CLX object hierarchy, their properties and methods are not guaranteed to be thread-safe. That is, accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main thread is set aside to access CLX objects.

If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's *Synchronize* method. For example:

D **Delphi example**

```
procedure TMyThread.PushTheButton;
begin
  Button1.Click;
end;
⋮
procedure TMyThread.Execute;
begin
```

```
  ⋮
  Synchronize(PushTheButton);
  ⋮
end;
```

C++ example

```
void __fastcall TMyThread::PushTheButton(void)
{
  Button1->Click();
}
⋮
void __fastcall TMyThread::Execute()
{
  ⋮
  Synchronize((TThreadMethod)PushTheButton);
  ⋮
}
```

Synchronize waits for the main thread to enter the message loop and then executes the passed method.

Note Because *Synchronize* uses the message loop, it does not work in console applications. You must use other mechanisms, such as critical sections, to protect access to CLX objects in console applications.

You do not always need to use the main thread. Some objects are thread-aware. Omitting the use of the *Synchronize* method when you know an object's methods are thread-safe will improve performance because you don't need to wait for the CLX thread to enter its message loop. You do not need to use the *Synchronize* method for the following objects:

- Data access components are thread-safe as follows: For dbDirect, as long as the vendor client library is thread-safe, the dbDirect components will be thread-safe.

  When using data access components, you must still wrap all calls that involve data-aware controls in the *Synchronize* method. Thus, for example, you need to synchronize calls that link a data control to a dataset by setting the *DataSet* property of the data source object, but you don't need to synchronize to access the data in a field of the dataset.
- DataCLX objects are thread-safe although VisualCLX objects are not.
- Graphics objects are thread-safe. You do not need to use the main CLX thread to access *TFont*, *TPen*, *TBrush, TBitmap, TDrawing*, or *TIcon*. Canvas objects can be used outside the *Synchronize* method by locking them (see "Locking objects" on page 12-9).
- While list objects are not thread-safe, you can use a thread-safe version, *TThreadList*, instead of *TList*.

Call the *CheckSynchronize* routine periodically within the main thread of your application so that background threads can synchronize their execution with the main thread. The best place to call *CheckSynchronize* is when the application is idle (for example, from an *OnIdle* event handler). This ensures that it is safe to make method calls in the background thread.

Using thread-local variables

Your *Execute* method and any of the routines it calls have their own local variables, just like any other Delphi or C++ routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads.

Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by:

Delphi example

In Delphi, declaring it in a **threadvar** section:

```
threadvar
    x : integer;
```

C++ example

In C++, adding the **__thread** modifier to the variable declaration:

```
int __thread x;
```

The previous code declares an integer type variable that is private to each thread in the application, but global within each thread.

The threadvar section (Delphi) or **__thread** modifier (C++) can only be used for global variables (file-scope and static variables in C++) . Pointer and Function variables can’t be thread variables. Types that use copy-on-write semantics, such as long strings (Delphi) or AnsiStrings (C++) don’t work as thread variables either.

C++ example

In C++, a program element that requires runtime initialization or runtime finalization cannot be declared to be a **__thread** type. The following declarations require runtime initialization and are therefore illegal.

```
int f( );
int __thread x = f( );   // illegal
```

In C++, instantiation of a class with a user-defined constructor or destructor requires runtime initialization and is therefore illegal:

```
class X  {
   X( );
   ~X( );
};
X __thread myclass;   // illegal
```

Checking for termination by other threads

Your thread begins running when the *Execute* method is called (see "Executing thread objects" on page 12-13) and continues until *Execute* finishes. This reflects the model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied.

You can allow other threads to signal that it is time for your thread to finish executing by checking the *Terminated* property. When another thread tries to terminate your thread, it calls the *Terminate* method. *Terminate* sets your thread's *Terminated* property to true. It is up to your *Execute* method to implement the *Terminate* method by checking and responding to the *Terminated* property. The following example shows one way to do this:

Delphi example

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

C++ example

```
void __fastcall TMyThread::Execute()
{
  while (!Terminated)
    PerformSomeTask();
}
```

Handling exceptions in the thread function

The *Execute* method must catch all exceptions that occur in the thread. If you fail to catch an exception in your thread function, your application can cause access violations. This may not be obvious when you are developing your application, because the IDE catches the exception, but when you run your application outside of the debugger, the exception will cause a runtime error and the application will stop running.

To catch the exceptions that occur inside your thread function, add a try...except block (Delphi) or try...catch block (C++) to the implementation of the *Execute* method:

Delphi example

```
procedure TMyThread.Execute;
begin
  try
    while not Terminated do
      PerformSomeTask;
  except
    { do something with exceptions }
  end;
end;
```

C++ example

```
void __fastcall TMyThread::Execute()
{
  try
  {
    while (!Terminated)
      PerformSomeTask();
  }
  catch (...)
  {
    // do something with exceptions
  }
}
```

Writing clean-up code

You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an *OnTerminate* event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the *Execute* method follows.

The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main CLX thread of your application. This has two implications:

- You can't use any thread-local variables in an *OnTerminate* event handler (unless you want the main CLX thread values).
- You can safely access any components and CLX objects from the *OnTerminate* event handler without worrying about clashing with other threads.

For more information about the main CLX thread, see "Using the main CLX thread" on page 12-4.

Coordinating threads

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

Avoiding simultaneous access

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

Locking objects

Some objects have built-in locking that prevents the execution of other threads from using that object instance.

For example, canvas objects (*TCanvas* and descendants) have a *Lock* method that prevents other threads from accessing the canvas until the *Unlock* method is called.

CLX also includes a thread-safe list object, *TThreadList*. Calling the *LockList* method returns the list object while also blocking other execution threads from using the list until the *UnlockList* method is called. Calls to *TCanvas's Lock* method or the *LockList* method can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call in the same thread.

Using critical sections

If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of *TCriticalSection. TCriticalSection* has two methods, *Acquire* (which blocks other threads from executing the section) and *Release* (which removes the block).

Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the *Acquire* method to ensure that no other thread is using it. When finished, threads call the *Release* method so that other threads can access the global memory by calling *Acquire*.

Warning Critical sections only work if every thread uses them to access the associated global memory. Threads that ignore the critical section and access the global memory without calling *Acquire* can introduce problems of simultaneous access.

For example, consider an application that has a global critical section variable, *LockXY* (Delphi) or *pLockXY* (C++), that blocks access to global variables X and Y. Any thread that uses X or Y must surround that use with calls to the critical section such as the following:

Delphi example

```
LockXY.Acquire; { lock out other threads }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

C++ example

```
pLockXY->Acquire(); // lock out other threads
try
{
  Y = sin(X);
}
__finally
{
  pLockXY->Release();
}
```

Using the multi-read exclusive-write synchronizer

When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it.

When you have some global memory that is read often, but to which threads occasionally write, you can protect it using *TMultiReadExclusiveWriteSynchronizer*. This object acts like a critical section, but allows multiple threads to read the memory it protects as long as no thread is writing to it. Threads must have exclusive access to write to memory protected by *TMultiReadExclusiveWriteSynchronizer*.

To use a multi-read exclusive-write synchronizer, create a global instance of *TMultiReadExclusiveWriteSynchronizer* that is associated with the global memory you want to protect. Every thread that reads from this memory must first call the *BeginRead* method. *BeginRead* ensures that no other thread is currently writing to the memory. When a thread finishes reading the protected memory, it calls the *EndRead* method. Any thread that writes to the protected memory must call *BeginWrite* first. *BeginWrite* ensures that no other thread is currently reading or writing to the memory. When a thread finishes writing to the protected memory, it calls the *EndWrite* method, so that threads waiting to read the memory can begin.

Warning Like critical sections, the multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Threads that ignore the synchronizer and access the global memory without calling *BeginRead* or *BeginWrite* introduce problems of simultaneous access.

Other techniques for sharing memory

When using objects in CLX, use the main CLX thread to execute your code. Using the main thread ensures that the object does not indirectly access any memory that is also used by CLX objects in other threads. See "Using the main CLX thread" on page 12-4 for more information on the main CLX thread.

If the global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads. See "Using thread-local variables" on page 12-6 for more information about thread-local variables.

Waiting for other threads

If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either wait for another thread to completely finish executing, or you can wait for another thread to signal that it has completed a task.

Waiting for a thread to finish executing

To wait for another thread to finish executing, use the *WaitFor* method of that other thread. *WaitFor* doesn't return until the other thread terminates, either by finishing its own *Execute* method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:

Delphi example

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
  end;
  ThreadList1.UnlockList;
end;
```

C++ example

```
if (pListFillingThread->WaitFor())
{
  TList *pList = ThreadList1->LockList();
  for (int i = 0; i < pList->Count; i++)
    ProcessItem(pList->Items[i]);
  ThreadList1->UnlockList();
}
```

In the previous example, the list items were only accessed when the *WaitFor* method indicated that the list was successfully filled. This return value must be assigned by the *Execute* method of the thread that was waited for. However, because threads that call *WaitFor* want to know the result of thread execution, not code that calls *Execute*, the *Execute* method does not return any value. Instead, the *Execute* method sets the *ReturnValue* property. *ReturnValue* is then returned by the *WaitFor* method when it is called by other threads. Return values are integers. Your application determines their meaning.

Waiting for a task to be completed

Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (*TEvent*) should be created with global scope so that they can act like signals that are visible to all threads.

When a thread completes an operation that other threads depend on, it calls the *SetEvent* method. *SetEvent* turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the *ResetEvent* method.

For example, consider a situation where you must wait for several threads to complete their execution rather than a single thread. Because you don't know which thread will finish last, you can't simply use the *WaitFor* method of one of the threads.

Instead, you can have each thread increment a counter when it is finished, and have the last thread signal that they are all done by setting an event.

The following code shows the end of the *OnTerminate* event handler for all of the threads that must complete. *CounterGuard* is a global critical section object that prevents multiple threads from using the counter at the same time. *Counter* is a global variable that counts the number of threads that have completed.

Delphi example

```
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
  ⋮
  CounterGuard.Acquire; { obtain a lock on the counter }
  Dec(Counter);   { decrement the global counter variable }
  if Counter = 0 then
    Event1.SetEvent; { signal if this is the last thread }
  CounterGuard.Release; { release the lock on the counter }
  ⋮
end;
```

C++ example

```
void __fastcall TDataModule::TaskThreadTerminate(TObject *Sender)
{
  ⋮
  CounterGuard->Acquire(); // lock the counter
  if (--Counter == 0)    // decrement the global counter
    Event1->SetEvent(); // signal if this is the last thread
  CounterGuard->Release(); // release the lock on the counter
}
```

The main thread initializes the Counter variable, launches the task threads, and waits for the signal that they are all done by calling the *WaitFor* method. *WaitFor* waits for a specified time period for the signal to be set, and returns one of the values from Table 12.1.

Table 12.1 WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the time-out period elapsed.
wrError	An error occurred while waiting.

The following shows how the main thread launches the task threads and then resumes when they have all completed:

Delphi example

```
Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
```

```
  TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{ now continue with the main thread. All task threads have finished }
```

C++ example

```
Event1->ResetEvent(); // clear the event before launching the threads
for (int i = 0; i < Counter; i++)
  new TaskThread(false); // create and launch task threads
if (Event1->WaitFor(20000) != wrSignaled)
  throw Exception;
// now continue with the main thread, all task threads have finished
```

Note If you do not want to stop waiting for an event after a specified time period, pass the *WaitFor* method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

Executing thread objects

Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the *Resume* method. To create a thread so that it starts up immediately, set the constructor's *CreateSuspended* parameter to false**.** For example, the following line creates a thread and starts its execution:

```
SecondThread := TMyThread.Create(false); {create and run the thread }
```

```
TMyThread *SecondThread = new TMyThread(false); // create and run the thread
```

Warning Do not create too many threads in your application. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.

Overriding the default priority

CPU time is based on priority. You can control how the kernel's process scheduler schedules a thread using the *Policy* and *Priority* properties. *Policy* assigns a schedule policy for the thread as compared to other threads on the system. Superuser privilege is required for setting the policy. Depending on the value of the *Policy* property, you can set a thread's priority relative to other threads in the process. Priority is a value between 0 and 99 that determined how much time the thread gets to use the CPU. The higher the priority, the more time the thread gets.

When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in "Initializing the thread" on page 12-3. However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:

Delphi example

```
SecondThread :=  TMyThread.Create(True); { create but don't run }
if SecondThread.Policy <> SCHED_OTHER then
SecondThread.Priority := 1Lower; { set the priority lower than normal }
SecondThread.Resume; { now run the thread }
```

C++ example

```
TMyThread *SecondThread = new TMyThread(true); // create but don't run
if (SecondThread->Policy != SCHED_OTHER)
SecondThread->Priority = 1Lower; // set the priority lower than normal
SecondThread->Resume(); // now run the thread
```

The *Policy* must be set to either SCHED_RR or SCHED_FIFO (and not SCHED_OTHER) for the *Priority* to be changed.

Note On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See *TThread* and *Priority* online Help for details.

Starting and stopping threads

A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its *Suspend* method. When it is safe for the thread to resume, call its *Resume* method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by a call to *Resume*.

You can request that a thread end execution prematurely by calling the *Terminate* method. *Terminate* sets the thread's *Terminated* property to true. If you have implemented the *Execute* method properly, it checks the *Terminated* property periodically, and stops execution when *Terminated* is true.

Debugging multi-threaded applications

When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose View | Debug Windows | Threads from the main menu.

When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread. Right-click the Thread Status box to access commands that locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

The Thread Status box lists all your application's execution threads by their thread ID. If you are using thread objects, the thread ID is the value of the *ThreadID* property. If you are not using thread objects, the thread ID for each thread is returned by the call to *BeginThread*.

For additional details on the Thread Status box, see online Help.

Chapter

13

Exception handling

Exceptions are exceptional conditions that require special handling. They include errors that occur at runtime, such as divide by zero, and the exhaustion of free store. Exception handling provides a standard way of dealing with errors, discovering both anticipated and unanticipated problems, and enables developers to recognize, track down, and fix bugs.

When an error occurs, the program raises an exception, meaning it creates an exception object and rolls back the stack to the first point it finds where you have code to handle the exception. The exception object usually contains information about what happened. This allows another part of the program to diagnose the cause of the exception.

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application will present a message box describing the error. Your job, then, is to recognize places where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

Note Both Delphi code and C++ code can use exceptions. You can’t however, use exceptions in C code.

Defining protected blocks

To prepare for exceptions, you place statements that might raise them in a try block. If one of these statements does raise an exception, control is transferred to an exception handler that handles that type of exception, then leaves the block. The exception handler is said to catch the exception and specifies the actions to take. By using try blocks and exception handlers, you can move error checking and error handling out of the main flow of your algorithms, resulting in simpler, more readable code.

In Delphi, you start a protected block using the keyword try. The exception handler must immediately follow the try block. It is introduced by the keyword except, and signals the end of the try block This syntax is illustrated in the following code. If the *SetFieldValue* method fails and raises an *EIntegerRange* exception, execution jumps to the exception-handling part, which displays an error message. Execution resumes outside the block.

```
try
  SetFieldValue(dataField, userValue);
except
  on E: EIntegerRange do
    ShowMessage(Format('Expected value between %d and %d, but got %d',
                       E.Min, E.Max, E.Value));
end;
  ⋮ { execution resumes here, outside the protected block }
```

In C++, you also start a protected block using the keyword try. The try block is the block immediately following the try keyword. This is a single statement or a set of statements enclosed in curly braces. The exception handler, which must immediately follow the try block, is introduced using the keyword catch. The C++ version of the previous example is as follows:

```
try
{
    SetFieldValue(dataField, userValue);
}
catch (EIntegerRange &E)
{
  ShowMessage(Format("Expected value between %d and %d, but got %d\n",
              E.Min, E.Max, E.Value));
}
// execution resumes here, outside the protected block
```

In either language, you must have an exception handling block (described in "Writing exception handlers" on page 13-5) or a finally block (described in "Writing finally blocks" on page 13-11) immediately after the try block. An exception handling block should include a handler for each exception that the statements in the try block can generate.

Writing the try block

The first part of a protected block is the try block. The try block contains code that can potentially raise an exception. The exception can be raised either directly in the try block, or by code that is called by statements in the try block. That is, if code in a try block calls a routine that doesn't define its own exception handler, then any exceptions raised inside that routine cause execution to pass to the exception-handler associated with the try block. Keep in mind that exceptions don't come just from your code. A call to an RTL routine or another component in your application can also raise an exception.

The following example demonstrates catching an exception thrown from a *TFileStream* object.

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fileStream: TFileStream;
begin

     try
        (* Attempt to open a non-existant file *)
        fileStream := TFileStream.Create('NOT_THERE.FILE', fmOpenRead);

        (* Process the file contents... *)

        fileStream.Free;
     except
        on EFOpenError do Application.MessageBox('EFOpenError Raised', 'Exception Demo',
[smbOk]);
     else
        Application.MessageBox('Exception Raised.', 'Exception Demo', [smbOk]);
     end;
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   TFileStream *fileStream;
   TMessageButtons buttons;


      try {
         buttons << smbOK;
         // Attempt to open a non-existent file
         fileStream = new TFileStream("NOT_THERE.FILE", fmOpenRead);

         // Process the file contents...

         delete fileStream;
      }
      catch(EFOpenError &e) {
         Application->MessageBox("EFOpenError Raised.", "Exception Demo", buttons);
      }
      catch(...) {
         Application->MessageBox("Exception Raised.", "Exception Demo", buttons);
      }
}
```

Using a try block makes your code easier to read. Instead of sprinkling error-handling code throughout your program, you isolate it in exception handlers so that the flow of your algorithms is more obvious.

This is especially true when performing complex calculations involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid. By using exceptions, you can spell out the normal expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every time to make sure you can proceed with each step in the calculation.

Raising an exception in Delphi

In Delphi, to indicate a disruptive error condition, you can raise an exception by constructing an instance of an exception object that describes the error condition and calling the reserved word raise.

To raise an exception, call the reserved word raise, followed by an instance of an exception object. This establishes the exception as coming from a particular address. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

For example, given the following declaration,

```
type
  EPasswordInvalid = class(Exception);
```

you can raise a "password invalid" exception at any time by calling raise with an instance of *EPasswordInvalid*, like this:

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered');
```

Raising an exception sets the *ErrorAddr* variable in the System unit to the address where the application raised the exception. You can refer to *ErrorAddr* in your exception handlers, for example, to notify the user where the error occurred. You can also specify a value in the raise clause that appears in *ErrorAddr* when an exception occurs.

Warning Do not assign a value to *ErrorAddr* yourself. It is intended as read-only.

To specify an error address for an exception, add the reserved word *at* after the exception instance, followed by an address expression such as an identifier.

Raising an exception in C++

To raise an exception in C++, use the throw keyword. The use of throw in C++ is far more general than the syntax for raising exceptions in Delphi. For example, objects in C++ may be thrown by value, reference, or pointer (in Delphi, you can only use by reference):

```
// throw an object, to be caught by value or reference
throw EIntegerRange(0, 10, userValue);

// throw an object to be caught by pointer
throw new EIntegerRange(0, 10, userValue);
```

In addition, while Delphi only lets you raise exception objects, the throw statement in C++ can throw other types as well. Although it is not recommended, C++ lets you throw built-in types, such as integers or pointers:

```
// throw an integer
throw 1;

// throw a char *
throw "catastrophic error";
```

For most cases, you want to throw exception objects because they can provide a more complete description of an error. Further, throwing pointers is considered poor practice. It is usually a good idea to catch exceptions by reference, and especially by const reference. Objects that are caught by value must be copied before they are assigned to the catch parameter. If a user supplies a copy constructor, this is called, and this can add inefficiency.

Writing exception handlers

The exception handling block appears immediately after the try block. This block incudes one or more exception handlers. An exception handler provides a specific response to a specific kind of exception. Handling an exception clears the error condition and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as “File not found,” are easy to correct and retry, while others, such as running out of memory, can be more difficult for the application or the user to correct.

The application executes the statements in and exception handler only if an exception occurs during execution of the statements in the preceding try block. When a statement in the try block raises an exception, execution immediately jumps to the exception handler, where it steps through the specified exception-handling statements, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

D

Exception-handling statements in Delphi

In Delphi code, the exception handling block starts with the except keyword. and ends with the keyword end. These two keywords are actually part of the same statement as the try block. That is, in Delphi, both the try block and the exception handling block are considered part of a single try...except statement.

Inside the exception handling block, you include one or more exception handlers. An exception handler is a statement of the form

```
on <type of exception> do <statement>;
```

For example, the following exception handling block includes multiple exception handlers for different exceptions that can arise from an arithmetic computation:

```
try
  { calculation statements }
except
  on EZeroDivide do Value := MAXINT;
  on EIntOverflow do Value := 0;
  on EIntUnderflow do Value := 0;
end;
```

Much of the time, as in the previous example, the exception handler doesn't need any information about an exception other than its type, so the statements following on..do are specific only to the type of exception. In some cases, however, you might need some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of on..do that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance. For example:

```
on E: EIntegerRange do
  ShowMessage(Format('Expected value between %d and %d', E.Min, E.Max));
```

The temporary variable (E in this example) is of the type specified after the colon (*EIntegerRange* in this example). You can use the as operator to typecast the exception into a more specific type if needed.

Warming Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating an access violation.

You can provide a single default exception handler to handle any exceptions for which you haven't provided specific handlers. To do that, add an else part to the exception-handling block:

```
try
  { statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from any containing block.

Warning It is not advisable to use this all-encompassing default exception handler. The else clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more information about the exception and how to handle it, then you can do so using a finally block. For details about finally blocks, see “Writing finally blocks” on page 13-11.

Exception-handling statements in C++

In C++, the exception handling block consists of one or more catch statements. For example:

```
try
  CommitChange(dataBase, recordMods);
catch (const EIntegerRange &rangeErr)
  printf("Got an integer range exception");
catch (const EFileError &fileErr)
  printf("Got a file I/O error");
```

The catch statement has several forms. Objects may be caught by value, reference, or pointer:

```
catch (const EValueException ByVal) // by value
  DoSomething(ByVal.value);
catch (const ERefException &ByRef) // by reference
  DoSomethingElse(ByRef.value);
catch (const EPointerException *EPoint) // by pointer
  DoAnotherThing(EPoint->value);
```

In addition, the const modifier can be applied to the catch parameter, as it has in the previous examples. The const modifier acts on catch parameters the same way it modifies function parameters. That is, if the const modifier appears, the exception handler can't modify the exception it catches.

If you want your handler to catch all exceptions that might be thrown past the try block, you use the special form catch(…). This tells the exception handling system that the handler should be invoked for any exception. For example:

```
try
   SetFieldValue(dataField, userValue);
catch (...)
   printf("Got an exception of some kind");
```

Warning It is not advisable to use this all-encompassing default exception handler, because it handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more information about the exception and how to handle it, then you can do so using a finally block. For details about finally blocks, see "Writing finally blocks" on page 13-11.

Handling classes of exceptions

Exceptions are always represented by classes in Delphi, and usually by classes in C++. As such, you usually work with a hierarchy of exception classes. For example, CLX defines the *ERangeError* exception as a descendant of *EIntError*.

When you provide an exception handler for a base exception class, it catches not only direct instances of that class, but instances of any of its descendants as well. For example, the following exception handler handles all integer math exceptions, including *ERangeError*, *EDivByZero*, and and *EIntOverflow*:

Delphi example

```
try
```

```
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

C++ example

```
try
{
  // statements that perform integer math operations
}
catch (EIntError &E)
{
  // special handling for integer math errors
}
```

You can combine error handlers for the base class with specific handlers for more specific (derived) exceptions. You do this by placing the catch statements in the order that you want them to be searched when an exception is thrown. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

Delphi example

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

C++ example

```
try
{
  // statements performing integer math
}
catch (const ERangeError &rangeErr)
{
  // out-of-range handling
}
catch (const EIntError &intErr)
{
  // handling for other integer math errors
}
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler for *ERangeError*.

Scope of exception handlers

You do not need to provide handlers for every kind of exception in every block. You only need handlers for exceptions that you want to handle specially within a particular block.

If a block does not handle a particular exception, execution leaves that block and returns to the block that contains it (or returns to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception.

Thus, you can nest your exception handling code. That is, you can use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. For example:

Delphi example

```
try
  { statements }
  try
    { special statements }
  except
    on ESomething do
    begin
      { handling for only the special statements }
    end;
  end;
  { more statements }
except
  on ESomething do
  begin
    {handling for statements and more statements, but not special statements}
  end;
end;
```

C++ example

```
try
{
  // statements
  try
  {
    // special statements
  }
  catch (const ESomething &E)
  {
    // handling for only the special statements;
  }
  // more statements
}
catch (const ESomething &E)
{
  // handling for statements and more statements, but not special statements
}
```

Note This type of nesting is not limited to exception-handling blocks. You can also use it with finally blocks (described in "Writing finally blocks" on page 13-11) or a mix of exception-handling and finally blocks.

Reraising exceptions

Sometimes when you handle an exception locally, you want to augment the handling in the enclosing block, rather than replace it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond. You do this by using the raise (Delphi) or throw (C++) command with no arguments. This is called reraising or rethrowing the exception. The following example illustrates this technique:

Delphi example

```
try
  { statements }
  try
    { special statements }
  except
    on ESomething do
    begin
      { handling for only the special statements }
      raise;{ reraise the exception }
    end;
  end;
except
  on ESomething do ...;{ handling you want in all cases }
end;
```

C++ example

```
try
{
  // statements
  try
  {
    // special statements
  }
  catch (const ESomething &E)
  {
    // handling for only the special statements;
  }
}
catch (const ESomething &E)
{
  // handling you want in all cases
}
```

If code in the *statements* part raises an *ESomething* exception, only the handler in the outer exception-handling block executes. However, if code in the *special statements* part raises *ESomething*, the handling in the inner exception-handling block executes, followed by the more general handling in the outer exception-handling block. By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

If the handler wants to throw a different exception, it can use the raise or throw statement in the normal way, as described in "Raising an exception in Delphi" on page 13-4 and "Raising an exception in C++" on page 13-4.

Writing finally blocks

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. However, there are times when you do not need to handle the exception, but you do have code that you want to execute after the protected block, even if an exception occurs. Typically, such code handles cleanup issues, such as freeing resources that were allocated before the protected block.

By using finally blocks, you can ensure that if your application allocates resources, it also releases them, even if an exception occurs. Thus, if your application allocates memory, you can make sure it eventually releases the memory, too. If it opens a file, you can make sure it closes the file later. Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are

- Files
- Memory
- widget library resources (Qt objects)
- Objects (instances of classes in your application)

The following event handler illustrates how an exception can prevent an application from freeing memory that it allocates:

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an exception }
  FreeMem(APointer, 1024);{ this never gets called because of the exception}
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  int ADividend = 0;
  void *ptr = malloc(1024); // allocate 1K of memory;
  int AnInteger = 10/ADividend; // this generates an exception
  free(ptr); // this never gets called because of the exception
}
```

Although most errors are not that obvious, the example illustrates an important point: When an exception occurs, execution jumps out of the block, so the statement that frees the memory never gets called.

To ensure that the memory is freed, you can use a try block with a finally block.

Writing a finally block in Delphi

In Delphi, finally blocks are introduced by the keyword finally. They are part of a try..finally statement, which has the following form:

```
try
  { statements that may raise an exception}
finally
  { statements that are called even if there is an exception in the try block}
end;
```

In a try..finally statement, the application always executes any statements in the finally part , even if an exception occurs in the try block. When any code in the try block (or any routine called by code in the try block) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the finally part, which is called the cleanup code. After the finally part executes, the exception handler is called. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the try block.

The following code illustrates an event handler that uses a finally block so that when it allocates memory and generates an error, it still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;

begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an exception }
  finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the exception }
  end;
end;
```

The statements in the finally block do not depend on an exception occurring. If no statement in the try part raises an exception, execution continues through the finally block.

Writing a finally block in C++

The Borland compiler includes extensions to the C++ language that let it use finally blocks as well. Like exception handlers, a finally block must appear directly after the try block, but it is introduced by the __finally keyword instead of the keyword catch :

```
try
{
  // statements that may raise an exception
}
__finally
{
// statements that are called even if there is an exception in the try block
}
```

The application always executes any statements in the finally part , even if an exception occurs in the try block. When any code in the try block (or any routine called by code in the try block) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the finally part. After the finally part executes, the exception handler is called. If no exception occurs, the code in the finally block executes in the normal order, after all the statements in the try block.

The following code illustrates an event handler that uses a finally block so that when it allocates memory and generates an error, it still frees the allocated memory:

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  int ADividend = 0;
  void *ptr = malloc(1024); // allocate 1K of memory;
  try
    int AnInteger = 10/ADividend; // this generates an exception
  __finally
    free(ptr); // this gets called anyway, despite the exception
}
```

The statements in the finally block do not depend on an exception occurring. If no statement in the try part raises an exception, execution continues through the finally block.

Note Traditional C++ code does not include support for finally block. Instead, it tends to use destructors to handle the freeing of resources. However, when working with CLX, which is written in Delphi, finally blocks are an important tool because of the way CLX objects must be allocated on the heap.

Handling exceptions in CLX applications

If you use CLX components or the CLX runtime libarary in your applications, you need to understand the CLX exception handling mechanism. Exceptions are built into many CLX classes and routines and they are thrown automatically when something unexpected occurs. Typically, these exceptions indicate programming errors that would otherwise generate a runtime error.

The mechanics of handling component exceptions are no different than handling any other type of exception, although there are some special considerations if you are writing your application in C++. These are described in "Handling CLX exceptions in C++" on page 13-18.

If you do not handle the exception, CLX handles it in a default manner. Typically, a message displays describing the type of error that occurred. While debugging your application, you can look up the exception class in online Help. The information provided will often help you to determine where the error occurred and its cause.

A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises a "List index out of bounds" exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('a string');{ add a string to list box }
  ListBox1.Items.Add('another string');{ add another string... }
  ListBox1.Items.Add('still another string');{ ...and a third string }
  try
    Caption := WideString(ListBox1.Items[3]);{ set form caption to fourth string }
  except
    on EStringListError do
      ShowMessage('List box contains fewer than four strings');
  end;
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  ListBox1->Items->Add("a string"); // add a string to list box
  ListBox1->Items->Add("another string"); // add another string ...
  ListBox1->Items->Add("still another string"); // ... and a third string
  try
    Caption = WideString(ListBox1->Items->Strings[3]); // set form caption to 4th string
  catch (EStringListError &E)
    ShowMessage(L"List box contains fewer than four strings");
}
```

If you click the button once, the list box has only three strings, so accessing the fourth string raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

CLX exception classes

CLX includes a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions. All CLX exception classes descend from one root object called *Exception*. *Exception* provides a consistent interface for applications to handle exceptions. It provides the string for the message that CLX exceptions display by default.

Table 13.1 lists a selection of the exception classes defined in CLX:

Table 13.1 Selected exception classes

Exception class	Description
EAbort	Stops a sequence of events without displaying an error message dialog box.
EAccessViolation	Checks for invalid memory access errors.
EBitsError	Prevents invalid attempts to access a Boolean array.
EComponentError	Signals an invalid attempt to register or rename a component.
EConvertError	Indicates string or object conversion errors.
EDatabaseError	Specifies a database access error.
EDBEditError	Catches data incompatible with a specified mask.
EDivByZero	Catches integer divide-by-zero errors.
EExternalException	Signifies an unrecognized exception code.
EInOutError	Represents a file I/O error.
EIntOverflow	Specifies integer calculations whose results are too large for the allocated register.
EInvalidCast	Checks for illegal typecasting.
EInvalidGraphic	Indicates an attempt to work with an unrecognized graphic file format.
EInvalidOperation	Occurs when invalid operations are attempted on a component.
EInvalidPointer	Results from invalid pointer operations.
EMenuError	Involves a problem with menu item.
EOleCtrlError	Detects problems with linking to ActiveX controls.
EOleError	Specifies OLE automation errors.
EPrinterError	Signals a printing error.
EPropertyError	Occurs on unsuccessful attempts to set the value of a property.
ERangeError	Indicates an integer value that is too large for the declared type to which it is assigned.
ERegistryException	Specifies registry errors.
EZeroDivide	Catches floating-point divide-by-zero errors.

There are other times when you will need to create your own exception classes to handle unique situations. You can declare a new exception class by making it a descendant of type *Exception* and creating as many constructors as you need (or copy the constructors from an existing class in the Sysutils unit).

Default exception handling in CLX

If your application code does not catch and handle the exceptions that are raised, the exceptions are ultimately caught and handled by the the *HandleException* method of the global *Application* object. For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. If there is no *OnException* event handler (and the exception is not *EAbort*), *HandleException* displays a message box with the error message associated with the exception.

There are certain circumstances where *HandleException* does not get called. Exceptions that occur before or after the execution of the application's Run method are not caught and handled by *HandleException*. When you write a callback function or a library (shared object) with functions that can be called by an external application, exceptions can escape the *Application* object. To prevent exceptions from escaping in this manner, you cn insert your own call to the *HandleException* method:

Delphi example

```
try
    { special statements }
except
  on Exception do
  begin
    Application.HandleException(Self);{ call HandleException }
  end;
end;
```

C++ example

```
try
{
  // special statements
}
catch (Exception &E)
{
  Application->HandleException(this);
}
```

Warning Do not call *HandleException* from a thread's exception handling code.

Silent exceptions

CLX applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to report an exception to the user, but you want to abort an operation. Aborting an operation is similar to using the *Break* or *Exit* procedures (Delphi) or the break or return commands (C++) to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type *EAbort*. The default exception handler for CLX applications displays the error-message dialog box for all exceptions that reach it except those descended from *EAbort*.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the *Abort* procedure. *Abort* automatically raises an *EAbort* exception, which breaks out of the current operation without displaying an error message.

Note There is a distinction between *Abort* and *abort. abort* kills the application.

The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's *OnClick* event:

Delphi example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I, J: Integer;
begin
  for I := 1 to 10 do{ loop ten times }
    for J := 1 to 10 do {loop ten times }
    begin
      ListBox1.Items.Add(IntToStr(I) + IntToStr(J));
      if I = 7 then Abort;{ abort after the 7th iteration of outer loop}
    end;
end;
```

C++ example

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  for (int i = 1; i <= 10; i++) // loop ten times
    for (int j = 1; j <= 10; j++) // loop ten times
    {
      ListBox1->Items->Add(IntToStr(i) + IntToStr(j));
      if (i == 7)
        Abort(); // abort after 7th iteration of outer loop
    }
}
```

Note that in this example, *Abort* causes the flow of execution to break out of both the inner and outer loops, not just the inner loop.

Defining your own CLX exceptions

Because CLX exceptions are classes, defining a new kind of exception is as simple as declaring a new class type. Although you can raise any object instance as an exception, the standard CLX exception handlers handle only exceptions that descend from *Exception*.

New exception classes should be derived from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by an exception handler specific to that exception, one of the standard handlers will handle it instead.

For example, consider the following declaration:

```
type
  EMyException = class(Exception);
```

```
class EMyException : public Exception
{
};
```

If you raise *EMyException* but don't provide a specific handler for it, a handler for *Exception* (or a default exception handler) will still handle it. Because the standard handling for *Exception* displays the name of the exception raised, you can see that it is your new exception that is raised.

Handling CLX exceptions in C++

Because CLX is written in the Delphi language, there are some special considerations for working with exceptions when you are writing your application in C++. The following topics describe some of these considerations.

Exceptions thrown from constructors

C++ destructors are only called for members and base classes that are fully constructed. CLX base class destructors are called even if the object or base class isn't fully constructed. These differences are described more fully in "Exceptions thrown from constructors" on page 14-18.

Catching and throwing exceptions

C++ exceptions can be caught by reference, pointer, or value. When using CLX exceptions (exceptions derived from *TObject*), you must obey the following rules:

- CLX exception classes may only be caught by pointer, if it is a software exception, or by reference (reference is preferred). An attempt to catch CLX exceptions by value results in a compile-time error. Hardware or operating system exceptions, such as *EAccessViolation*, should be caught by reference.
- CLX exceptions should be thrown with "by value" syntax.
- You cannot use throw to reraise an exception that was caught by CLX code.

Following is an example of how to throw and catch a CLX exception:

```
void __fastcall TForm1::ThrowException(TObject *Sender)
{
  try
  {
    throw Exception("An error has occurred"); // note it is thrown "by value"
  }
  catch(const Exception &E) // note it is caught by reference
  {
    ShowMessage(AnsiString(E.ClassName())+ E.Message);
  }
}
```

The throw statement in the previous example creates an instance of the *Exception* class and calls its constructor. All exceptions descended from *Exception* have a message that can be displayed, passed through constructors, and retrieved through the *Message* property.

Handling C++ operating system exceptions

Operating system exceptions include access violations, integer math errors, floating-point math errors, stack overflow, and *Ctrl+C* interrupts. These are handled in the C++ RTL and converted to CLX exception class objects before being dispatched to your application. You can then write C++ code that looks like this:

```
try
{
  char * p = 0;
  *p = 0;
}
// You should always catch by reference.
catch (const EAccessViolation &e)
{
  printf("You can't do that!\n");
}
```

Because operating system exceptions are converted to CLX exceptions, they should always be caught by reference.

Another important point about operating system exceptions is that you can't reraise an operating system exception once the catch frame has exited and have it caught by intervening CLX or operating system catch frames. That is, once an operating system exception is caught as a C++ exception, it cannot be rethrown as if it were an operating system exception or CLX exception unless you are in the catching stack frame.

Floating point and arithmetic exceptions

There is a risk in using fine-grained try, catch, and except blocks to trap floating point arithmetic errors. Because these operations do not reference memory, the compiler has more freedom in moving instructions around when optimizing code. In addition, future processors may mask floating point exceptions or the compiler might not check for floating point erors until late in the function body. For these reasons, you should avoid the following type of try/catch block:

```
void calc(float f)
if (something)
{
  // ...some code...
  try
    val=f/somenumber;
  catch (EZeroDivide & e)
    val=0.0;
  // ...some more code...
  }
....
}
```

Instead, if you want to trap exceptions such as divide-by-zero, move the code into a helper function and wrap the entire function body in a try/catch block:

```
float TryToDivide (float f,d)
{
  try {return f/d}
```

```
      catch=f/somenumber;
    catch (EZeroDivide & e)
  {return 0.0;}
    }
```

You can then replace the inline code with a call to the helper function.

```
val=TryToDivide (f, somenumber);
```

Portability considerations in C++

Several runtime libraries (RTLs) are delivered for your C++ applications. Most of them pertain to CLX applications, but one of them, cw32mt.lib, is the normal multi-threaded RTL that does not make any references to CLX. This RTL is provided for support of legacy applications that may be part of a project but do not depend on CLX. This RTL does not have support for catching operating system exceptions because those exception objects are derived from *TObject* and would require parts of CLX to be linked into your application.

You can use the cp32mt.lib library, a multi-threaded runtime library, which provides memory management and exception handling with CLX.

You can use two import libraries, cw32mti.lib and cp32mti.lib, for using the RTL DLL. Use cp32mti.lib for CLX exception support.

C++ exceptions in non-CLX code

When writing C++ code that does not use CLX, there are a number of features and considerations that do not apply when writing CLX-based code. The following topics describe these.

Exception specifications in C++

In C++, you can specify which exceptions a function may throw. It is a runtime error to throw an exception of the wrong type past a function. The syntax for an *exception specification* is:

```
exception-specification:
   throw (type-id-list)    //type-id-list is optional
   type-id-list:
      type-id
      type-id-list, type-id
```

The following examples are functions with exception specifications.

```
void f1();              // The function can throw any exception

void f2() throw();      // Should not throw any exceptions

void f3() throw( A, B* ); // Can throw exceptions publicly derived from A,
                        // or a pointer to publicly derived B
```

The definition and all declarations of such a function must have an exception specification containing the same set of *type-id*s. If a function throws an exception not listed in its specification, the program calls *unexpected*.

You may not want to specify an exception for the following reasons:

- You can get unexpected errors at runtime. For example, suppose your system uses exception specifications and uses another subsystem in its implementation. If the subsystem is changed so that it throws new exception types, when you use the new subsystem code, you could get runtime errors without ever getting an indication from the compiler that this might happen.
- If you use virtual functions, you can violate the program design. This is because the exception specification is not considered part of the function type. For example, in the following code, the derived class *BETA::vfunc* is defined so that it does not throw any exceptions—a departure from the original function declaration.

```
class ALPHA {
public:
  struct ALPHA_ERR {};
  virtual void vfunc(void) throw (ALPHA_ERR) {} // Exception specification
};

class BETA : public ALPHA {
  void vfunc(void) throw() {} // Exception specification is changed
};
```

Unwinding exceptions in C++

When an exception is thrown, the runtime library takes the thrown object, gets the type of the object, and looks upward in the call stack for a handler whose type matches the type of the thrown object. Once a handler is found, the RTL unwinds the stack to the point of the handler, and executes the handler.

In the unwind process, the RTL calls destructors for all local objects in the stack frames between where the exception was thrown and where it is caught. If a destructor causes an exception to be raised during stack unwinding and does not handle it, *terminate* is called. Destructors are called by default, but you can switch off the default by using the **-xd** compiler option.

Note During the unwind process, the RTL does not call destructors for objects that are allocated on the heap rather than the stack. This is why, for example, CLX applications use finally blocks to ensure that CLX objects, which are always allocated on the heap, are properly freed. There is one exception to this rule, which is the use of safe pointers.

Safe pointers in C++

If you have local variables that are pointers to objects and an exception is thrown, these pointers are not automatically deleted. This is because there is no good way for the compiler to distinguish between a pointer to data that was allocated for this function only and any other pointer. The class that you can use to ensure that objects allocated for local use are destroyed in the even of an exception is auto_ptr. There is a special case in which memory is freed for a pointer allocated in a function:

```
auto_ptr<TMyObject *>pMyObject(new TMyObject);
```

In this example, if the constructor for *TMyObject* throws an exception, then the pointer to the object allocated for *TMYObject* will be deleted by the RTL when it unwinds the exception. This is the only time that the compiler automatically deletes a pointer value for you.

Constructors in exception handling in C++

Class constructors can throw exceptions if they cannot successfully construct an object. If a constructor throws an exception, that object's destructor is not necessarily called. Destructors are called only for the base classes and for those objects that were fully constructed inside the classes since entering the try block.

Note This is not true for CLX base classes. For more information, see "Exceptions thrown from constructors" on page 14-18.

Handling uncaught and unexpected exceptions in C++

If an exception is thrown and no exception handler is found—that is, the exception is not caught—the program calls a termination function. You can specify your own termination function with *set_terminate*. If you do not specify a termination function, the *terminate* function is called. For example, the following code uses the *my_terminate* function to handle exceptions not caught by any handler.

```
void SetFieldValue(DF *dataField, int userValue)
{
  if ((userValue < 0) || (userValue) > 10));
    throw EIntegerRange(0, 10, userValue);
  . . .
}

void my_terminate()
{
  printf("Exception not caught");
  abort();
}

// Set my_terminate() as the termination function
set_terminate(my_terminate);
// Call SetFieldValue. This generates an exception because the user value is greater
// than 10. Because the call is not in a try block, my_terminate is called.
SetFieldValue(DF, 11);
```

If a function specifies which exceptions it throws and it throws an unspecified exception, an unexpected function is called. You can specify your own unexpected function with *set_unexpected*. If you do not specify an unexpected function, the *unexpected* function is called.

C++ exception handling options

Following are the exception handling options in the C++ compiler.

Table 13.2 C++ exception-handling compiler options

Command-line switch	Description
`-x`	Enable C++ exception handling (On by default)
`-xd`	Enable destructor cleanup. Calls destructors for all automatically declared objects between the scope of the catch and throw statements when an exception is thrown. (Advanced option–on by default)
`-xp`	Enable exception location information. Makes available runtime identification of exceptions by providing line numbers in the source code at the exception location. This lets the program query the file and line number where a C++ exception occurred using the __ThrowFileName and __ThrowLineNumber globals. (Advanced option)

Chapter 14

C++ language support for CLX

<~TOPIC<~HEAD
^#c++languagesupportforthevcl
^Ac++languagesupportforthevcl
^XTheC++BuilderDevelopmentEnvironment;CreatingApplications;VCLExceptionHandli
ng
^$ C++ language support for CLX
^+languagesupport:000
^K CLX, C++ and;Delphi
^T SUPP4VCL
^C 2 C++ language support for CLX;3
HEAD~>C++ language support for CLX

GalileoKylix leverages the Rapid Application Development (RAD) capabilities of the Component Library for Cross-Platform (CLX) written in the Delphi programming language. This chapter explains how Delphi language features, constructs, and concepts were implemented in the Borland extensions to C++ to support CLX. It is written for programmers using CLX objects in their applications and for developers creating new classes descended from CLX classes.

The first half of this chapter compares Delphi and C++ object models, describing how CLX combines these two approaches. The second half of the chapter describes how the Borland extensions to the C++ language accomodate Delphi language constructs. It includes details on C++ language keyword extensions that were added to support the CLX. Some of these extensions, like closures and properties, are useful features independent of their support for CLX-based code.

Note References to C++ classes derived from *TObject* refer to classes for which *TObject* is the ultimate, but not necessarily immediate, ancestor. For consistency with the compiler diagnostics, such classes are also referred to as “CLX-style classes.”

The following topics describe how Delphi language features, constructs, and concepts were implemented in the product to support CLX. The first topic compares C++ and Delphi object models, describing how CLX combines these two approaches:

- <~JMP C++ and Delphi object models~c++andobjectpascalmodels JMP~>

The second topic describes how Delphi language constructs were translated into C++ counterparts in GalileoKylix for C++:

- <~JMP Support for Delphi data types and language concepts~supportforobjectpascaldatatypesandlanguageconcepts JMP~>

These topics include details on keyword extensions that were added to support CLX. Some of these extensions, like closures and properties, are useful features independent of their support for CLX-based code.

TOPIC~>

C++ and Delphi object models

<~TOPIC<~HEAD
^#c++andobjectpascalmodels
^A c++andobjectpascalmodels
^Xc++languagesupportforthevcl
^$C++ and Delphi object models
^+languagesupport:000
^KDelphi;Delphi and C++;C++ and Delphi
^T SUPP4VCL
^C 3
HEAD~>C++ and Delphi object models

The C++ and Delphi class models are different in both obvious and subtle ways. One of the most obvious differences is that C++ allows multiple inheritance while Delphi is restricted to a single inheritance model. In addition, C++ and Delphi are subtly different in the way they create, initialize, reference, copy, and destroy objects. These differences and their impact on CLX-style classes are described in this sectionthe following topics.

- <~JMP Inheritance and interfaces~InheritanceAndVCLStyleClasses JMP~>
- <~JMP Object identity and instantiation~objectidentityandinstantiation JMP~>
- <~JMP Construction of CLX-style classes in C++~objectconstructionforc++buildervclclasses JMP~>
- <~JMP Calling virtual methods in base class constructors~callingvirtualmethodsinbaseclassconstructors JMP~>
- <~JMP Object destruction~objectdestruction JMP~>
- <~JMP AfterConstruction and BeforeDestruction~afterconstructionandbeforedestruction JMP~>
- <~JMP Class virtual functions~classvirtualfunctions JMP~>

TOPIC~>

Inheritance and interfaces

<~TOPIC<~HEAD
^#InheritanceAndVCLStyleClasses
^AInheritanceAndVCLStyleClasses;InheritanceAndInterfaces
^X DelphiInterfaces
^$Inheritance and interfaces
^+languagesupport:000
^K inheritance, restrictions;inheritance, multiple;interfaces;multiple inheritance
^T SUPP4VCL
^C 3
HEAD~>Inheritance and interfaces

Unlike C++, the Delphi language does not support multiple inheritance. Any classes that you create that have CLX ancestors inherit this restriction. That is, you can't use multiple base classes for a CLX-style C++ class, even if the CLX class is not the immediate ancestor.

To work around this limitation (partially), you can use interfaces. The following topics describe the use of interfaces with CLX-style classes:

- <~JMP Using interfaces instead of multiple inheritance~UsingInterfacesInsteadOfMultipleInheritance JMP~>
- <~JMP Declaring interface classes~DeclaringInterfaceclasses JMP~>
- <~JMP IUnknown and IInterface~IUnknownAndIInterface JMP~>
- <~JMP Creating classes that support IUnknown~CreatingClassesThatSupportIUnknown JMP~>
- <~JMP Interfaced classes and lifetime management~InterfaceClassesAndLifetimeManagement JMP~>

TOPIC~>

Using interfaces instead of multiple inheritance

<~TOPIC<~HEAD
^#UsingInterfacesInsteadOfMultipleInheritance
^AUsingInterfacesInsteadOfMultipleInheritance
^X DeclaringInterfaceClasses;InterfaceClassesAndLifetimeManagement
^$Using interfaces instead of multiple inheritance
^+languagesupport:000
^K interfaces, multiple inheritance and;multiple inheritance, interfaces and
^T SUPP4VCL
^C 3
HEAD~>Using interfaces instead of multiple inheritance
For many of the situations where you would use multiple inheritance in C++, Delphi code makes use of interfaces instead. There is no C++ construct that maps directly to the Delphi concept of interface. A Delphi interface acts like a class with no implementation. That is, an interface is like a class where all the methods are pure virtual and there are no data members. While a Delphi class can have only a single

parent class, it can support any number of interfaces. Delphi code can assign a class instance to variables of any of those interface types, just as it can assign the class instance to a variable of any ancestor class type. This allows polymorphic behavior for classes that share the same interface, even if they do not have a common ancestor.

The C++ compiler recognizes classes that have only pure virtual methods and no data members as corresponding to Delphi interfaces. Thus, when you create a CLX-style class, you are permitted to use multiple inheritance, but only if all of the base classes except the one that is a CLX or CLX-style class have no data members and only pure virtual methods.

Note The interface classes do not need to be CLX-style classes; the only requirement is that they have no data members and only pure virtual methods.

TOPIC~>

Declaring interface classes

<~TOPIC<~HEAD
^#DeclaringInterfaceClasses
^A DeclaringInterfaceClasses
^X
UsingInterfacesInsteadOfMultipleInheritance;DelphiInterfaces;thedeclspeckeyword extension
^$Declaring interface classes
^+languagesupport:000
^K interfaces, declaring;INTERFACE_UUID;__interface
^T SUPP4VCL
^C 3
HEAD~>Declaring interface classes
You can declare a class that represents an interface just like any other C++ class. However, by using certain conventions, you can make it clearer that the class is intended to act as an interface. These conventions are as follows:

- Instead of using the class keyword, interfaces are declared using the __interface macro. The __interface macro maps to the class keyword. It is not necessary, but makes it clearer that the class is intended to act as an interface.
- Interfaces typically have names that begin with the letter 'I.' Examples are *IComponentEditor* or *IDesigner*. By following this convention, you do not need to look back at the class declaration to realize when a class is acting as an interface.
- Interfaces typically have an associated GUID. This is not an absolute requirement, but most of the code that supports interfaces expects to find a GUID. You can use the __declspec modifier with the uuid argument to associate an interface with a GUID. For interfaces, the INTERFACE_UUID macro maps to the same thing.

In C++, the following interface declaration illustrates these conventions:

```
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}") IHelloWorld :
    public IInterface
{
public:
  virtual void __stdcall SayHelloWorld(void) = 0 ;
};
```

Typically, when declaring an interface class, C++ code also declares a corresponding <~JMP *DelphiInterface* class~DelphiInterfaces JMP~> that makes working with the interface more convenient:

```
typedef System::DelphiInterface< IHelloWorld >  _di_IHelloWorld;
```

For information about the DelphiInterface class, see “Delphi interfaces” on page 14-30.

TOPIC~>

IUnknown and IInterface

<~TOPIC<~HEAD
^# IUnknownAndIInterface
^AIUnknownAndIInterface
^X
UsingInterfacesInsteadOfMultipleInheritance;DelphiInterfaces;DeclaringInterfaceCl
asses
^$IUnknown and IInterface
^+languagesupport:000
^K IUnknown interface;IInterface interface
^T SUPP4VCL
^C 3
HEAD~>IUnknown and IInterface
All Delphi interfaces descend from a single common ancestor, *IInterface*. It is not necessary for C++ interface classes to use *IInterface* as a base class in the sense that a CLX-style class can use them as additional base classes, but CLX code that deals with interfaces assume that the *IInterface* methods are present.

In Delphi, *IInterface* maps directly to another interface, *IUnknown*, which is the basis for all interfaces in COM.

The Delphi definition of *IUnknown*, however, does not correspond to the definition of *IUnknown* that is used in the Borland extensions to C++. The file unknwn.h defines *IUnknown* to include the following three methods:

```
virtual HRESULT STDCALL QueryInterface( const GUID &guid, void ** ppv ) = 0;
virtual ULONG STDCALL AddRef() = 0;
virtual ULONG STDCALL Release() = 0;
```

This corresponds to the definition of *IUnknown* that is defined by Microsoft as part of the COM specification.

Note For information on *IUnknown* and its use, see Chapter 37, “Overview of COM technologies”<~JMP Overview of COM technologies~!al(oocOverviewOfCOMTechnologies,1) JMP~> or the Microsoft documentation.

Unlike the case in Delphi, the C++ definition of *IInterface* is not equivalent to the C++ definition of *IUnknown*. Instead, in C++, *IInterface* is a descendant of *IUnknown* that includes an additional method, *Supports*:

```
template <typename T>
HRESULT __stdcall Supports(DelphiInterface<T>& smartIntf)
{
  return QueryInterface(__uuidof(T),
         reinterpret_cast<void**>(static_cast<T**>(&smartIntf)));
}
```

Supports lets you obtain a *DelphiInterface* for another supported interface from the object that implements *IInterface*. Thus, for example, if you have a *DelphiInterface* for an interface *IMyFirstInterface*, and the implementing object also implements *IMySecondInterface* (whose *DelphiInterface* type is *_di_IMySecondInterface*), you can obtain the *DelphiInterface* for the second interface as follows:

```
_di_IMySecondInterface MySecondIntf;
MyFirstIntf->Supports(MySecondIntf);
```

CLX uses a mapping of *IUnknown* into Delphi. This mapping converts the types used in the *IUnknown* methods to Delphi types, and renames the *AddRef* and *Release* methods to *_AddRef* and *_Release* in order to indicate that they should never be called directly. (In Delphi, the compiler automatically generates the necessary calls to *IUnknown* methods.) When the *IUnknown* (or *IInterface*) methods that CLX objects support are mapped back into C++, this results in the following method signatures:

```
virtual HRESULT __stdcall QueryInterface(const GUID &IID, void *Obj);
int __stdcall _AddRef(void);
int __stdcall _Release(void);
```

This means that CLX objects that support *IUnknown* or *IInterface* in Delphi do not support the versions of *IUnknown* and *IInterface* that appear in the C++ header files.

TOPIC~>

Creating classes that support IUnknown

<~TOPIC<~HEAD
^# CreatingClassesThatSupportIUnknown
^A CreatingClassesThatSupportIUnknown
^X IUnknownAndIInterface;InterfaceClassesAndLifetimeManagement
^$Creating classes that support IUnknown
^+languagesupport:000
^K IUnknown, implementing;IInterface, implementing;TComponent, interfaces and;TInterfacedObject,
^T SUPP4VCL
^C 3
HEAD~>Creating classes that support IUnknown
Many CLX classes include interface support. In fact, the base class for all CLX-style classes, *TObject*, has a *GetInterface* method that lets you obtain a *DelphiInterface* class for any interface that an object instance supports. *TComponent* and *TInterfacedObject* both implement the *IInterface* (*IUnknown*) methods. Thus, any descendant of *TComponent* or *TInterfacedObject* that you create automatically inherits support for the common methods of all Delphi interfaces. In Delphi, when you create a descendant of either of these classes, you can support a new interface by implementing only

those methods introduced by the new interface, relying on a default implementation for the inherited *IInterface* methods.

Unfortunately, because the signatures of the <~JMP *IUnknown* and *IInterface* methods~IUnknownAndIInterface JMP~> differ between their C++ declarations and the versions used in CLX classes, CLX-style classes that you create do not automatically include *IInterface* or *IUnknown* support, even if they are derived (directly or indirectly) from *TComponent* or *TInterfacedObject*. That is, to support any interfaces you define in C++ that descend from *IUnknown* or *IInterface*, you must still add an implementation of the *IUnknown* methods.

The easiest way to implement the *IUnknown* methods in a class that descends from *TComponent* or *TInterfacedObject* is to take advantage of the built-in *IUnknown* support, in spite of the differences in function signatures. Simply add implementations for the C++ versions of the *IUnknown* methods, delegating to the inherited Delphi-based versions. For example:

```
virtual HRESULT __stdcall QueryInterface(const GUID& IID, void **Obj)
{
   return TInterfacedObject::QueryInterface(IID, (void *)Obj);
}
virtual ULONG __stdcall AddRef()
{
   return TInterfacedObject::_AddRef();
}
virtual ULONG __stdcall Release()
{
   return TInterfacedObject::_Release();
}
```

By adding the previous three method implementations to a descendant of *TInterfacedObject*, your class obtains full *IUnknown* or *IInterface* support.

TOPIC~>

Interfaced classes and lifetime management

<~TOPIC<~HEAD
^# InterfaceClassesAndLifetimeManagement
^A InterfaceClassesAndLifetimeManagement
^X IUnknownAndIInterface;CreatingClassesThatSupportIUnknown
^$Interfaced classes and lifetime management
^+languagesupport:000
^K IUnknown, lifetime management;IInterface, lifetime management;TComponent, interfaces and;TInterfacedObject,
^T SUPP4VCL
^C 3
HEAD~>Interfaced classes and lifetime management
The *IUnknown* methods of interface classes have implications for the way you allocate and free the objects that implement the interface. When *IUnknown* is implemented by a COM object, the object uses the *IUnknown* methods to keep track of how many references to its interface are in use. When that reference count drops to

zero, the object automatically frees itself. *TInterfacedObject* implements the *IUnknown* methods to perform this same type of lifetime management.

This interpretation of the *IUnknown* methods, however, is not strictly necessary. The default implementation of the *IUnknown* methods provided by *TComponent*, for example, ignores the reference count on the interface, so that the component does not free itself when its reference count drops to zero. This is because *TComponent* relies on the object specified by its *Owner* property to free it.

Some components use a hybrid of these two models. If their *Owner* property is NULL, they use the reference count on their interface for lifetime management, and free themselves when that reference count drops to zero. If they have an owner, they ignore the reference counting and allow the owner to free them. Note that for such hybrid objects, as well as for any other objects that use reference counting for lifetime management, the objects are not automatically freed if the application creates the object but never obtains an interface from it.

TOPIC~>

Object identity and instantiation

<~TOPIC<~HEAD
^#objectidentityandinstantiation
^Aobjectidentityandinstantiation
^Xc++andobjectpascalmodels
^$Object identity and instantiation
^+languagesupport:000
^Kobject identity and instantiation;instantiation and identity of objects
^T SUPP4VCL
^C 3
HEAD~>Object identity and instantiation

In C++, an instance of a class is an actual object. That object can be directly manipulated, or it can be accessed indirectly through either a reference or a pointer to it. For example, given a C++ class *CPP_class* with a constructor that takes no arguments, the following are all valid instance variables of that class:

```
CPP_class by_value;// an object of type CPP_class
CPP_class& ref = by_value;// a reference to the object by_value, above
CPP_class* ptr = new CPP_class();// a pointer to an object of type CPP_class
```

By contrast, in Delphi a variable of type *object* always refers to the object indirectly. The memory for all objects is dynamically allocated. For example, given a Delphi class *D_class*

```
ref: D_class;
ref := D_class.Create;
```

ref is a “reference” to an object of type *D_class*. Translated to C++ code, it would be

```
D_class* ref = new D_class;
```

The next few sections discuss the implications of these differences.

Distinguishing C++ and Delphi references

Documentation frequently refers to a Delphi class instance variable as a reference, but describes its behavior as that of a pointer. This is because it has properties of both. A Delphi object reference is like a C++ pointer with the following exceptions:

- A Delphi reference is implicitly dereferenced (in which case it acts more like a C++ reference).
- A Delphi reference does not have pointer arithmetic as a defined operation.

When comparing a Delphi reference with a C++ reference, there are also similarities and differences. References in both languages are implicitly dereferenced, however,

- A Delphi reference can be rebound, whereas a C++ reference cannot.
- A Delphi reference can be nil, whereas a C++ reference must refer to a valid object.

Some of the design decisions underlying the CLX framework are based upon the use of this type of instance variable. A pointer is the closest C++ language construct to a Delphi reference. Consequently, nearly all CLX object identifiers are translated into C++ pointers.

Note The Delphi <~JMP var parameter~c++languagecounterpartstotheobjectpascallanguage JMP~> type is a close match to a C++ reference. For more information about **var** parameters, see “Var parameters” on page 14-23.

Copying objects

Unlike C++, Delphi does not have built-in compiler support for making a copy of an object. This section describes the impact of this difference on assignment operators and copy constructors for CLX-style classes.

Assignment operators

The Delphi assignment operator (:=) is not a class assignment operator (operator=()). The assignment operator copies the reference, not the object. In the following code, *B* and *C* both refer to the same object:

```
B, C: TButton;
B:= TButton.Create(ownerCtrl);
C:= B;
```

This example translates to the following code in C++:

```
TButton *B = NULL;
TButton *C = NULL;
B = new TButton(ownerCtrl);
C = B;// makes a copy of the pointer, not the object
```

CLX-style classes in C++ follow the Delphi language rules for assignment operators. This means that, in the following code, assignments between dereferenced pointers are not valid because they attempt to copy the object, not the pointer:

```
TCLXStyleClass *p = new TCLXStyleClass;
TCLXStyleClass *q = new TCLXStyleClass;
*p = *q;// not allowed for CLX-style classes
```

Note For CLX-style classes, it is still valid to use C++ syntax to bind a reference. For example, the following code is valid:

```
TCLXStyleClass *ptr = new TCLXStyleClass;
TCLXStyleClass &ref = *ptr;// OK for CLX-style classes
```

Although this is not the same as using an assignment operator, the syntax is similar enough that it is mentioned here for clarification and comparison.

Copy constructors

Delphi does not have built-in copy constructors. Consequently, CLX-style classes in C++ do not have built-in copy constructors. The following example code attempts to create a *TButton* pointer by using a copy constructor:

```
TButton *B = new TButton(ownerCtrl);
TButton *C = new TButton(*B);// not allowed for CLX-style classes
```

Do not write code that depends upon a built-in copy constructor for CLX classes. To create a copy of a CLX-style class object in C++, you can write code for a member function that copies the object. Alternatively, descendants of the CLX *TPersistent* class can override the *Assign* method to copy data from one object to another. This is typically done, for example, in graphics classes such as *TBitmap* and *TIcon* that contain resource images. Ultimately, the manner of copying an object can be determined by the programmer (component writer); but be aware that some of the copy methods used in standard C++ are not available for CLX-style classes.

Objects as function arguments

As discussed previously, instance variables in C++ and in Delphi are not identical. You should be aware of this when passing objects as arguments to functions. In C++, objects can be passed to functions either by value, by reference, or by pointer. In Delphi, when an object is passed by value to a function, remember that the object argument is already a reference to an object. So, it is, in fact, the reference that is passed by value, not the actual object. There is no Delphi equivalent to passing an actual object by value as in C++. CLX-style objects passed to functions follow the Delphi rules.

TOPIC~>

Construction of CLX-style objects in C++

<~TOPIC<~HEAD
^#objectconstructionforC++BuilderVCLclasses
^AobjectconstructionforC++BuilderVCLclasses
^Xc++languagesupportforthevcl;callingvirtualmethodsinbaseclassconstructors
^$Construction of CLX-style objects in C++
^+languagesupport:000
^Kobject construction
^T SUPP4VCL
^C 3
HEAD~>Construction of CLX-style objects in C++

C++ and Delphi construct objects differently. This section is an overview of this topic and a description of how CLX combines these two approaches.

Standard C++ object construction

In standard C++, the order of construction is virtual base classes, followed by base classes, and finally the derived class. The C++ syntax uses the constructor initialization list to call base class constructors. The runtime type of the object is that of the class of the current constructor being called. Virtual method dispatching follows the runtime type of the object and changes accordingly during construction.

Delphi object construction

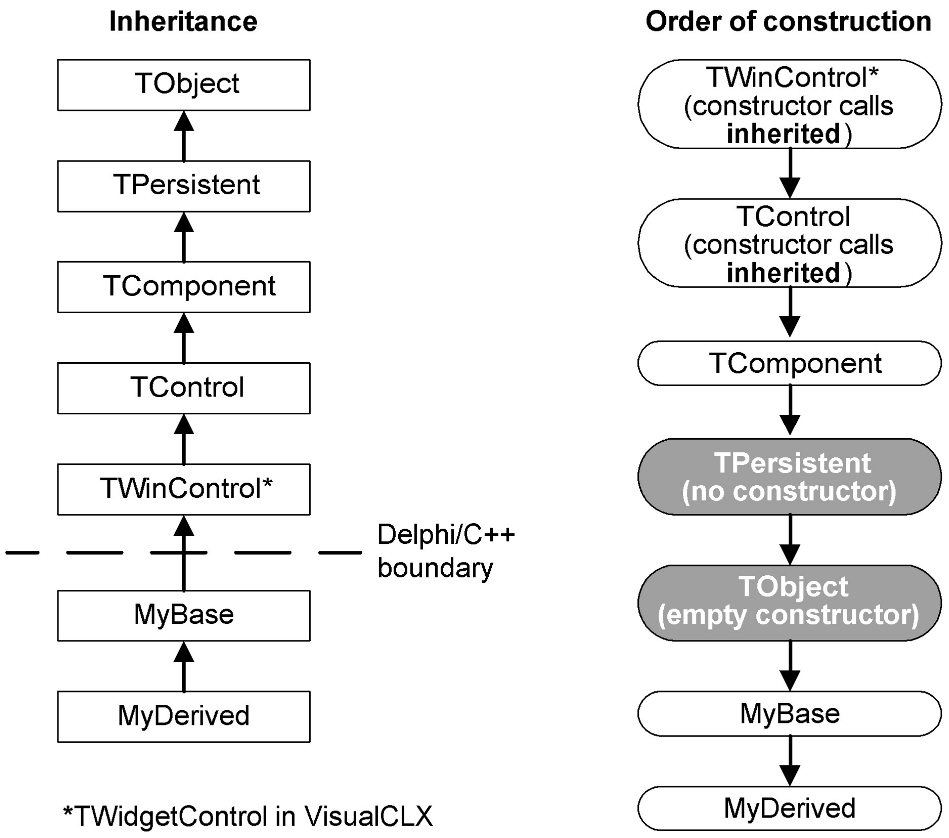
In Delphi, only the constructor for the instantiated class is guaranteed to be called; however, the memory for base classes is allocated. Constructing each immediate base class is done by calling inherited in the corresponding derived class's constructor. By convention, CLX classes use inherited to call (non-empty) base class constructors. Be aware, however, that this is not a requirement of the language. The runtime type of the object is established immediately as that of the instantiated class and does not change as base class constructors are called. Virtual method dispatching follows the runtime type of the object and, therefore, does not change during construction.

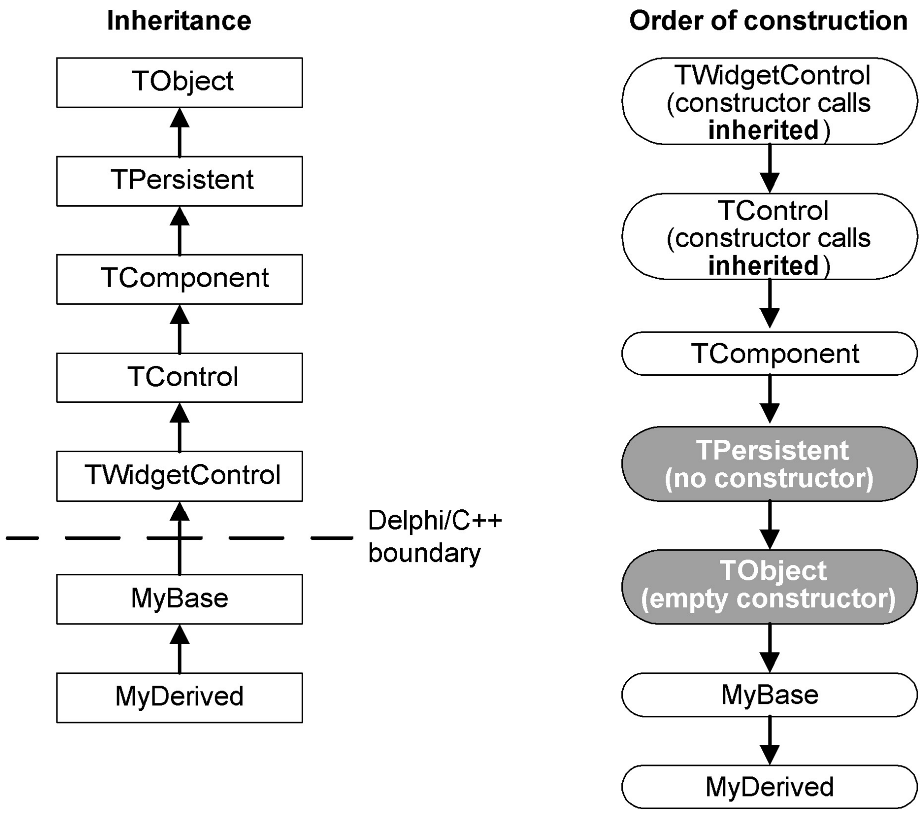
CLX-style object construction

CLX-style objects are constructed like Delphi objects, but using C++ syntax. This means that the method and order of calling base class constructors follows C++ syntax using the initialization list for all non-CLX base classes and the first immediate CLX ancestor. This CLX base class is the first class to be constructed. It constructs its own base class, optionally, using inherited, following the Delphi method. Therefore, the CLX base classes are constructed in the opposite order from which you might expect in C++. Then the C++ base classes are all constructed, from the most distant ancestor to the derived class. The runtime type of the object and virtual method dispatching are Delphi-based.

Figure 14.1The figure below illustrates the construction of an instance of a CLX-style class, *MyDerived*, descended from *MyBase*, which is a direct descendant of *TWinWidgetControl*. *MyDerived* and *MyBase* are implemented in C++. *TWinWidgetControl* is a CLX class implemented in Delphi.

Figure 14.1 Order of CLX-style object construction


Inheritance
TObject
TPersistent
TComponent
TControl
TWinControl*
Delphi/C++ boundary
MyBase
MyDerived
*TWidgetControl in VisualCLX
Order of construction
TWinControl* (constructor calls inherited)
TControl (constructor calls inherited)
TComponent
TPersistent (no constructor)
TObject (empty constructor)
MyBase
MyDerived



{bmc tcall11c.bmp}

{bmc tcall11k.bmp}

Note that the order of construction might seem backwards to a C++ programmer because it starts from the leaf-most ancestor to *TObject* for CLX classes, then constructs *MyBase,* and finally constructs the derived class.

Note *TComponent* does not call inherited because *TPersistent* does not have a constructor. *TObject* has an empty constructor, so it is not called. If these class constructors were called, the order would follow the diagram in which these classes appear in gray.

The object construction models for standard C++, Delphi, and CLX-style classes are summarized in Table 14.1the following table:

Table 14.1 Object model comparison

Standard C++	Delphi	CLX-style
Order of construction		
Virtual base classes, then base classes, finally the derived class.	Instantiated class constructor is the first and only constructor to be called automatically. If subsequent classes are constructed, they are constructed from leaf-most to root.	Most immediate CLX base class, then construction follows the Delphi model, then construction follows the C++ model (except that no virtual base classes are allowed).
Method of calling base class constructors		
Automatically, from the constructor initialization list.	Optionally, explicitly, and at any time during the body of the derived class constructor, by using the **inherited** keyword.	Automatically from the constructor initialization list through the most immediate ancestor that is a CLX base class constructor. Then according to the Delphi method, calling constructors with **inherited**.
Runtime type of the object as it is being constructed		
Changes, reflecting the type of the current constructor class.	Established immediately as that of the instantiated class.	Established immediately as that of the instantiated class.
Virtual method dispatching		
Changes according to the runtime type of the object as base class constructors are called.	Follows the runtime type of the object, which remains the same throughout calls to all class constructors.	Follows the runtime type of the object, which remains the same throughout calls to all class constructors.

The following section<~JMP Calling virtual methods in base class constructors~callingvirtualmethodsinbaseclassconstructors JMP~> describes the significance of these differences.

TOPIC~>

Calling virtual methods in base class constructors

<~TOPIC<~HEAD
^#callingvirtualmethodsinbaseclassconstructors
^Acallingvirtualmethodsinbaseclassconstructors
^Xc++languagesupportforthevcl;objectconstructionforC++BuilderVCLclasses;constructorinitializationofdatamembersforvirtualfunctions
^$Calling virtual methods in base class constructors
^+languagesupport:000
^Kobject construction;constructing objects;creating objects;virtual methods
^T SUPP4VCL

^C 3
HEAD~>Calling virtual methods in base class constructors

Virtual methods invoked from the body of CLX base class constructors, that is, classes implemented in Delphi, are dispatched as in C++, according to the runtime type of the object. Because CLX combines the Delphi model of setting the runtime type of an object immediately, with the C++ model of constructing base classes before the derived class is constructed, calling virtual methods from base class constructors for CLX-style classes can have subtle side-effects. The impact of this is described below, then illustrated in an example<~JMP example~examplecallingvirtualmethods JMP~> of an instantiated class that is derived from at least one base. In this discussion, the instantiated class is referred to as the derived class.

Standard C++ model

The C++ syntax does not have the **i**nherited keyword to call the base class constructor at any time during the construction of the derived class. For the C++ model, the use of inherited is not necessary because the runtime type of the object is that of the current class being constructed, not the derived class. Therefore, the virtual methods invoked are those of the current class, not the derived class. Consequently, it is not necessary to initialize data members or set up the derived class object before these methods are called.

Delphi model

In Delphi, programmers can use the inherited keyword, which provides flexibility for calling base class constructors anywhere in the body of a derived class constructor. Consequently, if the derived class overrides any virtual methods that depend upon setting up the object or initializing data members, this can happen before the base class constructor is called and the virtual methods are invoked.

CLX-style model

CLX-style objects have the runtime type of the derived class throughout calls to base class constructors. Therefore, if the base class constructor calls a virtual method, the derived class method is invoked if the derived class overrides it. If this virtual method depends upon anything in the initialization list or body of the derived class constructor, the method is called before this happens. For example, *CreateParams* is a virtual member function that is called indirectly in the constructor of *TWinWidgetControl*. If you derive a class from *TWinWidgetControl* and override *CreateParams* so that it depends on anything in your constructor, this code is processed after *CreateParams* is called. This situation applies to any derived classes of a base. Consider a class *C* derived from *B*, which is derived from *A*. Creating an instance of *C*, *A* would also call the overridden method of *B*, if *B* overrides the method but *C* does not.

Note Be aware of virtual methods like *CreateParams* that are not obviously called in constructors, but are invoked indirectly.

- <~JMP Example: calling virtual methods~examplecallingvirtualmethods JMP~>

- <~JMP Constructor initialization of data members for virtual functions~constructorinitializationofdatamembersforvirtualfunctions JMP~>

TOPIC~>

Example: calling virtual methods

<~TOPIC<~HEAD
^#examplecallingvirtualmethods
^Aexamplecallingvirtualmethods
^Xconstructorinitializationofdatamembersforvirtualfunctions;callingvirtualmethodsinbaseclassconstructors
^$Example: calling virtual methods
^+languagesupport:000
^Kobject construction;constructing objects;creating objects
^T SUPP4VCL
^C 3
HEAD~>Example: calling virtual methods
The following example compares standard C++ and CLX-style classes that have overridden virtual methods. This example illustrates how calls to those virtual methods from base class constructors are resolved in both cases. *MyBase* and *MyDerived* are standard C++ classes. *MyCLXBase* and *MyCLXDerived* are CLX-style classes descended from *TObject*. The virtual method *what_am_I()* is overridden in both derived classes, but is called *only* in the base class constructors, not in derived class constructors.

```
#include <sysutils.hpp>
#include <iostream.h>
// non-CLX style classes
class MyBase {
public:
   MyBase() { what_am_I(); }
   virtual void what_am_I() { cout << "I am a base" << endl; }
};
class MyDerived : public MyBase {
public:
   virtual void what_am_I() { cout << "I am a derived" << endl; }
};
// CLX-style classes
class MyCLXBase : public TObject {
public:
   __fastcall MyCLXBase() { what_am_I(); }
   virtual void __fastcall what_am_I() { cout << "I am a base" << endl; }
};
class MyCLXDerived : public MyCLXBase {
public:
   virtual void __fastcall what_am_I() { cout << "I am a derived" << endl; }
};
int main(void)
{
   MyDerived d;// instantiation of the C++ class
   MyCLXDerived *pvd = new MyCLXDerived;// instantiation of the CLX-style class
   return 0;
}
```

The output of this example is

```
I am a base
I am a derived
```

because of the difference in the runtime types of *MyDerived* and *MyCLXDerived* during the calls to their respective base class constructors.

TOPIC~>

Constructor initialization of data members for virtual functions

<~TOPIC<~HEAD
^#constructorinitializationofdatamembersforvirtualfunctions
^Aconstructorinitializationofdatamembersforvirtualfunctions
^Xcallingvirtualmethodsinbaseclassconstructors
^$Constructor initialization of data members for virtual functions
^+languagesupport:000
^Kobject construction;constructing objects;creating objects;virtual functions
^T SUPP4VCL
^C 3
HEAD~>Constructor initialization of data members for virtual functions
Because data members may be used in virtual functions, it is important to understand when and how they are initialized. In Delphi, all uninitialized data is zero-initialized. This applies, for example, to base classes whose constructors are not called with **inherited**. In standard C++, there is no guarantee of the value of uninitialized data members. The following types of class data members must be initialized in the initialization list of the class's constructor:

- References
- Data members with no default constructor

Nevertheless, the value of these data members, or those initialized in the body of the constructor, is undefined when the base class constructors are called. In C++, the memory for CLX-style classes is zero-initialized.

Note Technically, it is the memory of the CLX class that is zero, that is the bits are zero, the values are actually undefined. For example, a reference is zero.

A virtual function which relies upon the value of member variables initialized in the body of the constructor or in the initialization list may behave as if the variables were initialized to zero. This is because the base class constructor is called before the initialization list is processed or the constructor body is entered. The following example illustrates this:

```
#include <sysutils.hpp>

class Base : public TObject {
public:
   __fastcall Base() { init(); }
   virtual void __fastcall init() { }
};

class Derived : public Base {
```

```
public:
   Derived(int nz) : not_zero(nz) { }
   virtual void __fastcall init()
   {
      if (not_zero == 0)
         throw Exception("not_zero is zero!");
   }
private:
   int not_zero;
};

int main(void)
{
   Derived *d42 = new Derived(42);
   return 0;
}
```

This example throws an exception in the constructor of *Base*. Because *Base* is constructed before *Derived*, *not_zero*, has not yet been initialized with the value of 42 passed to the constructor. Be aware that you cannot initialize data members of your CLX-style class before its base class constructors are called.

TOPIC~>

Object destruction

<~TOPIC<~HEAD
^#objectdestruction
^Aobjectdestruction
^Xc++andobjectpascalmodels;objectconstructionforc++buildervclclasses
^$Object destruction
^+languagesupport:000
^Kobject destruction;destroying objects;exceptions thrown from constructors;virtual methods
^T SUPP4VCL
^C 3
HEAD~>Object destruction

Two mechanisms concerning object destruction work differently in standard C++ from the way they do in Delphi. These are:

- Destructors called because of exceptions raised from constructors.
- Virtual methods called from destructors.

CLX-style classes combine the methods of these two languages. The issues are discussed below.

Exceptions thrown from constructors

Destructors are called differently in C++ than in Delphi if an exception is raised during object construction. Take as an example, class *C*, derived from class *B*, which is derived from class *A*:

```
class A
```

```
{
   // body
};

class B: public A
{
   // body
};

class C: public B
{
   // body
};
```

Consider the case where an exception is raised in the constructor of class *B* when constructing an instance of *C*. What results in standard C++, Delphi, and CLX-style C++ classes is described here:

- In standard C++, first the destructors for all completely constructed object data members of *B* are called, then *A*'s destructor is called, then the destructors for all completely constructed data members of *A* are called. However, the destructors for *B* and *C* are not called.
- In Delphi, only the instantiated class destructor is called automatically. This is the destructor for *C*. As with constructors, it is entirely the programmer's responsibility to call inherited in destructors. In this example, if we assume all of the destructors call inherited, then the destructors for C, B, and A are called in that order. Moreover, whether or not inherited was already called in B's constructor before the exception occurred, A's destructor is called because inherited was called in B's destructor. Calling the destructor for A is independent of whether its constructor was actually called. More importantly, because it is common for constructors to call inherited immediately, the destructor for *C* is called whether or not the body of its constructor was completely executed.
- In CLX-style classes, the CLX bases (implemented in Delphi) follow the Delphi method of calling destructors. The derived classes (implemented in C++) do not follow either language method exactly. What happens is that all the destructors are called; but the bodies of those that would not have been called, according to C++ language rules, are not entered.

Classes implemented in Delphi thereby provide an opportunity to process any cleanup code you write in the body of the destructor. This includes code that frees memory for sub-objects (data members that are objects) that are constructed before a constructor exception occurs. Be aware that, for CLX-style classes, the clean-up code may not be processed for the instantiated class or for its C++-implemented bases, even though the destructors are called.

For more information on handling exceptions in C++ CLX applications, refer to "Handling CLX exceptions in C++" on page 13-18<~JMP Handling CLX exceptions in C++~!Alink(HandlingCLXExceptionsInCPP) JMP~>.

Virtual methods called from destructors

Virtual method dispatching in destructors follows the same pattern that it did for constructors. This means that for CLX-style classes, the derived class is destroyed first, but the runtime type of the object remains that of the derived class throughout subsequent calls to base class destructors. Therefore, if virtual methods are called in CLX base class destructors, you are potentially dispatching to a class that has already destroyed itself.

TOPIC~>

AfterConstruction and BeforeDestruction

<~TOPIC<~HEAD
^#afterconstructionandbeforedestruction
^Aafterconstructionandbeforedestruction
^Xc++andobjectpascalmodels;objectconstructionforc++buildervclclasses;objectdestr
uction
^$AfterConstruction and BeforeDestruction
^+languagesupport:000
^Kobject destruction;destroying objects;object construction;constructing
objects;creating objects
^T SUPP4VCL
^C 3
HEAD~>AfterConstruction and BeforeDestruction

TObject introduces two virtual methods, *BeforeDestruction* and *AfterConstruction*, that allow programmers to write code that is processed before and after objects are destroyed and created, respectively. *AfterConstruction* is called after the last constructor is called. *BeforeDestruction* is called before the first destructor is called. These methods are public and are called automatically.

TOPIC~>

Class virtual functions

<~TOPIC<~HEAD
^#classvirtualfunctions
^Aclassvirtualfunctions
^Xc++andobjectpascalmodels
^$Class virtual functions
^+languagesupport:000
^Kclass virtual functions;virtual functions;functions
^T SUPP4VCL
^C 3
HEAD~>Class virtual functions

Delphi has the concept of a class virtual function. The C++ analogy would be a static virtual function, if it were possible; but C++ has no exact counterpart to this type of function. These functions are safely called internally from CLX. However, you should never call a function of this type in C++ code. You can identify these functions in the header files because they are preceded by the following comment:

```
/* virtual class method */
```

TOPIC~>

Support for Delphi data types and language concepts

<~TOPIC<~HEAD
^#supportforobjectpascaldatatypesandlanguageconcepts
^Asupportforobjectpascaldatatypesandlanguageconcepts
^Xc++languagesupportforthevcl;c++andobjectpascalmodels
^$Support for Delphi data types and language concepts
^+languagesupport:000
^KDelphi;Delphi and C++;C++ and Delphi
^T SUPP4VCL
^C 3
HEAD~>Support for Delphi data types and language concepts

To support CLX, the Borland extensions to C++ implement, translate, or otherwise map most Delphi data types, constructs, and language concepts to the C++ language. This is done through typedefs to native C++ types; classes, structs, and class templates; C++ language counterparts; macros; and keywords that are ANSI-conforming language extensions.This is done in the following ways:

- Typedefs to native C++ types
- Classes, structs, and class templates
- C++ language counterparts
- Macros
- Keywords that are ANSI-conforming language extensions

Not all aspects of the Delphi language map cleanly to C++. Occasionally, using these parts of the language can lead to unexpected behavior in your application. For example:

- Some types exist in both Delphi and in C++, but are defined differently. These can require caution when sharing code between these two languages.
- Some extensions were added to Delphi for the purpose of supporting the Borland extensions to C++. Occasionally these can have subtle interoperability impact.
- Delphi types and language constructs that have no mapping to the C++ language should be avoided in C++ when sharing code between these languages.

This sectionThe following topics summarizes how the Borland extensions to C++ implement the Delphi language, and suggests when to use caution.

Topics in this section:

- <~JMP Typedefs~typedefs JMP~>
- <~JMP Classes that support the Delphi language~classesthatsupporttheobjectpascallanguage JMP~>
- <~JMP C++ language counterparts to the Delphi language~c++languagecounterpartstotheobjectpascallanguage JMP~>
- <~JMP Open arrays~openarrays JMP~>

- <~JMP Types defined differently~typesdefineddifferently JMP~>
- <~JMP Resource strings~resourcestrings JMP~>
- <~JMP Default parameters~defaultparameters JMP~>
- <~JMP Runtime type information~runtimetypeinformation JMP~>
- <~JMP Unmapped types~unmappedtypes JMP~>
- <~JMP Keyword extensions~keywordextensions JMP~>
- <~JMP The __declspec keyword extension~thedeclspeckeywordextension JMP~>

TOPIC~>

Typedefs

<~TOPIC<~HEAD
^#typedefs
^Atypedefs
^Xsupportforobjectpascaldatatypesandlanguageconcepts;unmappedtypes
^$Typedefs
^+languagesupport:000
^Ktypedefs;types, C++ and Delphi;data types, C++ and Delphi
^T SUPP4VCL
^C 3
HEAD~>Typedefs

Most Delphi intrinsic data types are implemented in C++ using a **typedef** to a native C++ type. These are found in sysmac.h. Whenever possible you should use the native C++ type, rather than the Delphi type.

TOPIC~>

Classes that support the Delphi language

<~TOPIC<~HEAD
^#classesthatsupporttheobjectpascallanguage
^Aclassesthatsupporttheobjectpascallanguage
^Xsupportforobjectpascaldatatypesandlanguageconcepts
^$Classes that support the Delphi language
^+languagesupport:000
^KDelphi,special classes for compatibility;emulation classes,Delphi
^T SUPP4VCL
^C 3
HEAD~>Classes that support the Delphi language

Some Delphi data types and language constructs that do not have a built-in C++ counterpart are implemented as classes or structs. Class templates are also used to implement data types as well as Delphi language constructs, like **set,** from which a specific type can be declared. The declarations for these are found in the following header files:

- dstring.h
- wstring.h
- sysclass.h
- syscomp.h
- syscurr.h
- sysdyn.h
- sysopen.h
- sysset.h
- systdate.h
- systobj.h
- systvar.h
- sysvari.h

The classes implemented in these header files were created to support native types used in Delphi routines. They are intended to be used when calling these routines in CLX-based code.

TOPIC~>

C++ language counterparts to the Delphi language

<~TOPIC<~HEAD
^#c++languagecounterpartstotheobjectpascallanguage
^Ac++languagecounterpartstotheobjectpascallanguage;varparameters
^Xsupportforobjectpascaldatatypesandlanguageconcepts
^$C++ language counterparts to the Delphi language
^+languagesupport:000
^Kparameters, var;var parameters;parameters, untyped;untyped parameters;
^T SUPP4VCL
^C 3
HEAD~>C++ language counterparts to the Delphi language

Delphi **var** and untyped parameters are not native to C++. Both have C++ language counterparts that are used in CLX.

Var parameters

Both C++ and Delphi have the concept of "pass by reference." These are modifiable arguments. In Delphi they are called var parameters. The syntax for functions that take a **var** parameter is

```
procedure myFunc(var x : Integer);
```

In C++, you should pass these types of parameters by reference:

```
void myFunc(int& x);
```

Both C++ references and pointers can be used to modify the object. However, a reference is a closer match to a **var** parameter because, unlike a pointer, a reference cannot be rebound and a **var** parameter cannot be reassigned; although, either can change the *value* of what it references.

Untyped parameters

Delphi allows parameters of an unspecified type. These parameters are passed to functions with no type defined. The receiving function must cast the parameter to a known type before using it. The C++ version of CLX interprets untyped parameters as pointers-to-void (*void* *). The receiving function must cast the void pointer to a pointer of the desired type. Following is an example:

```
int myfunc(void* MyName)
{
   // Cast the pointer to the correct type; then dereference it.
   int* pi = static_cast<int*>(MyName);
   return 1 + *pi;
}
```

TOPIC~>

Open arrays

<~TOPIC<~HEAD
^#openarrays
^Aopenarrays
^Xsupportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartstotheobjectpascallanguage
^$Open arrays
^+languagesupport:000
^Kopen arrays;arrays, open
^T SUPP4VCL
^C 3
HEAD~>Open arrays

Delphi has an "open array" construct that permits an array of unspecified size to be passed to a function. While there is no direct support in C++ for this type, a Delphi function that has an open array parameter can be called by explicitly passing both pointer to the first element of an array, and the value of the last index (number of array elements, minus one). For example, the *Mean* function in math.hpp has this declaration in Delphi:

```
function Mean(Data: array of Double): Extended;
```

The C++ declaration is:

```
Extended __fastcall Mean(const double * Data, const int Data_Size);
```

The following code illustrates calling the *Mean* function from C++:

```
double d[] = { 3.1, 4.4, 5.6 };

// explicitly specifying last index
long double x = Mean(d, 2);

// better: use sizeof to ensure that the correct value is passed
long double y = Mean(d, (sizeof(d) / sizeof(d[0])) - 1);

// use macro in sysopen.h
long double z = Mean(d, ARRAYSIZE(d) - 1);
```

Note In cases similar to the above example, but where the Delphi function takes a var parameter, the C++ function declaration parameters will not be const.

- <~JMP Calculating the number of elements~calculatingthenumberofelements JMP~>
- <~JMP Temporaries~temporaries JMP~>
- <~JMP array of const~arrayofconst JMP~>
- <~JMP OPENARRAY macro~openarraymacro JMP~>
- <~JMP EXISTINGARRAY macro~existingarraymacro JMP~>
- <~JMP C++ functions that take open array arguments~c++functionsthattakeopenarrayarguments JMP~>

TOPIC~>

Calculating the number of elements

<~TOPIC<~HEAD
^#calculatingthenumberofelements
^Acalculatingthenumberofelements
^Xopenarrays;existingarraymacro
^$Calculating the number of elements
^+languagesupport:000
^Kopen arrays;arrays, open
^T SUPP4VCL
^C 3
HEAD~>Calculating the number of elements
When using *sizeof()*, the ARRAYSIZE macro, or the EXISTINGARRAY macro to calculate the number of elements in an array, be careful not to use a pointer to the array. Instead, pass the name of the array itself:

```
double d[] = { 3.1, 4.4, 5.6 };
int n = ARRAYSIZE(d); // sizeof(d)/sizeof(d[0]) => 24/8 => 3

double *pd = d;
int m = ARRAYSIZE(pd); // sizeof(pd)/sizeof(pd[0]) => 4/8 => 0 => Error!
```

Taking the "sizeof" an array is not the same as taking the "sizeof" a pointer. For example, given the following declarations,

```
double d[3];
double *p = d;
```

taking the size of the array as shown here

```
sizeof(d)/sizeof d[0]
```

does not evaluate the same way as taking the size of the pointer:

```
sizeof(p)/sizeof(p[0])
```

This example and those following use the ARRAYSIZE macro instead of the *sizeof()* operator. For more information about the ARRAYSIZE macro, see the online Help.

TOPIC~>

Temporaries

<~TOPIC<~HEAD
^#temporaries
^Atemporaries
^Xopenarrays
^$Temporaries
^+languagesupport:000
^Kopen arrays;arrays, open;temporaries;temporary open arrays
^T SUPP4VCL
^C 3
HEAD~>Temporaries
Delphi provides support for passing unnamed temporary open arrays to functions. There is no syntax for doing this in C++. However, since variable definitions can be intermingled with other statements, one approach is to simply provide the variable with a name.

In Delphi:

```
Result := Mean([3.1, 4.4, 5.6]);
```

In C++, using a named “temporary”:

```
double d[] = { 3.1, 4.4, 5.6 };
return Mean(d, ARRAYSIZE(d) - 1);
```

To restrict the scope of the named “temporary” to avoid clashing with other local variables, open a new scope in place:

```
long double x;
{
   double d[] = { 4.4, 333.1, 0.0 };
   x = Mean(d, ARRAYSIZE(d) - 1);
}
```

For another solution, see “OPENARRAY macro” on page 14-27use the <~JMP OPENARRAY macro~openarraymacro JMP~>.

TOPIC~>

### array of const

<~TOPIC<~HEAD
^#arrayofconst
^Aarrayofconst
^Xopenarrays
^$array of const
^+languagesupport:000
^Kopen arrays;arrays, open;array of const;ARRAYOFCONST
^T SUPP4VCL
^C 3
HEAD~>array of const
Delphi supports a language construct called an array of const. This argument type is the same as taking an open array of *TVarRec* by value.

The following is a Delphi code segment declared to accept an array of const:

```
function Format(const Format: string; Args: array of const): string;
```

In C++, the prototype is

```
AnsiString __fastcall Format(const AnsiString Format,
      TVarRec const *Args, const int Args_Size);
```

The function is called just like any other function that takes an open array:

```
void show_error(int error_code, AnsiString const &error_msg)
{
   TVarRec v[] = { error_code, error_msg };
   ShowMessage(Format("%d: %s", v, ARRAYSIZE(v) - 1));
}
```

TOPIC~>

OPENARRAY macro

<~TOPIC<~HEAD
^#openarraymacro
^Aopenarraymacro
^Xopenarrays;existingarraymacro
^$ OPENARRAY macro
^+languagesupport:000
^Kopen arrays;arrays, open;OPENARRAY macro
^T SUPP4VCL
^C 3
HEAD~>OPENARRAY macro
The OPENARRAY macro defined in sysopen.h can be used as an alternative to using a named variable for passing a temporary open array to a function that takes an open array by value. The use of the macro looks like

```
OPENARRAY(T, (value1, value2, value3)) // up to 19 values
```

where *T* is the type of open array to construct, and the *value* parameters will be used to fill the array. The parentheses around the *value* arguments are required. For example:

```
void show_error(int error_code, AnsiString const &error_msg)
{
   ShowMessage(Format("%d: %s", OPENARRAY(TVarRec, (error_code, error_msg))));
}
```

Up to 19 values can be passed when using the OPENARRAY macro. If a larger array is needed, an explicit variable must be defined. Additionally, using the OPENARRAY macro incurs an additional (but small) runtime cost, due both to the cost of allocating the underlying array, and to an additional copy of each value.

TOPIC~>

EXISTINGARRAY macro

<~TOPIC<~HEAD
^#existingarraymacro
^Aexistingarraymacro
^Xopenarrays;openarraymacro
^$EXISTINGARRAY macro
^+languagesupport:000
^Kopen arrays;arrays, open;EXISTINGARRAY macro
^T SUPP4VCL
^C 3
HEAD~>EXISTINGARRAY macro
The EXISTINGARRAY macro defined in sysopen.h can be used to pass an existing array where an open array is expected. The use of the macro looks like

```
long double Mean(const double *Data, const int Data_Size);
double d[] = { 3.1, 3.14159, 2.17128 };
Mean(EXISTINGARRAY (d));
```

Note The section "Calculating the number of elements" on page 14-25topic <~JMP Calculating the number of elements~calculatingthenumberofelements JMP~> also applies to the EXISTINGARRAY macro.

TOPIC~>

C++ functions that take open array arguments

<~TOPIC<~HEAD
^#c++functionsthattakeopenarrayarguments
^Ac++functionsthattakeopenarrayarguments
^Xopenarrays
^$C++ functions that take open array arguments
^+languagesupport:000
^Kopen arrays;arrays, open
^T SUPP4VCL
^C 3
HEAD~>C++ functions that take open array arguments
When writing a C++ function that will be passed an open array from Delphi, it is important to explicitly maintain "pass by value" semantics. In particular, if the declaration for the function corresponds to "pass by value", be sure to explicitly copy any elements before modifying them. In Delphi, an open array is a built-in type and can be passed by value. In C++, the open array type is implemented using a pointer, which will modify the original array unless you make a local copy of it.

TOPIC~>

Types defined differently

<~TOPIC<~HEAD
^#typesdefineddifferently
^Atypesdefineddifferently

^Xsupportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartstotheobjectpascallanguage;unmappedtypes;typedefs
^$Types defined differently
^+languagesupport:000
^Ktypes, C++ and Pascal;data types, C++ and Pascal
^T SUPP4VCL
^C 3
HEAD~>Types defined differently

Types that are defined differently in Delphi and C++ are not normally cause for concern. The rare cases in which they are problematic may be subtle. For this reason, these types are mentioned in this sectiontopic.

Boolean data types

The true value for the Delphi *ByteBool*, *WordBool*, and *LongBool* data types is represented in Delphi as –1. *False* is represented as 0.

Note The Boolean data type remains unchanged (true = 1, false = 0).

While the C++ bool type converts these Delphi types correctly, there is a problem when sharing a function that uses an integer type (having a value of 1) to represent the boolean value (Windows API functions do this with the Win32 type BOOL) . If true is passed to a parameter of an integer type (BOOL on Windows), it evaluates to –1 in Delphi and to 1 in C++. Therefore, if you are sharing code between these two languages, any comparison of the two identifiers may fail unless they are both 0 (*False*, false). As a workaround, you can use the following method of comparison:

```
!A == !B;
```

Table 14.2The table below shows the results of using this method of equality comparison:

Table 14.2 Equality comparison !A == !B of BOOL variables

Delphi	C++	!A == !B
0 (False)	0 (false)	!0 == !0 (TRUE)
0 (False)	1 (true)	!0 == !1 (FALSE)
–1 (True)	0 (false)	!–1 == !0 (FALSE)
–1 (True)	1 (true)	!–1 == !1 (TRUE)

With this method of comparison, any set of values will evaluate correctly.

Char data types

The char type in C++ is a signed type, whereas it is an unsigned type in Delphi. It is extremely rare that a situation would occur in which this difference would be a problem when sharing code.

TOPIC~>

Delphi interfaces

<~TOPIC<~HEAD
^#DelphiInterfaces
^A DelphiInterfaces
^X DelphiInterface_object;DeclaringInterfaceClasses
^$Delphi interfaces
^+languagesupport:000
^K interfaces, Delphi;DelphiInterface,
^T SUPP4VCL
^C 3
HEAD~>Delphi interfaces

The Delphi compiler handles many of the details in working with interfaces automatically. It automatically augments the reference count on an interface when application code acquires an interface pointer and decrements that reference count when the interface goes out of scope.

In C++, the *DelphiInterface* template class provides some of that convenience for C++ interface classes. For properties and methods in CLX that use interface types in Delphi, the C++ wrappers use a *DelphiInterface* that is built using the underlying interface class.

The *DelphiInterface* constructor, copy constructor, assignment operator, and destructor all increment or decrement reference counts as needed. However, *DelphiInterface* is not quite as convenient as the compiler support for interfaces in Delphi. Other operators that provide access to the underlying interface pointer do not handle reference counting because the class can't always tell where that is appropriate. You may need to explicitly call AddRef or Release to ensure proper reference counting.

TOPIC~>

Resource strings

<~TOPIC<~HEAD
^#resourcestrings
^Aresourcestrings
^X
supportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartstoth
eobjectpascallanguage
^$ Resource strings
^+languagesupport:000
^Kresource strings, Pascal
^T SUPP4VCL
^C 3
HEAD~>Resource strings
If you have code in a Delphi unit that uses resource strings, the Delphi compiler (DCC32) generates a global variable and a corresponding preprocessor macro for each resource string when it generates the header file. The macros are used to

automatically load the resource strings, and are intended to be used in your C++ code in all places where the resource string is referenced. For example, the **resourcestring** section in the Delphi code could contain

```
unit borrowed;
interface
resourcestring
   Warning = 'Be careful when accessing string resources.';
implementation
begin
end.
```

The corresponding code using the Borland extensions to C++ would be:

```
extern PACKAGE System::Resource ResourceString _Warning;
#define Borrowed_Warning System::LoadResourceString(&Borrowed::_Warning)
```

This enables you to use the exported Delphi resource string without having to explicitly call *LoadResourceString*.

TOPIC~>

Default parameters

<~TOPIC<~HEAD
^#defaultparameters
^Adefaultparameters
^Xsupportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartstotheobjectpascallanguage
^$Default parameters
^+languagesupport:000
^Kdefault parameters
^T SUPP4VCL
^C 3
HEAD~>Default parameters

The Delphi compiler now accepts default parameters for compatibility with C++ regarding constructors. Unlike C++, Delphi constructors can have the same number and types of parameters, since they are uniquely named. In such cases, dummy parameters are used in the Delphi constructors to distinguish them when the C++ header files are generated. For example, for a class named *TInCompatible*, the Delphi constructors could be:

```
constructor Create(AOwner: TComponent);
constructor CreateNew(AOwner: TComponent);
```

which would translate, without default parameters, to the following ambiguous code in C++ for both constructors:

```
__fastcall TInCompatible(Classes::TComponent* Owner);// C++ version of the Pascal Create
constructor

__fastcall TInCompatible(Classes::TComponent* Owner);// C++ version of the Pascal CreateNew
constructor
```

However, using default parameters, for a class named *TCompatible*, the Delphi constructors are:

```
constructor Create(AOwner: TComponent);
constructor CreateNew(AOwner: TComponent; Dummy: Integer = 0);
```

They translate to the following unambiguous code in C++:

```
__fastcall TCompatible(Classes::TComponent* Owner);// C++ version of the Pascal Create
constructor

__fastcall TCompatible(Classes::TComponent* Owner, int Dummy);// C++ version of the Pascal
CreateNew constructor
```

Note The main issue regarding default parameters is that the Delphi compiler strips out the default value of the default parameter. Failure to remove the default value would lead to the ambiguity that would occur if there were not defaults at all. You should be aware of this when using CLX classes or when using third-party components.

TOPIC~>

Runtime type information

<~TOPIC<~HEAD
^#runtimetypeinformation
^Aruntimetypeinformation
^Xsupportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartst otheobjectpascallanguage
^$Runtime type information
^+languagesupport:000
^Ktype information, runtime
^T SUPP4VCL
^C 3
HEAD~>Runtime type information

Delphi has language constructs dealing with RTTI. Some have C++ counterparts. These are listed in Table 14.3the table below:

Table 14.3 Examples of RTTI mappings from Delphi to C++

Delphi RTTI	C++ RTTI
if Sender **is** TButton...	**if** (dynamic_cast <TButton*> (Sender) // dynamic_cast returns **NULL** on failure.
b := Sender **as** TButton; (* raises an exception on failure *)	TButton& ref_b = dynamic_cast <TButton&> (*Sender) // throws an exception on failure.
ShowMessage(Sender.ClassName);	ShowMessage(typeid(*Sender).name());

In Table 14.3, *ClassName* is a *TObject* method that returns a string containing the name of the actual type of the object, regardless of the type of the declared variable. Other RTTI methods introduced in *TObject* do not have C++ counterparts. These are all public and are listed here:

- *ClassInfo* returns a pointer to the runtime type information (RTTI) table of the object type.
- *ClassNameIs* determines whether an object is of a specific type.
- *ClassParent* returns the type of the immediate ancestor of the class. In the case of *TObject*, *ClassParent* returns nil (Delphi) because *TObject* has no parent. It is used by the **is** and **as** operators, and by the *InheritsFrom* method.
- *ClassType* dynamically determines the actual type of an object. It is used internally by the Delphi **is** and **as** operators.
- *FieldAddress* uses RTTI to obtain the address of a published field. It is used internally by the steaming system.
- *InheritsFrom* determines the relationship of two objects. It is used internally by the Delphi **is** and **as** operators.
- *MethodAddress* uses RTTI to find the address of a method. It is used internally by the steaming system.

Some of these methods of *TObject* are primarily for internal use by the compiler or the streaming system. For more information about these methods, see the online Help.

TOPIC~>

Unmapped types

<~TOPIC<~HEAD
^#unmappedtypes
^Aunmappedtypes
^Xsupportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartst otheobjectpascallanguage;typesdefineddifferently;typedefs
^$Unmapped types
^+languagesupport:000
^Ktypes, C++ and Pascal;data types, C++ and Pascal
^T SUPP4VCL
^C 3
HEAD~>Unmapped types

6-byte Real types

The old Delphi (old Object Pascal) 6-byte floating-point format is now called *Real48*. The old *Real* type is now a double. C++ does not have a counterpart for the *Real48* type. Consequently, you should not use Delphi code that includes this type with C++ code. If you do, the header file generator will generate a warning.

Arrays as return types of functions

In Delphi a function can take as an argument, or return as a type, an array. For example, the syntax for a function *GetLine* returning an array of 80 characters is

```
type
  Line_Data = array[0..79] of char;
function GetLine: Line_Data;
```

C++ has no counterpart to this concept. In C++, arrays are not allowed as return types of functions. Nor does C++ accept arrays as the type of a function argument.

Be aware that, although CLX does not have any properties that are arrays, the Delphi language does allow this. Because properties can use Get and Set read and write methods that take and return values of the property's type, you cannot have a property of type array in a CLX-style class in C++.

Note Array properties, which are also valid in Delphi, are not a problem in C++ because the Get method takes an index value as a parameter, and the Set method returns an object of the type contained by the array. For more information about array properties, see "Creating array properties" on page 37-11<~JMP Creating array properties~!Alink(creatingarrayproperties) JMP~>.

TOPIC~>

Keyword extensions

<~TOPIC<~HEAD
^#keywordextensions
^Akeywordextensions;vclspecifickeywordextensions
^Xsupportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartst otheobjectpascallanguage;vclclassdeclarations
^$Keyword extensions
^+languagesupport:000
^Kkeywords, for Pascal compatibility;classid keyword;closure keyword;property keyword;published keyword
^T SUPP4VCL
^C 3
HEAD~>Keyword extensions

This section describes ANSI-conforming keyword extensions implemented in the Borland extensions to C++ to support CLX. For a complete list of keywords and keyword extensions in the Borland extensions to C++, see the online Help.

The <~JMP __declspec~thedeclspeckeywordextension JMP~> keyword also provides CLX support.

__classid

The __classid operator is used by the compiler to generate a pointer to the vtable for the specified *classname*. This operator is used to obtain the meta class from a class.

Syntax

```
__classid(classname)
```

For example, __classid is used when registering property editors, components, and classes, and with the *InheritsFrom* method of *TObject*. The following code illustrates the use of __classid for creating a new component derived from *TWinWidgetControl*:

```
namespace Ywndctrl
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(MyWndCtrl)};
    RegisterComponents("Additional", classes, 0);
  }
}
```

__closure

The **__closure** keyword is used to declare a special type of pointer to a member function. In standard C++, the only way to get a pointer to a member function is to use the fully qualified member name, as shown in the following example:

```
class base
{
  public:
    void func(int x) { };
};

typedef void (base::* pBaseMember)(int);

int main(int argc, char* argv[])
{
  base        baseObject;
  pBaseMember m = &base::func; // Get pointer to member 'func'

  // Call 'func' through the pointer to member
  (baseObject.*m)(17);
  return 0;
}
```

However, you cannot assign a pointer to a member of a derived class to a pointer to a member of a base class. This rule (called *contravariance*) is illustrated in the following example:

```
class derived: public base
{
  public:
    void new_func(int i) { };
};

int main(int argc, char* argv[])
{
  derived        derivedObject;
  pBaseMember m = &derived::new_func; // ILLEGAL

  return 0;
}
```

The __closure keyword extension allows you to skirt this limitation, and more. Using a closure, you can get a pointer to member function for an object (i.e. a particular instance of a class). The object can be *any* object, regardless of its inheritance hierarchy. The object's **this** pointer is automatically used when calling the member function through the closure. The following example shows how to declare and use a closure. The *base* and *derived* classes provided earlier are assumed to be defined.

```
int main(int argc, char* argv[])
{
  derived         derivedObject;
  void (__closure *derivedClosure)(int);

     derivedClosure = derivedObject.new_func; // Get a pointer to the 'new_func' member.
                                              // Note the closure is associated with the
                                              // particular object, 'derivedObject'.
     derivedClosure(3);  // Call 'new_func' through the closure.
     return 0;
}
```

Closures also work with pointers to objects, as illustrated in this example:

```
void func1(base *pObj)
{
  // A closure taking an int argument and returning void.
  void ( __closure *myClosure )(int);

  // Initialize the closure.
  myClosure = pObj->func;

  // Use the closure to call the member function.
  myClosure(1);
  return;
}

int main(int argc, char* argv[])
{
  derived         derivedObject;
  void (__closure *derivedClosure)(int);

     derivedClosure = derivedObject.new_func; // Same as before...
     derivedClosure(3);

     // We can use pointers to initialize a closure, too.
     // We can also get a pointer to the 'func' member function
     // in the base class.
     func1(&derivedObject);
     return 0;
}
```

Notice that we are passing a pointer to an instance of the *derived* class, and we are using it to get a pointer to a member function in the *base* class - something standard C++ does not allow us to do.

Closures are a key part of the GalileoC++ RAD environment. They give us the ability to assign an event handler in the Object Inspector. For example, a *TButton* object has an event called *OnClick*. In the *TButton* class, the *OnClick* event is a property that uses the **__closure** keyword extension in its declaration. The __closure keyword allows us to assign a member function of another class (typically a member function in a *TForm* object) to the property. When you place a *TButton* object on a form, and then create a handler for the button's *OnClick* event, the forms designer creates a member function in the button's *TForm* parent, and assigns that member function to the *OnClick* event of *TButton*. This way, the event handler is associated with that particular instance of *TButton*, and no other.

For more information about events and closures, see Chapter 38, "Creating events."<~JMP Creating events~!Alink(CreatingEvents) JMP~>

__property

The __property keyword declares an attribute of a class. Properties appear to the programmer just like any other attribute (field) of a class. However, like its Delphi counterpart, the __property keyword adds significantly more functionality beyond just examining and changing the value of the attribute. Since property attributes completely control access to the property, there are no restrictions on how you implement the property within the class itself.

Syntax

```
__property type propertyName[index1Type index1][indexNType indexN] = { attributes };
```

where

- *type* is an intrinsic or previously declared data type.
- *propertyName* is any valid identifier.
- *indexNType* is an intrinsic or previously declared data type.
- *indexN* is the name of an index parameter that will be passed to the property's read and write functions.
- *attributes* is a comma separated sequence of read, write, stored, default (or nodefault), or index.

The *indexN* parameters in square brackets are optional. If present, they declare an array property. The index parameters are passed to the read and write methods of the array property.

The following example shows some simple property declarations:

```
class PropertyExample {
   private:
      int Fx,Fy;
      float Fcells[100][100];

   protected:
      int  readX()         { return(Fx); }
      void writeX(int newFx) { Fx = newFx;  }

      double computeZ() {
            // Do some computation and return a floating point value...
            return(0.0);
      }
```

```
        float cellValue(int row, int col) { return(Fcells[row][col]); }

    public:
        __property int    X = { read=readX, write=writeX };
        __property int    Y = { read=Fy };
        __property double Z = { read=computeZ };
        __property float Cells[int row][int col] = { read=cellValue };
};
```

This example shows several property declarations. Property *X* has read-write access, through the member functions *readX* and *writeX*, respectively. Property *Y* corresponds directly to the member variable *Fy*, and is read-only. Property *Z* is a read-only value that is computed; it is not stored as a data member in the class. Finally, the *Cells* property demonstrates an array property with two indices. The next example shows how you would access these properties in your code:

```
PropertyExample myPropertyExample;
myPropertyExample.X = 42;          // Evaluates to: myPropertyExample.WriteX(42);
int    myVal1 = myPropertyExample.Y; // Evaluates to: myVal1 = myPropertyExample.Fy;
double myVal2 = myPropertyExample.Z; // Evaluates to: myVal2 = myPropertyExample.ComputeZ();
float  cellVal = myPropertyExample[3][7]; // Evaluates to:
                                          // cellVal = myPropertyExample.cellValue(3,7);
```

Properties have many other variations and features not shown in this example. Properties can also:

- Associate the same read or write method with more than one property, by using the index attribute
- Have default values
- Be stored in a form file
- Extend a property defined in a base class

For more information about properties, see Chapter 37, "Creating properties."<~JMP Creating properties~!Alink(creatingproperties) JMP~>

__published

The __published keyword specifies that properties in that section are displayed in the Object Inspector, if the class is on the Component palette. Only classes derived from TObject can have __published sections.

The visibility rules for published members are identical to those of public members. The only difference between published and public members is that Delphi-style runtime type information (RTTI) is generated for data members and properties declared in a __published section. RTTI enables an application to dynamically query the data members, member functions, and properties of an otherwise unknown class type.

Note No constructors or destructors are allowed in a __published section. Properties, Delphi intrinsic or CLX-derived data-members, member functions, and closures are allowed in a __published section. Fields defined in a __published section must be of a class type. Properties defined in a __published section cannot be array properties. The type of a property defined in a __published section must be an ordinal type, a real type, a string type, a small set type, a class type, or a method pointer type.

TOPIC~>

The __declspec keyword extension

<~TOPIC<~HEAD
^#thedeclspeckeywordextension
^Athedeclspeckeywordextension;delphiclass;delphireturn;dynamic;hidesbase;package;pascalimplementation;delphirtti;uuid
^Xsupportforobjectpascaldatatypesandlanguageconcepts;c++languagecounterpartstotheobjectpascallanguage;keywordextensions
^$The __declspec keyword extension
^+languagesupport:000
^Kkeywords, Pascal compatibility;declspec keyword;__declspec;delphireturn;delphiclass;hidesbase;package,declspec keyword;dynamic,declspec keyword;pascalimplementation;delphirtti;uuid
^T SUPP4VCL
^C 3
HEAD~>The __declspec keyword extension
Some arguments to the __declspec keyword extension provide language support for CLX. These arguments are listed below. Macros for the declspec arguments and combinations of them are defined in sysmac.h. In most cases you do not need to specify these. When you do need to add them, you should use the macros.

__declspec(delphiclass)

The **delphiclass** argument is used for declarations for classes derived from *TObject*. These classes will be created with the following compatibility:

- Delphi-compatible RTTI
- CLX-compatible constructor/destructor behavior
- CLX-compatible exception handling

A CLX-compatible class has the following restrictions:

- No virtual base classes are allowed.
- No multiple inheritance is allowed except for the case described in "Inheritance and interfaces" on page 14-3<~JMP Inheritance and interfaces~InheritanceAndVCLStyleClasses JMP~>.
- Must be dynamically allocated by using the global new operator.
- Must have a destructor.
- Copy constructors and assignment operators are not compiler-generated for CLX-derived classes.

A class declaration that is translated from Delphi will need this modifier if the compiler needs to know that the class is derived from *TObject*.

__declspec(delphireturn)

The **delphireturn** argument is for internal use only by the CLX framework in GalileoKylix for C++. It is used for declarations of classes that were created in GalileoKylix for C++ to support Delphi's built-in data types and language constructs because they do not have a native C++ type. These include *Currency*, *AnsiString*, *Variant*, *TDateTime*, and *Set*. The delphireturn argument marks C++ classes for CLX-compatible handling in function calls as parameters and return values. This modifier is needed when passing a structure by value to a function between Delphi and C++.

__declspec(delphirtti)

The **delphirtti** argument causes the compiler to include runtime type information in a class when it is compiled. When this modifier is used, the compiler generates runtime type information for all fields, methods, and properties that are declared in a published section. For interfaces, the compiler generates runtime type information for all methods of the interface. If a class is compiled with runtime type information, all of its descendants also include runtime type information. Because the class *TPersistent* is compiled with runtime type information, this means that there is no need to use this modifier with any classes you create that have *TPersistent* as an ancestor. This modifier is primarily used for interfaces in applications that implement or use Web Services.

__declspec(dynamic)

The **dynamic** argument is used for declarations for dynamic functions. Dynamic functions are similar to virtual functions except that they are stored only in the vtables for the objects that define them, not in descendant vtables. If you call a dynamic function, and that function is not defined in your object, the vtables of its ancestors are searched until the function is found. Dynamic functions are only valid for classes derived from *TObject*.

__declspec(hidesbase)

The **hidesbase** argument preserves Delphi program semantics when porting Delphi-virtual and override functions to C++. In Delphi, virtual functions in base classes can appear in the derived class as a function of the same name, but which is intended to be a completely new function with no explicit relation to the earlier one.

The compilers use the HIDESBASE macro, defined in sysmac.h, to specify that these types of function declarations are completely separate. For example, if a base class T1 declares a virtual function, *func*, taking no arguments, and its derived class T2 declared a function with the same name and signature, DCC32 -jphn would produce an HPP file with the following prototype:

```
virtual void T1::func(void);
HIDESBASE void T2::func(void);
```

Without the HIDESBASE declaration, the C++ program semantics indicate that virtual function *T1::func()* is being overridden by *T2::func()*.

__declspec(package)

The **package** argument indicates that the code defining the class can be compiled in a package<~~JMP package~!Alink(aboutpackages) JMP~>. This modifier is auto-generated by the compiler when creating packages in the IDE. See Chapter 16, "Working with packages and components" for more information about packages.

__declspec(pascalimplementation)

The **pascalimplementation** argument indicates that the code defining the class was implemented in Delphi. This modifier appears in a Delphi portability header file with an .hpp extension.

Windows only

__declspec(uuid)

The **uuid** argument associates a class with a globally unique identifier (GUID). This can be used with any class, but is typically used with classes that represent Delphi interfaces (or COM interfaces). You can retrieve the GUID of a class that was declared using this modifier by calling the **__uuidof** directive.

TOPIC~>

Chapter 15

Developing cross-platform applications

You can use Kylix to develop cross-platform 32-bit applications that run on both the Windows and Linux operating systems. Cross-platform applications use CLX components and don't make any operating system-specific API calls.

To develop a cross-platform application on Linux, either:

- Create a new application.
- Port and modify an existing Windows cross-platform application to Linux.

Then compile and deploy it on the platform you are running it on.

This chapter describes how to port Windows applications so they can compile on Linux or Windows and how to write code that is platform-independent and portable between the two environments. It also includes information on the differences between developing applications on Windows and Linux.

Creating cross-platform applications

You create cross-platform applications in the same way as you create any Galileo application. You need to use CLX visual and nonvisual components, and you should not use operating system specific APIs if you want the application to be completely cross-platform. (See "Writing portable code" on page 15-16 for tips on writing cross-platform applications.)

To create a new application on Kylix:

1 In the IDE, choose File | New | Application.

2 Develop, compile and test the application using the Delphi or C++ IDE.

For information on writing platform-independent database or Internet applications, see "Cross-platform database applications" on page 15-22 and "Cross-platform Internet applications" on page 15-28.

Porting Windows applications to Linux

If you have Borland RAD applications that were written for the Windows environment, you can port them to the Linux environment. How easy it will be depends on the nature and complexity of the application and how many Windows dependencies there are.

The following sections describe some of the major differences between the Windows and Linux environments and provide guidelines on how to get started porting an application.

Porting techniques

The following are different approaches you can take to port an application from one platform to another:

Table 15.1 Porting techniques

Technique	Description
Platform-specific port	Targets an operating system and underlying APIs
Cross-platform port	Targets a cross-platform API
Windows emulation	Leaves the code alone and ports the API it uses

Platform-specific ports

Platform-specific ports tend to be time-consuming, expensive, and only produce a single targeted result. They create different code bases, which makes them particularly difficult to maintain. However, each port is designed for a specific operating system and can take advantage of platform-specific functionality. Thus, the application typically runs faster.

Cross-platform ports

Cross-platform ports tend to be time-saving because the ported applications target multiple platforms. However, the amount of work involved in developing cross-platform applications is highly dependent on the existing code. If code has been developed without regard for platform independence, you may run into scenarios where platform-independent logic and platform-dependent implementation are mixed together.

The cross-platform approach is the preferable approach because business logic is expressed in platform-independent terms. Some services are abstracted behind an internal interface that looks the same on all platforms, but has a specific implementation on each. The runtime library is an example of this. The interface is very similar on both platforms, although the implementation may be vastly different.

You should separate cross-platform parts, then implement specific services on top. In the end, this approach is the least expensive solution, because of reduced maintenance costs due to a largely shared source base and an improved application architecture.

Windows emulation ports

Windows emulation is the most complex method and it can be very costly, but the resulting Linux application will look most similar to an existing Windows application. This approach involves implementing Windows functionality on Linux. From an engineering point of view, this solution is very hard to maintain.

Porting your application

If you are porting an application to Linux that you want to run on Linux only, you may choose to remove Windows-specific features entirely. If, however, you are porting an application that you want to run on both platforms, you need to modify your code or use conditional compiler directives to indicate sections of the code that apply specifically to Windows or Linux.

Follow these general steps to port your Windows cross-platform application to Linux:

1 Move your Windows CLX application source files and other project-related files onto your Linux computer. You can share source files between Linux and Windows if you want the program to run on both platforms. Or you can transfer the files using a tool such as ftp using the ASCII mode.

   Source files should include your Delphi or C++ unit files (.pas or .cpp files), project files (.dpr or .bpr file), and any package files (.dpk or bpk files). Project-related files include form files (.dfm files), resource files (.res files), and project options file (.dof (Delphi) and default.bpr (C++)—in Kylix .dof changes to .kof). If you want to compile your application from the command line only (rather than using the IDE), you'll need the configuration file (.cfg file—in Kylix this changes to .conf).

2 Open your project files in Kylix.

   In C++, you must create a new project and add the source files first, rather than open a Windows cross-platform C++ project.

3 Reset your project options.

   The .dof file (Delphi) or default.bpr file (C++) that stores the default project options is recreated on Kylix with a .kof extension (Delphi) or as a default.bpr file (C++). In the Delphi IDE, you can also store many of the compiler options with the application by typing Ctrl+O+O. The options are placed at the beginning of the currently open file.

4 If you plan to single source the application for use on both Windows and Linux, rename your form files (.dfm) to cross-platform form files (.xfm). For example, in Delphi, rename unit1.dfm to unit1.xfm. Or add a conditional compiler directive. An .xfm form file works on both Windows or Linux but a .dfm form only works on Windows.

- In Delphi, change `{$R *.dfm}` to `{$R *.xfm}` in the implementation section.
- In C++, change `#pragma resource "*.dfm"` to `#pragma resource "*.xfm"`in the header file.

5 Change all uses clauses (Delphi) and header files in your source file (C++) so they refer to the correct units in CLX. (See "Windows-only and cross-platform unit comparison" on page 15-10 for information.)

Delphi example

In Delphi, change the following uses clause:

```
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls;
```

to the following:

```
uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;
```

C++ example

In C++, change the following #include statements in the header file:

```
#include <vcl.h>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
```

to the following:

```
#include <clx.h>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
```

6 Rewrite any code that requires Windows dependencies by making the code more platform-independent. Do this using the runtime library routines and constants. (See "Writing portable code" on page 15-16 for information.)

7 Find equivalent functionality for features that are different on Linux. Use conditional compiler directives such as $IFDEF (Delphi) or #ifdefs (C++) (sparingly) to delimit Windows-specific information. (See "Using conditional directives" on page 15-18 for information.)

For example, you can use conditional compiler directives for platform-specific code in your source files:

Delphi example

```
{$IFDEF MSWINDOWS}
  IniFile.LoadfromFile('c:\x.txt');
{$ENDIF}

{$IFDEF LINUX}
  IniFile.LoadfromFile('/home/name/x.txt');
{$ENDIF}
```

C++ example

```
#ifdef WINDOWS //If using a C++ compiler, use __WIN32__
  IniFile->LoadfromFile("c:\\x.txt");
#endif

#ifdef __linux__
  IniFile->LoadfromFile("/home/name/x.txt");
#endif
```

8 Search for references to pathnames in all the project files.

- Pathnames in Linux use a forward slash / as a delimiter (for example, /usr/lib) and files may be located in different directories on the Linux system. Use the PathDelim constant (in SysUtils) to specify the path delimiter that is appropriate for the system. Determine the correct location for any files on Linux.
- Change references to drive letters (for example, C:\) and code that looks for drive letters by looking for a colon at position 2 in the string. Use the DriveDelim constant (in SysUtils) to specify the location in terms that are appropriate for the system.
- In places where you specify multiple paths, change the path separator from semicolon (;) to colon (:). Use the PathSep constant (in SysUtils) to specify the path separator that is appropriate for the system.
- Because file names are case-sensitive in Linux, make sure that your application doesn't change the case of file names or assume a certain case.

9 Compile, test, and debug your application in Kylix. You will receive warnings on Windows-specific features that are in use.

You can also port a Kylix application to Windows:

1 Move the application source files to Windows.

2 Recompile and test the application on Windows using Delphi or C++Builder.

CLX class library

Kylix applications use the Borland Component Library for Cross-Platform (CLX). CLX is a class library made up of sublibraries and their classes that you use when developing cross-platform applications. CLX classes are grouped into the following sublibraries:

Table 15.2 CLX libraries

Part	Description
BaseCLX	Low-level classes and routines available for all CLX applications. BaseCLX includes the CLX Runtime Library up to and including the Classes unit.
DataCLX	Client data-access components. These components are used in applications that access databases. They can access data from a file on disk or from a database server using dbExpress.

Table 15.2 CLX libraries

Part	Description
NetCLX	Components for building Web Server applications. These include support for applications that use Apache or CGI Web Servers.
VisualCLX	GUI components and graphics classes. VisualCLX classes make use of an underlying widget library (Qt).
WinCLX	Classes that are available only on the Windows platform. These include controls that are wrappers for Native Windows controls, database access components that use mechanisms (such as the Borland Database Engine or ADO) that are not available on Linux, and components that support Windows-only technologies (such as COM, NT Services, or control panel applets). WinCLX is not available in Kylix.

Within WinCLX, many controls provide an easy way to access Windows controls by making calls into the Windows API libraries. Similarly, on Linux, VisualCLX provides access to Qt widgets by making calls into the Qt shared libraries. Kylix includes VisualCLX only.

VisualCLX looks much like the WinCLX. Most of the components and properties have the same names. In addition, VisualCLX, as well as WinCLX, is available on .

Widgets in VisualCLX replace Windows controls. For example, *TWidgetControl* in CLX replaces *TWinControl* in WinCLX. Other WinCLX components (such as *TScrollingWinControl*) have corresponding names in VisualCLX (such as *TScrollingWidget*). However, you do not need to change occurrences of *TWinControl* to *TWidgetControl*. Class declarations, such as the following:

```
TWinControl = TWidgetControl;
```

```
TWinControl : TWidgetControl;
```

appear in the QControls unit file to simplify sharing of source code. *TWidgetControl* and all its descendants have a *Handle* property that references the Qt object and a *Hooks* property that references the hook object that handles the event mechanism.

Unit names and locations of some classes are different in CLX on Kylix. You will need to modify the **uses** clauses (Delphi) or the header file (C++) you include in your source files to eliminate references to units that don't exist in Kylix and to change the names to CLX units. Most project files and the interface sections of most units contain a uses clause (header files in C++). In Delphi, the implementation section of a unit can also contain its own uses clause.

See "Understanding the class libraries" on page 3-1 .

What VisualCLX does differently

Although much of VisualCLX is implemented so that it is consistent with WinCLX, some features are implemented differently. This section provides an overview of some of the differences between WinCLX and VisualCLX implementations.

Look and feel

The visual environment in Linux looks somewhat different than it does in Windows. The look of dialogs may differ depending on which window manager you are using, such as KDE or Gnome.

Styles

Application-wide styles can be used in addition to the *OwnerDraw* properties. You can use *TApplication*'s *Style* property to specify the look and feel of an application's graphical elements. Using styles, a widget or an application can take on a whole new look. You can still use owner draw on Linux but using styles is recommended.

Variants

All of the variant/safe array code that was in the System unit is in the Variants and VarUtils units.

The operating system dependent code is now isolated in the VarUtils unit, and it also contains generic versions of everything needed by the Variants unit. If you are porting code from Windowsthat includes Windows calls, you need to replace these calls to calls into the VarUtils unit.

If you want to use variants, you must include the Variants unit to your uses clause (Delphi) or the header file in your source file (C++).

VarIsEmpty does a simple test against *varEmpty* to see if a variant is clear, and on Linux you can use the *VarIsClear* function see whether the value of the variant is undefined.

D

Custom variant data handler in Delphi

In Delphi, you can define custom data types for variants. This introduces operator overloading while the type is assigned to the variant. To create a new variant type, descend from the class, *TCustomVariantType*, and instantiate your new variant type.

For an example, see VarCmplx.pas. This unit implements complex mathematics support via custom variants. It supports the following variant operations: addition, subtraction, multiplication, division (not integer division), and negation. It also handles conversion to and from: SmallInt, Integer, Single, Double, Currency, Date, Boolean, Byte, OleStr, and String. Any of the float/ordinal conversion will lose any imaginary portion of the complex value.

Registry

Linux does not use a registry to store configuration information. Instead, you use text configuration files and environment variables instead of using the registry. System configuration files on Linux are often located in /etc, such as /etc/hosts. Other user profiles are located in hidden files (preceded with a dot), such as .bashrc, which holds bash shell settings or .XDefaults, which is used to set defaults for X programs.

Registry-dependent code may be changed to using a local configuration text file instead. Settings that users can change must be saved in their home directory so that they have permission to write to it. Configuration options that need to be set by the root are stored in /etc. Writing a unit containing all the registry functions but diverting all output to a local configuration file is one way you could handle a former dependency on the registry.

To place information in a global location on Linux, you can store a global configuration file in the /etc directory or the user's home directory as a hidden file. Therefore, all of your applications can access the same configuration file. However, you must be sure that the file permissions and access rights are set up correctly.

You can also use .ini files in cross-platform applications. However, in Kylix, you need to use *TMemIniFile* instead of *TRegIniFile.*

Sharing methods

Sharing methods between languages requires use of a common application binary interface (ABI). Kylix virtual method tables (VMTs) conform to the currently proposed UNIX ABI specification.

To make gcc generate conforming VMTs for C++ objects, you must use gcc version 2.95 or later and specify the COM_INTERFACE attribute in the C++ class declaration. Without this attribute, gcc may insert information that changes the VMT offsets and causes Kylix calls to C++ methods fail.

In addition, the Kylix interface or class declaration must use the same calling convention as the C++ class declaration. The recommended calling convention is stdcall, though Kylix also supports cdecl.

Other differences

The VisualCLX implementation also has some other differences from the WinCLX that affect the way your components work. This section describes some of those differences.

- You can select a VisualCLX component from the Component palette and click either the left or right mouse button to add it to a form. For a WinCLX component, you can click the left mouse button only.
- The VisualCLX *TButton* control has a *ToggleButton* property that the equivalent WinCLX control doesn't have.
- In VisualCLX, *TColorDialog* does not have an *Options* property to set. Therefore, you cannot customize the appearance and functionality of the color selection dialog. Also, depending on which window manager you are using in Linux, *TColorDialog* is not always modal or nonresizable. On Windows, *TColorDialog* is always modal and nonresizable.
- At runtime, combo boxes work differently in VisualCLX than they do in WinCLX. In VisualCLX (but not in the WinCLX), you can add an item to a drop-down list by entering text and pressing *Enter* in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to *ciNone*. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key when the edit box is closed, it does not stop at the last item of the combo box list. It cycles around to the top again.
- The key values used in events can be different between the WinCLX and VisualCLX. For example, the Enter key has a value of 13 on the WinCLX and a

value of 4100 on VisualCLX. If you hard code key values in your VisualCLX applications, you need to change these values when porting from Windows to Linux or vice versa.

In general, the classes between WinCLX and VisualCLX function the same. However, some classes are missing certain properties, methods, or events in VisualCLX:

- Bi-directional properties (*BidiMode*) for right-to-left text output or input.
- Generic bevel properties on common controls (note that some objects still have bevel properties).
- Docking properties and methods.
- Backward compatibility components such as those on the Win3.1 tab and *Ctl3D.*
- *DragCursor* and *DragKind* (but drag and drop is included).

Additional differences exist. Refer to the CLX online documentation for details on all of the CLX objects or in editions of Kylix that include the source code, located in {install directory}/kylix/source/clx.

Features that do not port directly or are missing

In general, the functionality between Windows and cross-platform applications is the same. However, some Windows-specific features do not port directly to Linux environments. For example, ActiveX, ADO, BDE, COM, and OLE are dependent on Windows technology and not available in Kylix. The following table lists features that are different on the two platforms and lists the equivalent Linux or VisualCLX feature, if one is available.

Table 15.3 Changed or different features

Windows/WinCLX feature	Linux/VisualCLX feature
ADO components	Regular database components
Automation Servers	Not available
BDE	dbExpress and regular database components
COM+ components (including ActiveX)	Not available
DataSnap	Functionality for Web Services only.
FastNet	Not available
Legacy components (such as items on the Win 3.1 Component palette tab)	Not available
Messaging Application Programming Interface (MAPI) includes a standard library of Windows messaging functions.	SMTP and POP3 let you send, receive, and save e-mail messages
Quick Reports	Not available
Windows API calls	CLX methods, Qt calls, libc calls, or calls to other system libraries

Table 15.3 Changed or different features (continued)

Windows/WinCLX feature	Linux/VisualCLX feature
Windows messaging	Qt events
Winsock	BSD sockets

Other features not supported or supported differently on Kylix include:

- The Linux equivalent of Windows DLLs are shared object libraries (.so files), which contain position-independent code (PIC). Thus, global memory references and calls to external functions are made relative to the EBX register, which must be preserved across calls. In Delphi, this means that variables referring to an absolute address in memory (using the absolute directive) are not allowed. You only need to worry about global memory references and calls to external functions if using assembler—Kylix generates the correct code. (For information, see "Including inline assembler code" on page 15-21.)
- In Kylix applications, library modules and packages are implemented using .so files.
- In Delphi applications, absolute addresses are used in variable declarations. You can use the absolute directive to refer to the name of another variable; for example:

  ```
  var Var2: Byte absolute Var1;
  ```
- In C++:
  - In the IDE, you cannot open a ported Windows project; you must create a new Kylix project and add your source files.
  - There is no equivalent to the Windows WinMain and DLLMain functions. The main source file for a package contains no default entry point (DllEntryPoint). The main source file must have a #define SharedObject directive in it.
- Borland's make utility. Use the GNU make utility instead.
- TASM is not supported. You cannot import external assembler routines unless they use syntax supported by an assembler such as NASM, the Netwide Assembler, one of the free, portable x86 assemblers supported by Kylix.
- Resource introspection is not supported. Applications must know at compile time the names of all resources they will use. Resources cannot be browsed dynamically.

Windows-only and cross-platform unit comparison

All of the objects in CLX are defined in unit (Delphi) or header (C++) files. For example, you can find the implementation of *TObject* in the System unit and the base *TComponent* class defined in the Classes unit. When you drop an object onto a form or use an object within your application, the name of the unit is added to the uses clause (Delphi) or the header file included in your source file (C++), which tells the compiler which units to link into the project.

Some of the units that are in Windows-only applications are also in cross-platform applications, such as Classes, DateUtils, DB, System, SysUtils and many more units such as those in the runtime library (RTL). However, many of the cross-platform units are in the VisualCLX sublibrary and are different from those in the WinCLX (Windows-only) sublibrary. If you are porting applications from Windows to Linux, you'll have to change the names of these units in the uses clause (Delphi) or header file (C++) of your application. The most common name change is made by adding a Q to the beginning of the unit or header file name.

Table 15.4 lists the names of Windows-only (WinCLX) units that have different names in cross-platform (VisualCLX) units. Units that are either the same in both Windows-only and cross-platform applications or are third-party units are not listed.

Table 15.4 Windows-only and equivalent cross-platform units

Windows-only units	Cross-platform units
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Controls	QControls
DBActns	QDBActns
DBCtrls	QDBCtrls
DBGrids	QDBGrids
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Mask	QMask
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdActns	QStdActns
StdCtrls	QStdCtrls
VclEditors	ClxEditors

Some Windows units are not included in Kylix because they are used for Windows-specific features that are not available on Linux. For example, Kylix does not use ADO units, BDE units, COM units, or Windows units such as CtlPanel, Messages, Registry, and Windows.

References to these units and the classes within these units must be eliminated from applications you want to run on Linux. If you try to compile a program with units that do not exist in Kylix, you will receive the following error message:

```
File not found: 'unitname.dcu'
Unable to open include file: 'unitname.hpp'
```

Delete that unit from the uses clause (Delphi) or header file (C++) and try again.

Note Which components, and therefore units, are included depend on which edition of Kylix you have.

Differences in CLX object constructors

When a CLX object is created, either implicitly by placing that object on the form or explicitly in code by using the object's *Create* method (Delphi) or constructor (C++), an instance of the underlying associated widget is created also. The CLX object owns this instance of the widget. When the CLX object is deleted, the underlying widget is also deleted. In Delphi, the object is deleted by calling the *Free* method or automatically deleted by the CLX object's parent container.

When you explicitly create a CLX object in your code by calling into the Qt interface library such as QWidget_Create(), you are creating an instance of a Qt widget that is not owned by a CLX object. This passes the instance of an existing Qt widget to the CLX object to use during its construction. This CLX object does not own the Qt widget that is passed to it. Therefore, after creating the object in this manner, only the CLX object is destroyed and not the underlying Qt widget instance. This is different from WinCLX on Windows-only applications.

A few CLX graphics objects, such as *TBrush* and *TPen*, let you assume ownership of the underlying widget using the *OwnHandle* method. After calling *OwnHandle*, if you delete the CLX object, the underlying widget is destroyed as well.

Some property assignments in CLX have moved from the *Create* method (Delphi) or constructor (C++) to *InitWidget*. This allows delayed construction of the Qt object until it's really needed. For example, say you have a property named *Color*. In *SetColor*, you can check with *HandleAllocated* to see if you have a Qt handle. If the handle is allocated, you can make the proper call to Qt to set the color. If not, you can store the value in a private field variable, and, in *InitWidget*, you set the property.

For more information about object construction in C++, see "Construction of CLX-style objects in C++" on page 14-10.

Handling system and widget events

System and widget events, which are mainly of concern when writing components, are handled differently by WinCLX and VisualCLX. The most important difference is that VisualCLX controls do not respond directly to Windows messages, even when running on Windows (see Chapter 41, "Handling system notifications.") Instead, they respond to notifications from the underlying widget layer. Because the notifications use a different system, the order and timing of events can sometimes differ between corresponding WinCLX and VisualCLX objects. This difference occurs even if your cross-platform application is running on Windows rather than Linux. If you are porting a Windows application to Linux, you may need to change the way your event handlers respond to accommodate these differences.

For information on writing components that respond to system and widget events (other than those that are reflected in the published events of CLX components), see "Responding to system notifications using CLX" on page 41-1.

Sharing source files between Windows and Linux

If you want your application to run on both Windows and Linux, you can share the source files making them accessible to both operating systems. You can do this in several ways, such as placing the source files on a server that is accessible to both computers or by using Samba on the Linux machine to provide access to files through Microsoft network file sharing for both Linux and Windows. You can choose to keep the source on Linux and create a shared drive on Linux. Or you can keep the source on Windows and create a share on Windows for the Linux machine to access.

You can continue to develop and compile the file onWindows using objects that are supported by CLX. When you are finished, you can compile on both Linux and Windows.

Form files (.dfm files on Windows-only applications) are called .xfm files in Kylix. In Kylix, if you create a new application, the IDE creates an .xfm form file instead of a .dfm file. If you want to single-source your code, you should copy the .dfm from Windows to an .xfm on Linux, maintaining both files. Otherwise, the .dfm file will be modified on Linux and may no longer work on Windows. If you plan to write cross-platform applications, the .xfm will work on editions of Delphi that support CLX.

Environmental differences between Windows and Linux

Currently, cross-platform means an application that can compile virtually unchanged on both the Windows and Linux operating systems. The following table lists some of the differences between Linux and the Windows operating environments.

Table 15.5 Differences in the Linux and Windows operating environments

Difference	Description
File name case sensitivity	In Linux, file names are case sensitive. The file Test.txt is *not* the same file as test.txt. You need to pay close attention to capitalization of file names on Linux.
Line ending characters	On Windows, lines of text are terminated by CR/LF (that is, ASCII 13 + ASCII 10), but on Linux it is LF. While the Code editor can handle the difference, you should be aware of this when importing code from Windows.
End of file character	In MS-DOS and Windows, the character value #26 (Ctrl-Z) is treated as the end of the text file, even if there is data in the file after that character. Linux uses Ctrl+D as the end-of-file character.

Table 15.5 Differences in the Linux and Windows operating environments (continued)

Difference	Description
Batch files/shell scripts	The Linux equivalent of .bat files are shell scripts. A script is a text file containing instructions, saved and made executable with the command, `chmod +x <scriptfile>`. The scripting language depends on the shell you are using on Linux. Bash is commonly used.
Command confirmation	In MS-DOS or Windows, if you try to delete a file or folder, it asks for confirmation ("Are you sure you want to do that?"). Generally, Linux won't ask; it will just do it. This makes it easy to accidentally destroy a file or the entire file system. There is no way to undo a deletion on Linux unless a file is backed up on another media.
Command feedback	If a command succeeds on Linux, it redisplays the command prompt without a status message.
Command switches	Linux uses a dash (-) to indicate command switches or a double dash (--) for multiple character options where DOS uses a slash (/) or dash (-).
Configuration files	On Windows, configuration is done in the registry or in files such as autoexec.bat. On Linux, configuration files are created as hidden files in the user's home directory. Configuration files in the /etc directory are usually not hidden files. Linux also uses environment variables such as LD_LIBRARY_PATH (search path for libraries). Other important environment variables: HOME Your home directory (/home/sam) TERM Terminal type (xterm, vt100, console) SHELL Path to your shell (/bin/bash) USER Your login name (sfuller) PATH List to search for programs They are specified in the shell or in files such as .bashrc.
DLLs/Shared object files	On Linux, you use shared object files (.so). In Windows, these are dynamic link libraries (DLLs).
Drive letters	Linux doesn't have drive letters. An example Linux pathname is /lib/security. See DriveDelim in the runtime library.
Exceptions	Operating system exceptions are called signals on Linux.
Executable files	On Linux, executable files require no extension. On Windows, executable files have an exe extension.
File name extensions	Linux does not use file name extensions to identify file types or to associate files with applications.
File permissions	On Linux, files (and directories) are assigned read, write, and execute permissions for the file owner, group, and others. For example, `-rwxr-xr-x` means, from left to right: `-` is the file type (`-` = ordinary file, `d` = directory, `l` = link); `rwx` are the permissions for the file owner (read, write, execute); `r-x` are the permissions for the group of the file owner (read, execute); and `r-x` are the permissions for all other users (read, execute). The root user (superuser) can override these permissions. You need to make sure that your application runs under the correct user and has proper access to required files.

Table 15.5 Differences in the Linux and Windows operating environments (continued)

Difference	Description
Make utility	Borland's make utility is not available on the Linux platform. Instead, you can use Linux's GNU make utility.
Multitasking	Linux fully supports multitasking. You can run several programs (in Linux, called processes) at the same time. You can launch processes in the background (using & after the command) and continue working straight away. Linux also lets you have several sessions.
Pathnames	Linux uses a forward slash (/) wherever DOS uses a backslash (\). A PathDelim constant can be used to specify the appropriate character for the platform. See PathDelim in the runtime library.
Search path	When executing programs, Windows always checks the current directory first, then looks at the PATH environment variable. Linux never looks in the current directory but searches only the directories listed in PATH. To run a program in the current directory, you usually have to type ./ before it. You can also modify your PATH to include ./ as the first path to search.
Search path separator	Windows uses the semicolon as a search path separator. Linux uses a colon. See PathDelim in the runtime library.
Symbolic links	On Linux, a symbolic link is a special file that points to another file on disk. Place symbolic links in the global bin directory that points to your application's main files and you don't have to modify the system search path. A symbolic link is created with the ln (link) command. Windows has shortcuts for the GUI desktop. To make a program available at the command line, Windows install programs typically modify the system search path.

Directory structure on Linux

Directories are different in Linux. Any file or device can be mounted anywhere on the file system.

Note Linux pathnames use forward slashes whereas Windows pathnames use backslashes. The initial slash stands for the root directory.

Following are some of the commonly used directories in Linux.

Table 15.6 Common Linux directories

Directory	Contents
/	The root or top directory of the entire Linux file system
/root	The root file system; the Superuser's home directory
/bin	Commands, utilities
/sbin	System utilities
/dev	Devices shown as files
/lib	Libraries
/home/username	Files owned by the user where username is the user's login name.
/opt	Optional
/boot	Kernel that gets called when the system starts up

Table 15.6 Common Linux directories (continued)

Directory	Contents
/etc	Configuration files
/usr	Applications, programs. Usually includes directories like /usr/spool, /usr/man, /usr/include, /usr/local
/mnt	Other media mounted on the system such as a CD or a floppy disk drive
/var	Logs, messages, spool files
/proc	Virtual file system and reporting system statistics
/tmp	Temporary files

Note Different distributions of Linux sometimes place files in different locations. A utility program may be placed in /bin in a Red Hat distribution but in /usr/local/bin in a Debian distribution.

Refer to www.pathname.com for additional details on the organization of the UNIX/ Linux hierarchical file system and to read the *Filesystem Hierarchy Standard*.

Writing portable code

If you are writing cross-platform applications that are meant to run on both Windows and Linux, you can write code that compiles under different conditions. Using conditional compilation, you can maintain your Windows coding, yet also make allowances for Linux operating system differences.

To create applications that are easily portable between Windows and Linux, remember to:

- Reduce or isolate calls to platform-specific (Win32 or Linux) APIs; use CLX methods or calls to the Qt library.
- Eliminate Windows messaging (PostMessage, SendMessage) constructs within an application. In CLX, call the *QApplication_postEvent* and *QApplication_sendEvent* methods instead. For information on writing components that respond to system and widget events, see "Responding to system notifications using CLX" on page 41-1.
- Use *TMemIniFile* instead of *TRegIniFile*.
- Observe and preserve case-sensitivity in file and directory names.
- Port any external assembler TASM code. The GNU assembler, "as," does not support the TASM syntax. (See "Including inline assembler code" on page 15-21.)

Try to write the code to use platform-independent runtime library routines and use constants found in System, SysUtils, and other runtime library units. For example, use the PathDelim constant to insulate your code from '/' versus '\' platform differences.

Another example involves the use of multibyte characters on both platforms. Windows code traditionally expects only two bytes per multibyte character. In Linux, multibyte character encoding can have many more bytes per char (up to six bytes for

UTF-8). Both platforms can be accommodated using the StrNextChar function in SysUtils.

Delphi example

In Delphi, existing Windows code such as:

```
while p^ <> #0 do
begin
   if p^ in LeadBytes then
      inc(p);
   inc(p);
end;
```

can be replaced with platform-independent code like this:

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    p := StrNextChar(p)
  else
    inc(p);
end;
```

C++ example

In C++, existing Windows code such as:

```
while(*p != 0)
{
   if(LeadBytes.Contains(*p))
      p++;
   p++;
}
```

can be replaced with platform-independent code like this:

```
while(*p != 0)
{
   if(LeadBytes.Contains(*p))
     p = StrNextchar(p);
   else
     p++;
}
```

The previous example is platform-portable but still avoids the performance cost of a procedure call for non-multibyte locales.

If using runtime library functions is not a workable solution, try to isolate the platform-specific code in your routine into one chunk or into a subroutine. Try to limit the number of conditional compiler directive (**$IFDEF/#ifdef**) blocks to maintain source code readability and portability. The conditional symbol WIN32 is not defined on Linux. The conditional symbol LINUX is defined, indicating the source code is being compiled for the Linux platform.

Using conditional directives

Using conditional compiler directives such as $IFDEF (Delphi)/#ifdef (C++) is a reasonable way to conditionalize your code for the Windows and Linux platforms. However, because conditional compiler directives make source code harder to understand and maintain, you need to understand when it is reasonable to use them. When considering the use of conditional compiler directive, the top questions should be "Why does this code require a conditional compiler directive?" and "Can this be written without a conditional compiler directive?"

Follow these guidelines for using conditional compiler directives within cross-platform applications:

- Try not to use $IFDEFs/#ifdefs unless absolutely necessary. $IFDEFs/#ifdefs in a source file are only evaluated when source code is compiled. Unlike C/C++, Delphi does not require unit sources (header files) to compile a project. Full rebuilds of all source code is an uncommon event for most Kylix projects.
- Do not use $IFDEFs/#ifdefs in package files. Limit their use to source files. Component writers need to create two design-time packages when doing cross-platform development, not one package using $IFDEFs/#ifdefs.
- In general, use $IFDEF MSWINDOWS (#ifdef WINDOWS in C++) to test for any Windows platform including WIN32. Reserve the use of $IFDEF WIN32 (#ifdef WIN32 in C++) for distinguishing between specific Windows platforms, such as 32-bit versus 64-bit Windows. Don't limit your code to WIN32 unless you know for sure that it will not work in WIN64.
- Avoid negative tests like $IFNDEF/#ifndef unless absolutely required. $IFNDEF LINUX (#ifndef __linux__ in C++) is not equivalent to $IFDEF MSWINDOWS (#ifdef WINDOWS in C++).
- Avoid $IFNDEF/$ELSE (#ifndef/#else in C++) combinations. Use a positive test instead ($IFDEF/#ifdef) for better readability.
- Avoid $ELSE/#else clauses on platform-sensitive $IFDEF/#ifdefs. Use separate $IFDEF/#ifdef blocks for Linux- and Windows-specific code instead of $IFDEF LINUX/$ELSE (#ifdef __linux__/#else in C++) or $IFDEF MSWINDOWS/$ELSE (#ifdef WINDOWS/#else in C++).

  For example, old code may contain:

Delphi example

```
{$IFDEF WIN32}
   (32-bit Windows code)
{$ELSE}
   (16-bit Windows code)   //!! By mistake, Linux could fall into this code.
{$ENDIF}
```

C++ example

```
#ifdef WIN32
   (32-bit Windows code)
```

```
#else
   (16-bit Windows code)   //!! By mistake, Linux could fall into this code.
#endif
```

For any non-portable code in $IFDEFs/#ifdefs, it is better for the source code to fail to compile than to have the platform fall into an $ELSE/#else clause and fail mysteriously at runtime. Compile failures are easier to find than runtime failures.

- Use the $IF/#if syntax for complicated tests. Replace nested $IFDEFs/#ifdefs with a boolean expression in an $IF/#if directive. You should terminate the $IF/#if directive using $IFEND (#endif in C++), not $ENDIF. This allows you to place $IF/#if expressions within $IFDEFs/#ifdefs to hide the new $IF/#if syntax from previous compilers.

All of the conditional directives are documented in the online Help. Also, see the topic "conditional directives" in Help for more information.

Terminating Delphi conditional directives

In Delphi, use the $IFEND directive to terminate $IF and $ELSEIF conditional directives. This allows $IF/$IFEND blocks to be hidden from older compilers inside of using $IFDEF/$ENDIF. Older compilers won't recognize the $IFEND directive. $IF can only be terminated with $IFEND. You can only terminate old-style directives ($IFDEF, $IFNDEF, $IFOPT) with $ENDIF.

Note When nesting an $IF inside of $IFDEF/$ENDIF, do not use $ELSE with the $IF. Older compilers will see the $ELSE and think it is part of the $IFDEF, producing a compile error down the line. You can use {$ELSEIF True} as a substitute for {$ELSE} in this situation, since the $ELSEIF won't be taken if the $IF is taken first, and the older compilers won't know $ELSEIF. Hiding $IF for backwards compatibility is primarily an issue for third party vendors and application developers who want their code to run on several different versions.

$ELSEIF is a combination of $ELSE and $IF. The $ELSEIF directive allows you to write multi-part conditional blocks where only one of the conditional blocks will be taken. For example:

```
{$IFDEF doit}
   do_doit
{$ELSEIF  RTLVersion >= 14}
    goforit
{$ELSEIF  somestring = 'yes'}
    beep
{$ELSE}
    last chance
{$IFEND}
```

Of these four cases, only one is taken. If none of the first three conditions is true, the $ELSE clause is taken. $ELSEIF must be terminated by $IFEND. $ELSEIF cannot appear after $ELSE. Conditions are evaluated top to bottom like a normal $IF...$ELSE sequence. In the example, if doit is not defined, then RTLVersion is 15 and somestring is 'yes.' Only the "goforit" block is taken and not the "beep" block, even though the conditions for both are true.

If you forget to use an $ENDIF to end one of your $IFDEFs, the compiler reports the following error message at the end of the source file:

```
Missing ENDIF
```

If you have more than a few $IF/$IFDEF directives in your source file, it can be difficult to determine which one is causing the problem. The Delphi IDE reports the following error message on the source line of the last $IF/$IFDEF compiler directive with no matching $ENDIF/$IFEND:

```
Unterminated conditional directive
```

You can start looking for the problem at that location.

Emitting messages

The **$MESSAGE** (Delphi) or **#pragma message** (C++) compiler directive allows source code to emit hints (Delphi), warnings, and errors just as the compiler does.

Delphi example

```
{$MESSAGE  HINT|WARN|ERROR|FATAL 'text string' }
```

The message type is optional. If no message type is indicated, the default is HINT. The text string is required and must be enclosed in single quotes.

For example:

```
{$MESSAGE 'Boo!'} // emits a hint.
{$Message Hint 'Feed the cats'}  //emits a hint.
{$Message Warn 'Looks like rain.'} //emits a warning.
{$Message Error 'Not implemented'} //emits an error, continues compiling.
{$Message Fatal 'Bang. Yer dead.'} //emits an error, terminates the compiler.
```

C++ example

In C++, use #pragma message to specify a user-defined message within your program code, using one of the following formats.

If you have a variable number of string constants, use:

```
#pragma message( "hi there" )
#pragma message( "hi" " there" )
```

To write text following a message, use:

```
#pragma message text
```

To expand a previously defined value, use:

```
#pragma message (text)
```

```
#define text "a test string"
#pragma message (text)
```

For example, to display these messages on a button, use:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    #pragma message( "hi there 1" )
    #pragma message( "hi " "there 2" )

    #pragma message hi there 3

    #define text "a test string"
    #pragma message (text)
}
```

In C++, to display the messages in the IDE, choose Projects | Options | Compiler, click the Compiler tab, and check the Show general messages check box.

Including inline assembler code

If you include inline assembler code in your Windows applications, you may not be able to use the same code on Linux because of position-independent code (PIC) requirements on Linux. Linux shared object libraries (DLL equivalents) require that all code be relocatable in memory without modification. This primarily affects inline assembler routines that use global variables or other absolute addresses, or that call external functions.

For units that contain only Delphi or C++ code, the compiler automatically generates PIC when required. It's a good idea to compile every unit (Delphi) or source file (C++) into both PIC and non-PIC formats; use the -p (Delphi) or VP (C++) compiler switch to generate PIC.

D In Delphi, precompiled units are available in both PIC and non-PIC formats. In Delphi, PIC units have a .dpu extension (instead of .dcu).

You may want to code assembler routines differently depending on whether you'll be compiling to an executable or a shared library; use {**$IFDEF** PIC} (Delphi) or **#ifdef**__PIC__ (C++) to branch the two versions of your assembler code. Or you can consider rewriting the routine in Delphi or C++ to avoid the issue.

Following are the PIC rules for inline assembler code:

- PIC requires all memory references be made relative to the EBX register, which contains the current module's base address pointer (in Linux called the Global Offset Table or GOT). So, instead of

  ```
  MOV EAX,GlobalVar
  ```

  use

  ```
  MOV EAX,[EBX].GlobalVar
  ```

- PIC requires that you preserve the EBX register across calls into your assembly code (same as on Win32), and also that you restore the EBX register *before* making calls to external functions (different from Win32).
- While PIC code will work in base executables, it may slow the performance and generate more code. You don't have any choice in shared objects, but in executables you probably still want to get the highest level of performance that you can.

Programming differences on Linux

The Linux wchar_t widechar is 32 bits per character. The 16-bit Unicode standard that CLX supports (widechars in Delphi) is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. widechar data (Delphi) or WideString types (C++) must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

In Linux, WideStrings are reference counted like long strings (in Windows, they're not).

In Windows, multibyte characters (MBCS) are represented as one- and two-byte char codes. In Linux, they are represented in one to six bytes.

AnsiStrings can carry multibyte character sequences, depending upon the user's locale settings. The Linux encoding for multibyte characters such as Japanese, Chinese, Hebrew, and Arabic may not be compatible with the Windows encoding for the same locale. Unicode is portable, whereas multibyte is not. See "Enabling application code" on page 17-2 for details on handling strings for various locales in international applications.

D In Linux, you cannot use variables on absolute addresses. The Delphi syntax:

```
var X: Integer absolute $1234;
```

is not supported in PIC and is not allowed.

Cross-platform database applications

On Windows, there are several choices for how to access database information. These include using ADO, the Borland Database Engine (BDE), and InterBase Express. These three choices are not available on Kylix, however. On Windows and Linux, you can use *dbExpress*, a cross-platform data access technology, depending on which edition of Kylix you have.

Before you port a database application to *dbExpress* so that it will run on Linux, you should understand the differences between using *dbExpress* and the data access mechanism you were using. These differences occur at different levels.

- At the lowest level, there is a layer that communicates between your application and the database server. This could be ADO, the BDE, or the InterBase client software. This layer is replaced by *dbExpress*, which is a set of lightweight drivers for dynamic SQL processing.
- The low-level data access is wrapped in a set of components that you add to data modules or forms. These components include database connection components, which represent the connection to a database server, and datasets, which represent the data fetched from the server. Although there are some very important differences, due to the unidirectional nature of *dbExpress* cursors, the differences are less pronounced at this level, because datasets all share a common ancestor, as do database connection components.

- At the user-interface level, there are the fewest differences. CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. The major differences at the user interface level arise from changes needed to accommodate the use of cached updates.

For information on porting existing database applications to *dbExpress*, see "Porting database applications to Linux" on page 15-25. For information on designing new *dbExpress* applications, see Chapter 19, "Designing database applications."

dbExpress differences

On Linux, *dbExpress* manages the communication with database servers. *dbExpress* consists of a set of lightweight drivers that implement a set of common interfaces. Each driver is a shared object (.so file) that must be linked to your application. Because *dbExpress* is designed to be cross-platform, it is also available on Windows as a set of dynamic-link libraries (.dlls).

As with any data-access layer, *dbExpress* requires the client-side software provided by the database vendor. In addition, it uses a database-specific driver, plus two configuration files, dbxconnections and dbxdrivers. This is markedly less than you need for, say, the BDE, which requires the main Borland Database Engine library (Idapi32.dll) plus a database-specific driver and a number of other supporting libraries.

There are other differences between *dbExpress* and the other data-access layers from which you need to port your application. For example, *dbExpress*:

- *A*llows for a simpler and faster path to remote databases. As a result, you can expect a noticeable performance increase for simple, straight-through data access.
- Processes queries and stored procedures, but does not support the concept of opening tables.
- Returns only unidirectional cursors.
- Has no built-in update support other than the ability to execute an INSERT, DELETE, or UPDATE query.
- Does no metadata caching; the design time metadata access interface is implemented using the core data-access interface.
- Executes only queries requested by the user, thereby optimizing database access by not introducing any extra queries.
- Manages a record buffer or a block of record buffers internally. This differs from the BDE, where clients are required to allocate the memory used to buffer records.
- *S*upports only local tables that are SQL-based, such as InterBase and Oracle.
- Uses drivers for DB2, Informix, InterBase, MySQL, Oracle, and PostgreSQL. If you are using a different database server, you must either convert your data to one of these databases, write a *dbExpress* driver for the database server you are using, or obtain a third-party *dbExpress* driver for your database server.

Component-level differences

When you write a *dbExpress* application, it requires a different set of data access components than those used in your existing database applications. The *dbExpress* components share the same base classes as other data access components (*TDataSet* and *TCustomConnection*), which means that many of the properties, methods, and events are the same as the components used in your existing applications.

Table 15.7 lists some of the important database components used in InterBase Express, BDE, and ADO in the Windows environment and shows the comparable *dbExpress* components for use on Linux and in cross-platform applications.

Table 15.7 Comparable data-access components

InterBase Express components	BDE components	ADO components	dbExpress components
TIBDatabase	*TDatabase*	*TADOConnection*	*TSQLConnection*
TIBTable	*TTable*	*TADOTable*	*TSQLTable*
TIBQuery	*TQuery*	*TADOQuery*	*TSQLQuery*
TIBStoredProc	*TStoredProc*	*TADOStoredProc*	*TSQLStoredProc*
TIBDataSet		*TADODataSet*	*TSQLDataSet*

The *dbExpress* datasets (*TSQLTable*, *TSQLQuery*, *TSQLStoredProc*, and *TSQLDataSet*) are more limited than their counterparts, however, because they do not support editing and only allow forward navigation. For details on the differences between the *dbExpress* datasets and the other datasets that are available on Windows, see Chapter 24, "Using unidirectional datasets."

Because of the lack of support for editing and navigation, most *dbExpress* applications do not work directly with the *dbExpress* datasets. Rather, they connect the *dbExpress* dataset to a client dataset, which buffers records in memory and provides support for editing and navigation. For more information about this architecture, see "Database architecture" on page 19-4.

Note For very simple applications, you can use *TSQLClientDataSet* instead of a *dbExpress* dataset connected to a client dataset. This has the benefit of simplicity, because there is a 1:1 correspondence between the dataset in the application you are porting and the dataset in the ported application, but it is less flexible than explicitly connecting a *dbExpress* dataset to a client dataset. For most applications, it is recommended that you use a *dbExpress* dataset connected to a *TClientDataSet* component.

User interface-level differences

CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. As a result, porting the user interface portion of your database applications introduces few additional considerations beyond those involved in porting any Windows application to CLX.

The major differences at the user interface level arise from differences in the way *dbExpress* datasets or client datasets supply data.

If you are using only *dbExpress* datasets, then you must adjust your user interface to accommodate the fact that the datasets do not support editing and only support forward navigation. Thus, for example, you may need to remove controls that allow users to move to a previous record. Because *dbExpress* datasets do not buffer data, you can't display data in a data-aware grid: only one record can be displayed at a time.

If you have connected the *dbExpress* dataset to a client dataset, then the user interface elements associated with editing and navigation should still work. You need only reconnect them to the client dataset. The main consideration in this case is handling how updates are written to the database. By default, most datasets on Windows write updates to the database server automatically when they are posted (for example, when the user moves to a new record). Client datasets, on the other hand, always cache updates in memory. For information on how to accommodate this difference, see "Updating data in dbExpress applications" on page 15-27.

Porting database applications to Linux

Porting your database application to *dbExpress* allows you to create a cross-platform application that runs on both Windows and Linux. The porting process involves making changes to your application because the technology is different. How difficult it is to port depends on the type of application it is, how complex it is, and what it needs to accomplish. An application that heavily uses Windows-specific technologies such as ADO will be more difficult to port than one that uses Kylix database technology.

Follow these general steps to port your Windows database application to Linux/CLX:

1. Make sure your data is stored in a database that is supported by *dbExpress*, such as DB2, Informix, InterBase, MySQL, Oracle, and PostgreSQL. The data needs to reside on one of these SQL servers. If your data is not already stored in one of these databases, find a utility to transfer it.

   For example, you can use the IDE's Data Pump utility (not available in all editions) to convert certain databases (such as dBase, FoxPro, and Paradox) to a dbExpress-supported database.

2. Create data modules containing the datasets and connection components so they are separate from your user interface forms and components. That way, you isolate the portions of your application that require a completely new set of components into data modules. Forms that represent the user interface can then be ported like any other application. For details, see "Porting your application" on page 15-3.

   The remaining steps assume that your datasets and connection components are isolated in their own data modules.

3. Create a new data module to hold the CLX versions of your datasets and connection components.

4 For each dataset in the original application, add a *dbExpress* dataset, *TDataSetProvider* component, and *TClientDataSet* component. Use the correspondences in Table 15.7 to decide which *dbExpress* dataset to use. Give these components meaningful names.

- Set the *ProviderName* property of the *TClientDataSet* component to the name of the *TDataSetProvider* component.
- Set the *DataSet* property of the *TDataSetProvider* component to the *dbExpress* dataset.
- Change the *DataSet* property of any data source components that referred to the original dataset so that it now refers to the client dataset.

5 Set properties on the new dataset to match the original dataset:

- If the original dataset was a *TTable*, *TADOTable*, or *TIBTable* component, set the new *TSQLTable*'s *TableName* property to the original dataset's *TableName*. Also copy any properties used to set up master/detail relationships or specify indexes. Properties specifying ranges and filters should be set on the client dataset rather than the new *TSQLTable* component.
- If the original dataset was a *TQuery*, *TADOQuery*, or *TIBQuery* component, set the new *TSQLQuery* component's *SQL* property to the original dataset's *SQL* property. Set the *Params* property of the new *TSQLQuery* to match the value of the original dataset's *Params* or *Parameters* property. If you have set the *DataSource* property to establish a master/detail relationship, copy this as well.
- If the original dataset was a *TStoredProc*, *TADOStoredProc*, or *TIBStoredProc* component, set the new *TSQLStoredProc* component's *StoredProcName* to the *StoredProcName* or *ProcedureName* property of the original dataset. Set the *Params* property of the new *TSQLStoredProc* to match the value of the original dataset's *Params* or *Parameters* property.

6 For any database connection components in the original application (*TDatabase*, *TIBDatabase*, or *TADOConnection*), add a *TSQLConnection* component to the new data module. You must also add a *TSQLConnection* component for every database server to which you connected without a connection component (for example, by using the *ConnectionString* property on an ADO dataset or by setting the *DatabaseName* property of a BDE dataset to a BDE alias).

7 For each *dbExpress* dataset placed in step 4, set its *SQLConnection* property to the *TSQLConnection* component that corresponds to the appropriate database connection.

8 On each *TSQLConnection* component, specify the information needed to establish a database connection. To do so, double-click the *TSQLConnection* component to display the Connection Editor and set parameter values to indicate the appropriate settings. If you had to transfer data to a new database server in step 1, then specify settings appropriate to the new server. If you are using the same server as before, you can look up some of this information on the original connection component:

- If the original application used *TDatabase*, you must transfer the information that appears in the *Params* and *TransIsolation* properties.

- If the original application used *TADOConnection*, you must transfer the information that appears in the *ConnectionString* and *IsolationLevel* properties.
- If the original application used *TIBDatabase*, you must transfer the information that appears in the *DatabaseName* and *Params* properties.
- If there was no original connection component, you must transfer the information associated with the BDE alias or that appeared in the dataset's *ConnectionString* property.

You may want to save this set of parameters under a new connection name. For more details on this process, see "Controlling connections" on page 21-2 and "Describing the server connection" on page 21-2.

Updating data in dbExpress applications

dbExpress applications use client datasets to support editing. When you post edits to a client dataset, the changes are written to the client dataset's in-memory snapshot of the data, but are not automatically written to the database server. If your original application used a client dataset for caching updates, then you do not need to change anything to support editing on Linux. However, if you relied on the default behavior of most datasets on Windows, which is to write edits to the database server when you post records, you must make changes to accommodate the use of a client dataset.

There are two ways to convert an application that did not previously cache updates:

- You can mimic the behavior of the dataset on Windows by writing code to apply each updated record to the database server as soon as it is posted. To do this, supply the client dataset with an *AfterPost* event handler that applies update to the database server:

```
procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
  with DataSet as TClientDataSet do
    ApplyUpdates(1);
end;
```

```
void __fastcall TForm1::ClientDataSet1AfterPost(TDataSet *DataSet)
{
  TClientDataSet *pCDS = dynamic_cast<TClientDataSet *>(DataSet);
  if (pCDS)
    pCDS->ApplyUpdates(1);
}
```

- You can adjust your user interface to deal with cached updates. This approach has certain advantages, such as reducing the amount of network traffic and minimizing transaction times. However, if you switch to using cached updates, you must decide when to apply those updates back to the database server, and probably make user interface changes to let users initiate the application of updates or inform them about whether their edits have been written to the database. Further, because update errors are not detected when the user posts a record, you will need to change the way you report such errors to the user, so that they can see which update caused a problem as well as what type of problem occurred.

If your original application used the support provided by the BDE or ADO for caching updates, you will need to make some adjustments in your code to switch to using a client dataset. The following table lists the properties, events, and methods that support cached updates on BDE and ADO datasets, and the corresponding properties, methods and events on *TClientDataSet*:

Table 15.8 Properties, methods, and events for cached updates

On BDE datasets (or TDatabase)	On ADO datasets	On TClientDataSet	Purpose
CachedUpdates	*LockType*	Not needed, client datasets always cache updates.	Determines whether cached updates are in effect.
Not supported	*CursorType*	Not supported.	Specifies how isolated the dataset is from changes on the server.
UpdatesPending	Not supported	*ChangeCount*	Indicates whether the local cache contains updated records that need to be applied to the database.
UpdateRecordTypes	*FilterGroup*	*StatusFilter*	Indicates the kind of updated records to make visible when applying cached updates.
UpdateStatus	*RecordStatus*	*UpdateStatus*	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError	Not supported	*OnReconcileError*	An event for handling update errors on a record-by-record basis.
ApplyUpdates (on dataset or database)	*UpdateBatch*	*ApplyUpdates*	Applies records in the local cache to the database.
CancelUpdates	*CancelUpdates* or *CancelBatch*	*CancelUpdates*	Removes pending updates from the local cache without applying them.
CommitUpdates	Handled automatically	*Reconcile*	Clears the update cache following successful application of updates.
FetchAll	Not supported	*GetNextPacket* (and *PacketRecords*)	Copies database records to the local cache for editing and updating.
RevertRecord	*CancelBatch*	*RevertRecord*	Undoes updates to the current record if updates are not yet applied.

Cross-platform Internet applications

An Internet application is a client/server application that uses standard Internet protocols for connecting the client to the server. Because your applications use standard Internet protocols for client/server communications, you can make your applications cross-platform. For example, a server-side program for an Internet application communicates with the client through the Web server software for the machine. The server application is typically written for Linux or Windows, but can also be cross-platform. The clients can be on either platform.

You can use Kylix to create Web server applications as CGI or Apache applications to deploy on Linux. On Windows, you can create other types of Web servers such as Microsoft Server DLLs (ISAPI), Netscape Server DLLs (NSAPI), and Windows CGI applications. Only straight CGI applications and some applications that use Web Broker will run on both Windows and Linux.

Porting Internet applications to Linux

If you have an existing Windows Internet applications that you want to make cross-platform, you can either port your Web server application to Kylix or create a new application on Kylix. See Chapter 29, "Creating Internet server applications" for information on writing Web servers. If your application uses Web Broker, writes to the Web Broker interface, and does not use native API calls, it is not as difficult to port it to Linux.

If your application writes to ISAPI, NSAPI, Windows CGI, or other Web APIs, it is more difficult to port. You need to search through your source files and translate these API calls into Apache (see ..\source\internet\HTTPD.pas (Delphi) or ..\include\vcl\HTTPD.hpp (C++) for function prototypes for Apache APIs) or CGI calls. You also need to make all other suggested changes described in "Porting Windows applications to Linux" on page 15-2.

Chapter

16

Working with packages and components

A *package* is a special shared object file used by CLX applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages.

Packages are stored in the bin directory in shared object files (typically prefixed with bpl) such as bpl<*packagename*>.so.*version#* where *version#* is the main version number of the package. (The actual package name may also include a revision number as well and a symbolic link points to the name with only the version number. So, for example bplindy.so.6 might point to bplindy.so.6.2.) Package documentation refers only to the package name without the bpl prefix, version number and revision number.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used CLX components reside in a package called visualclx. Each time you create a new default application, it automatically uses visualclx. When you build an application created this way, the application's executable image contains only the code and data unique to it; the common code is in visualclx. A computer with several package-enabled applications installed on it needs only a single copy of visualclx, which is shared by all the applications and the IDE itself.

Kylix ships with several runtime packages that encapsulate CLX components. Kylix also uses design-time packages to manipulate components in the IDE.

You can build applications with or without packages. However, if you want to add custom components to the IDE, you must install them as design-time packages.

You can create your own runtime packages to share among applications. If you write CLX components, you can build your components into design-time packages before installing them.

Why use packages?

Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By building reused code into a runtime library, you can share it among applications. For example, all of your applications—including Kylix itself—can access standard components through packages. Since the applications don’t have separate copies of the component library bound into their executables, the executables are much smaller, saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

Packages and standard shared object files

Create a package when you want to make a custom component that’s available through the IDE. Create a standard shared object file when you want to build a library that can be called from any application, regardless of the development tool used to build the application.

D The following table lists the file types associated with Delphi packages:

Table 16.1 Delphi package files

File extension	Contents
bpl<*packagename*>.so	The runtime package. This file is a shared object file with special Kylix-specific features. The base name for the package is the base name of the of the .dpk or .dpkl source file.
dcp	A binary image containing a package header and the concatenation of all dcu and dpu files in the package, including all symbol information required by the compiler. A single dcp file is created for each package. The base name for the dcp is the base name of the .dpk source file. You must have a .dcp file to build an application with packages.
dcu, dpu, and pas	The binary images for a unit file contained in a package. One .dcu and .dpu (for pic) is created, when necessary, for each unit file.
dpk and dpkl	The source files listing the units contained in the package.

The following table lists the file types associated with C++ packages:

Table 16.2 C++ package files

File extension	Contents
.bpf	A source file required for a package.
bpi	A Borland package import library. A .bpi is created for each package. This file is passed to the linker by applications using the package to resolve references to functions in the package. The base name for the bpi is the base name for the package source file.
bpk and bpkl	The project options source file. This file is the XML portion of the package project. The *ProjectName*.bpk is used to manage settings, options, and files used by the package project.

Table 16.2 C++ package files (continued)

File extension	Contents
bpl<*packagename*>.so	The runtime package. This file is a shared object file with special Kylix-specific features. The base name for the package is the base name of the .bpk or .bpkl source file.
h	The header file or interface for the component. *componentname*.h is the companion to *componentname*.cpp.
a	A static library, or collection of .o files, used in place of a .bpi when the application does not use runtime packages.
o	A binary image for a unit file contained in a package. One ..o is created, when necessary, for each .cpp file.

Note Packages share their global data with other modules in an application.

For more information about Delphi shared object files and packages, see the *Delphi Pascal Language Guide*.

Runtime packages

Runtime packages are deployed with your applications. They provide functionality when a user runs the application.

To run an application that uses packages, a computer must have both the application's executable file and all the packages that the application uses. The package files must be on the LD_LIBRARY_PATH for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required packages.

Loading packages in an application

You can dynamically load packages by either:

- Choosing Project Options dialog box in the IDE; or
- Using the *LoadPackage* function.

To load packages using the Project | Options dialog box:

1. Load or create a project in the IDE.
2. Choose Project | Options.
3. Choose the Packages tab.
4. Select the Build with runtime packages check box, and enter one or more package names in the edit box underneath. Each package is loaded implicitly only when it is needed (that is, when you refer to an object defined in one of the units in that package). (Runtime packages associated with installed design-time packages are already listed in the edit box.)

5 To add a package to an existing list, click the Add button and enter the name of the new package in the Add Runtime Package dialog. To browse from a list of available packages, click the Add button, then click the Browse button next to the Package Name edit box in the Add Runtime Package dialog.

If you edit the Search Path edit box in the Add Runtime Package dialog, you can change the global Library Path.

You do not need to include file extensions with package names. If you type directly into the Runtime Packages edit box, be sure to separate multiple names with colons.

Packages listed in the Runtime Packages edit box are automatically linked to your application when you build the project. Duplicate package names are ignored, and if the Build with runtime packages check box is unchecked, the application is built as a single executable that does not use packages.

Runtime packages are selected for the current project only. To make the current choices into automatic defaults for new projects, select the Defaults check box at the bottom of the dialog.

Delphi example

In Delphi, when you create an application with packages, you must include the names of the original Delphi units in the uses clause of your source files. For example, the source file for your main form might begin like this:

```
unit MainForm;

interface

uses
SysUtils, Types, Classes, QGraphics, QControls, QForms,
QDialogs;
```

In Delphi, the units referenced in this example are contained in the visualclx and rtl packages. Nonetheless, you must keep these references in the uses clause or you will get compiler errors. In generated source files, the Form Designer adds these units to the uses clause automatically.

C++ example

In C++, when you create an application with packages, you must include header files for the packaged units that it uses. For example, the header file for your main form contains the following:

```
#include <QControls.hpp>
#include <QForms.hpp>
```

In generated source files, the Code editor creates these #include statements automatically.

Loading packages with the *LoadPackage* function

You can also load a package at runtime by calling the *LoadPackage* function. *LoadPackage* loads the package specified by its name parameter, checks for duplicate units, and calls the initialization blocks of all units contained in the package. For example, the following code could be executed when a file is chosen in a file-selection dialog.

Delphi example

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

C++ example

```
if (OpenDialog1->Execute())
  PackageList->Items->AddObject(OpenDialog1->FileName, (TObject *)LoadPackage(OpenDialog1-
>FileName));
```

To unload a package dynamically, call *UnloadPackage*. Be careful to destroy any instances of classes defined in the package and to unregister classes that were registered by it.

Deciding which runtime packages to use

Several runtime packages, including rtl and visualclx, supply basic language and component support. This is how the components are distributed with the product and the packages include units with somewhat related functionality.

For example, the visualclx package contains the most commonly used components and user interface elements such as buttons and list boxes; the rtl package includes all the non-component system functions. It does not include database or other special components, which are available in separate packages.

To create a client/server database application that uses packages, you need several runtime packages, including rtl and dataclx. If you want to use visual components in your application, you also need visualclx and possibly visualdbclx if using data-aware controls. To use these packages, choose Project|Options, select the Packages tab, and enter the packages you want to use in the Runtime Packages edit box.You need netclx for Web server applications, as well as baseclx and probably visualclx.

The following table lists the runtime packages. Which packages you have depends on which edition of the product you have and which ones you need for your application depends on the type of application it is.

Table 16.3 Runtime packages

Package	Contents
rtl	Runtime library and system support units. Required for all applications.
dataclx	Database components. Required for all database applications.
indy	Cross-platform Internet components for the server and client. Open source components required for Internet applications that use the Indy components.
netclx	Internet components. Required for all Internet applications.

Table 16.3 Runtime packages (continued)

Package	Contents
netdataclx	Content producers for Web Broker applications. Required for Web Broker applications that use database information.
soaprtl	Web Services objects for building modular applications that can be published and invoked over a network (such as the World Wide Web). Required for all Web Services applications.
visualclx	Visual components to support cross-platform applications. Required for all GUI applications.
visualdbclx	Data-aware controls that appear on the Data Controls tab of the Component palette (such as DBImage, DBGrid, and DBEdit). Required for applications that display live data from a database.
webdsnapclx	Components for building DataSnap applications. Required for multi-tiered database applications that include client applications and application servers that fetch data from remote database servers or XML documents.
websnapclx	Dispatcher, adapter, page producer, session, and user list components for producing scriptable HTML page templates. Required for building Web server applications that use WebSnap components.
xmlrtl	Components that support XML documents, XML schemas, and Document Object Model (DOM) standard interfaces. Required for applications that work with XML data.

This table should give you a general idea of which packages your application may need. Another way you can determine which packages are called by an application is to run it then review the event log (choose View | Debug Windows | Event Log). The event log displays every module that is loaded including all packages. The full package names are listed. So, for example, for visualclx, you would see a line similar to the following:

```
Module Load: bplvisualclx.so.6. Has Debug Info. Base Address $40017000. Process Proj1 (2906)
```

Note You don't have to include rtl, because they are referenced in the Requires list. Your application can be built just the same whether or not rtl is included in the Runtime Packages edit box.

Custom packages

A custom package is either a package you code and build yourself or an existing package from a third-party vendor. To use a custom runtime package with an application, choose Project | Options and add the name of the package to the Runtime Packages edit box on the Packages page. If you create your own packages, you can add them to the list as needed.

For example, suppose you have a statistical package called bplstats.so. To use it in an application, include it in the Runtime Packages edit box:

```
rtl:visualclx:visualdbclx:stats
```

You must also include stats.dcp (Delphi) or stats.bpi and stats.a (C++).

Design-time packages

Design-time packages are used to install components on the IDE's Component palette. Which ones are installed depends on which edition you are using and whether you have customized it. You can view a list of what packages are installed on your system by choosing Component | Install Packages.

The design-time packages work by calling runtime packages, which they reference in their Requires lists, as described on page 16-10. For example, dclstd references visualclx. The dclstd package contains additional functionality that makes many of the standard components available on the Component palette.

By convention, IDE design packages start with a dcl and reside in the bin directory.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The dcluser design-time package is provided as a default container for new components.

Installing component packages

All components are installed in the IDE as packages. If you've written your own components, a package that contains them. (See "Creating and editing packages" on page 16-8.) Your component source code must follow the model described in Part IV, "Creating custom components."

To install or uninstall your own components, or components from a third-party vendor, follow these steps:

1 If you are installing a new package, copy or move the package files to a local directory. If the package is shipped with the following files, be sure to copy all of them:

- In Delphi: .so, .dcp, .dcu, and .dpu files.
- In C++: .bpi, .so, .a, .o, and header files.

For information about these files, see , "Packages and standard shared object files," on page 16-2.

The directory where you store the .dcp file (Delphi) or .bpi, .a, and header files (C++) must be in the ibrary path. If you are including .dcu and .dpu files (Delphi) or .a and .o files (C++) with the distribution, also add the directory for these files to the library path.

2 Choose Component | Install Packages from the IDE menu, or choose Project | Options and click the Packages tab.

3 A list of available packages appears in the Design packages list box.

- To install a package in the IDE, select the check box next to it.
- To uninstall a package, uncheck its check box.
- To see a list of components included in an installed package, select the package and click Components.

- To add a package to the list, click Add and browse in the Add Design Package dialog box for the directory where the bpl* or dcl* file resides (see step 1). Select the bpl*.so or dcl*.so file and click Open.
- To remove a package from the list, select the package and click Remove.

4 Click OK.

The components in the package are installed on the Component palette pages specified in the components' *RegisterComponents* procedure, with the names they were assigned in the same procedure.

New projects are created with all available packages installed, unless you change the default settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the Packages tab of the Project Options dialog box.

To remove components from the Component palette without uninstalling a package, select Component | Configure Palette, or select Tools | Environment Options and click the Palette tab. The Palette options tab lists each installed component along with the name of the Component palette page where it appears. Selecting any component and clicking Hide removes the component from the palette.

Creating and editing packages

Creating a package involves specifying:

- A *name* for the package.
- A list of other packages to be *required* by, or linked to, the new package.
- A list of unit files to be *contained* by, or bound into, the package when it is built. The package is essentially a wrapper for these source-code units. The Contains list is where you put the source-code units for custom components that you want to build into a package.

The Package editor generates:

- In Delphi, a package source file (.dpk).
- In C++, a project options file (.bpk).

Creating a package

To create a package, follow the procedure below. Refer to "Understanding the structure of a package" on page 16-10 for more information about the steps outlined here.

In Delphi, do not use conditional compiler directives in a package file (.dpk) such as when doing cross-platform development. You can use them in the source code, however.

1 Choose File | New | Other, select the Package icon, and click OK.

2 The generated package is displayed in the Package editor.

3 The Package editor shows a *Requires* node and a *Contains* node for the new package.

4 To add a unit to the Contains list, click the Add button. In the Add Unit page, type a .pas (Delphi) or .cpp (C++) file name in the Unit file name edit box, or click Browse to browse for the file, and then click OK. The unit you've selected appears under the Contains node in the Package editor. You can add additional units by repeating this step.

5 To add a package to the Requires list, click the Add button. In the Requires page, type a .dcp (Delphi) or .bpi (C++) file name in the Package name edit box, or click Browse to browse for the file, and then click OK. The package you've selected appears under the Requires node in the Package editor. You can add additional packages by repeating this step.

6 Click the Options button, and decide what kind of package you want to build.

   - To create a design-time only package (a package that cannot be used at runtime), check the Designtime only option. Or, in Delphi, add the {$DESIGNONLY} compiler directive to the dpk file. In C++, add the **-Gpd** linker switch to your bpk file: `LFLAGS = ... -Gpd ....`
   - To create a runtime-only package (a package that cannot be installed), check the Runtime only option. Or, in Delphi, add the {$RUNONLY} compiler directive to the dpk file. In C++, add the **-Gpr** linker switch to your .bpk file: `LFLAGS = ... -Gpr ....`
   - To create a package that is available at both design time and runtime, select the Designtime and runtime radio button.

7 In the Package editor, click the Compile package button to compile your package.

   You can also click the Install button to force a make.

Editing an existing package

You can open an existing package for editing in several ways:

- Choose File | Open (or File | Reopen) and select a .dpk (Delphi) or .cpp and .bpk (C++) file.
- Choose Component | Install Packages, select a package from the Design packages list, and click the Edit button.
- When the Package editor is open, select one of the packages in the Requires node, right-click, and choose Open.

To edit a package's description or set usage options, click the Options button in the Package editor and select the Description tab.

The Project Options dialog has a Default check box in the lower left corner. If you click OK when this box is checked, the options you've chosen are saved as default settings for new projects. To restore the original defaults, delete or rename the defproj.dof (Delphi) or default.bpk (C++) file.

Understanding the structure of a package

Packages include the following parts:

- Package name
- Requires list
- Contains list

Naming packages

Package names must be unique within a project. If you name a package stats:

- In Delphi, the Package editor generates a source file called stats.dpk. The compiler generates an executable and a binary image called bplstats.so and stats.dcp, respectively.
- In C++, the Package editor generates a source file and project options file called stats.bpf and stats.bpk, respectively. The compiler and linker generates an executable, a binary image (bplstats.so), and (optionally) a static library called stats.a and a package import file called stats.bpi.

Use stats to refer to the package in the Requires list of another package, or when using the package in an application.
In C++, to use the package in an application, add Stats to the Runtime Packages edit box (after choosing Project | Options and clicking the Packages tab).

You can also add a prefix, suffix, and version number to your package name. While the Package editor is open, click the Options button. On the Description page of the Project Options dialog box, enter text or a value for SO Suffix, SO Prefix, or SO Version. For example, to add a version number to your package project, enter 3 after SO Version so that *Package1* generates bpl*Package1*.so.3.

Requires list

The Requires list specifies other, external packages that are used by the current package. An external package included in the Requires list is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in your package make references to other packaged units, the other packages should appear in your package's Requires list or you should add them. If the other packages are omitted from the Requires list, the compiler will import them into your package 'implicitly contained units.' If a package does not contain any units that use units in another package, then it doesn't need a Requires list.

Note Most packages that you create require rtl and visualclx. .

Avoiding circular package references

Packages cannot contain circular references in their Requires list. This means that

- A package cannot reference itself in its own Requires list.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Handling duplicate package references

Duplicate references in a package's Requires list—or in the Runtime Packages edit box—are ignored when the package is built. For programming clarity and readability, however, you should catch and remove duplicate package references.

Contains list

The Contains list identifies the unit files to be compiled and bound into the package. If you are writing your own package, put your source code in pas (Delphi) or .cpp (C++) files and include them in the Contains list. All units used by contained units which do not exist in required packages will also be bound into the package, although they won't be listed in the contains list (the compiler will give a warning).

Avoiding redundant source code uses

A package cannot appear in the Contains list of another package.

All units included directly in a package's Contains list, or included indirectly in any of those units, are bound into the package when it is built.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application, including the IDE. This means that if you create a package that contains one of the units in visualclx, you won't be able to install your package in the IDE. To use an already-packaged unit file in another package, put the first package in the second package's Requires list.

D Editing package source files manually in Delphi

Package source files, like project files, are generated by the IDE from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Delphi package) extension to avoid confusion with other files containing Delphi source code.

To open a package source file in the Code editor,

1 Open the package in the Package editor.

2 Right-click in the Package editor and select View Source.

To understand the elements of the source file, see "Understanding the structure of a package" on page 16-10.

For example, the following code declares the MyPack package (in the source file visualclx):

```
package MyPack;
{$R *.res}
    ...{compiler directives omitted}
   requires
    rtl,
    visualclx;
   contains
    Db,
    mycomponent in 'usr/components/mycomponent.pas';
end.
```

Package source files and project options files in C++

In C++, package source files have the .cpp extension. Package project options files are created using the XML format and have the .bpk extension. Display the package project options file from the Package editor by right-clicking on the Contains or Requires list and choosing Edit Option Source.

Note Kylix for C++ maintains the .bpk file. You do not normally need to edit it manually. You should make changes using the Packages tab of the Project Options dialog box.

The project options file (bpk.xml) for a package called MyPack might look, in part, like this:

```
<MACROS>
    <VERSION value="BCB.06.00"/>
    <PROJECT value="MyPack.so"/>
    <OBJFILES value="MyPack.o Unit1.o"/>
    <RESFILES value=""/>
    <IDLFILES value=""/>
    <IDLGENFILES value=""/>
    <DEFFILE value=""/>
    <RESDEPEN value="$(RESFILES) Unit1.xfm"/>
    <LIBFILES value=""/>
    <LIBRARIES value=""/>
    <SPARELIBS value="rtl.a clx.a"/>
    <PACKAGES value="rtl.bpi clx.bpi"/>
     ⋮
```

File information is stored in <Filelist> section:

```
<FILELIST>
    <FILE FILENAME="MyPack.cpp" FORMNAME="" UNITNAME="MyPack" ... LOCALCOMMAND=""/>
    <FILE FILENAME="rtl.bpi" FORMNAME="" UNITNAME="rtl" ... LOCALCOMMAND=""/>
    ⋮
```

MyPack's Contains list includes two units: MyPack itself and Unit1. MyPack's Requires list includes rtl.

Packaging components in C++

In C++, if you use the New Component wizard (by choosing Component|New Component) to create components in C++, the wizard inserts the PACKAGE macro where it is needed. But if you have custom components from an older version of the C++ IDE, you'll have to add PACKAGE manually in two places.

The header file declaration for a C++ component must include the predefined PACKAGE macro after the word **class**:

```
class PACKAGE mycomponent : ...
```

And in the cpp file where the component is defined, include the PACKAGE macro in the declaration of the *Register* function:

```
void __fastcall PACKAGE Register()
```

The PACKAGE macro expands to a statement that allows classes to be imported and exported from the resulting package file.

Building packages

You can build a package from the IDE or from the command line. To rebuild a package by itself from the IDE:

1. Choose File|Open and select either a Delphi package file or a C++ source file or project options file.
   - In the Delphi IDE, choose a package (.dpk or .dpkl).
   - In the C++ IDE, choose a project options file (.bpk or .bpkl).
2. Click Open.
3. When the Package editor opens:
   - Click the Package editor's Compile button.
   - In the IDE, choose Project|Build (Delphi), Project|Make (C++) or Project|Build.

Note You can also choose File|New|Other and double-click the Package icon. Click the Install button to make the package project. Right-click the package project nodes for options to install, compile (Delphi), make (C++) , or build.

If you are adding a unit or .cpp file that is not automatically generated, add one of the compiler directives into your package source code. For more information, see "Package-specific compiler directives" below.

If compiling or linking using the command line, you can use several package-specific switches. For more information, see "Compiling and linking from the command line" on page 16-15.

Package-specific compiler directives

You can insert package-specific compiler directives into your source code.

D For Delphi package-specific compiler directives, see Table 16.4:

Table 16.4 Delphi package-specific compiler directives

Directive	Purpose
{$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{$G–} or {$IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{$WEAKPACKAGEUNIT ON}	Packages unit "weakly," as explained in the online Help.
{$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

For more infomation, see Chapter 9, "Libraries and packages," in the *Delphi Language Guide* for package-specific compiler directives.

C++ For C++ package-specific compiler directives, see Table 16.5:

Table 16.5 C++ package-specific compiler directives

Directive	Purpose
#pragma package(smart_init)	Assures that packaged units are initialized in the order determined by their dependencies. (Included by default in package source file.)
#pragma package(smart_init, weak)	Packages unit "weakly." See "Weak packaging" later. (Put directive in unit source file.)

See "Working with shared object libraries" on page 8-5 for additional directives that can be used in all libraries.

Weak packaging

The $WEAKPACKAGEUNIT (Delphi) or #pragma package(smart_init, weak) (C++) compiler directive affects the way a .dcu and .dpu (Delphi) or .o (C++) file is stored in a package's .dcp (Delphi) or .bpi and bpl<*packagename*>.so (C++) files. (For information about files generated by the compiler, see "Package files created when building" on page 16-16.) If the weak packaging compiler directive appears in a unit file, the unit won't be built into the packages, and a non-packaged local copy of the unit is created when it is required by another application or package. A unit compiled with this directive is said to be *weakly packaged*.

For example, suppose you've created a package called pack that contains only one unit, unit1. Suppose unit1 does not use any additional units, but it makes calls to rare.so. If you put the {$WEAKPACKAGEUNIT ON}/#pragma package(smart_init, weak) directive in unit1.pas (Delphi) or unit1.cpp (C++) when you build your package, unit1 will not be included in bplpack1.so; you will not have to distribute copies of rare.so with pack1. However, unit1 will still be included in bplpack1.so (Delphi) or pack1.bpi (C++). If unit1 is referenced by another package or application that uses pack, it will be copied from bplpack1.so (Delphi) or pack1.bpi (C++) and linked directly into the project.

Now suppose you add a second unit, unit2, to pack. Suppose that unit2 uses unit1. This time, even if you compile pack1 with the weak packaging directive in unit1.pas (Delphi) or unit1.cpp (C++), unit1 is still included in bplpack1.so. However, other packages or applications that reference unit1 will use the (non-packaged) copy taken from pack1.dcp (Delphi) or pack1.bpi (C++).

Note Unit files containing the {$WEAKPACKAGEUNIT ON}/#pragma package(smart_init, weak) directive must not have global variables, initialization sections (Delphi), or finalization sections (Delphi).

The weak packaging directive is an advanced feature intended for developers who distribute their packages to other programmers. It can help you to avoid distribution of infrequently used shared object files, and to eliminate conflicts among packages that may depend on the same external library.

Compiling and linking from the command line

When you compile and link from the command line, you can use the package-specific switches listed in the following tables.

D In Delphi, the command-line compiler switches include:

Table 16.6 Delphi package-specific command-line compiler switches

Switch	Purpose
-$G-	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
-LE*path*	Specifies the directory where the bpl*package*.so file will be placed.
-LN*path*	Specifies the directory where the *package*.dcp file will be placed.
-LU*package*	Use packages.
-Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

D In Delphi, using the **-$G-** switch may prevent a package from being used in the same application with other packages. Other command-line options may be used, if appropriate, when compiling packages. See "The Command-line compiler" in the online Help for information on command-line options not discussed here.

In C++, the command-line compiler switches include:

Table 16.7 C++ package-specific command-line compiler and linker switches

Switch	Purpose
tP	Generates a project as a package.
-t	Generates a GUI executable.
-tC	Generates a console executable.
-tD	Generates a shared object.
-D "*description*"	Saves the specified description with the package.
-Gb	Generates the .bpi filename.
-Gi	Saves the generated .bpi file. Included by default in package project files.
-Gpd	Generates a design-time-only package.
-Gpr	Generates a runtime-only package.
-Gl	Generates a library file.
-Tpp	Builds the project as a package. Included by default in package project files.
-Tpc -aa	Builds
-Tpe -aa	Builds
-ap	Builds
-Tpd -aa	Builds

In C++, when you link from the command line, use the **-Tpp** linker switch to ensure that the project is built as a package. The **-Gpr** and **-Gpd** switches correspond to the Runtime only and Designtime only check boxes on the Description page of the Project Options dialog (available for package projects only); if neither **-Gpr** nor **-Gpd** is used, the resulting package works at both design time and runtime. The **-D** switch corresponds to the Description edit box on the same page.

In C+, you can generate a makefile to use on the command line by choosing Project | Export Makefile.

Package files created when building

In Delphi, to create a package, you compile a source file that has a .dpk extension. The base name of the .dpk file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called traypak.dpk, the compiler creates a package called bpltraypak.so.

In C++, to create a package, you compile a source file (.cpp) using a project options file with the .bpk extension. The base name of the source file should match the base name of the files generated by the compiler; that is, if the source file is called traypak.cpp, the project options file—traypak.bpk—should include `<PROJECT value="bpltraypak.so"/>`. In this case, compiling and linking the C++ project creates a package called bpltraypak.so.

A successfully compiled package includes .dcp, .dcu and .dpu, and bpl<*packagename*>.so files (Delphi) or .bpi, bpl<*packagename*>.so, .o, and .a files (C++). For a detailed description of these files, see "Packages and standard shared object files" on page 16-2.

These files are generated by default in the directories specified in Library page of the Tools | Environment Options dialog. You can override the default settings by clicking the Options button in the Package editor to display the Project Options dialog; make any changes on the Directories/Conditionals page.

Deploying packages

You deploy packages much like you deploy other applications. The files you distribute with a deployed package may vary. The bpl<*packagename*>.so and any packages or shared objects required by the bpl<*packagename*>.so must be distributed.

In C++, the following table lists the files that may be necessary, depending on the intended use of the package.

Table 16.8 C++ files deployed with a package

File	Description
componentname.h	Allows end users access to the class interfaces.
componentname.cpp	Allows end users access to the component source.
bpi, .o, and .a	Allows end users to link applications.

For general deployment information, refer to Chapter 18, “Deploying applications.”

Deploying applications that use packages

When distributing an application that uses runtime packages, make sure that your users have the application’s executable file as well as all the library files that the application calls. If the library files are in a different directory from the executable file, they must be accessible through the user’s path.

Distributing packages to other developers

If you distribute runtime or design-time packages to other developers, be sure to supply both .dcp (Delphi) or .bpi (C++) and package files as well as any required .dcu and .dpu (Delphi) or header files (C++).

Note If the administrator (root) adds a new package to Kylix, users won't see the package until they remove the delphi69rc (Delphi) or bcb69rc (C++) file in their .borland directory.

In C++, to link components statically into their applications—that is, to build applications that don’t use runtime packages—developers will also need .a (or .o) files for any packages you supply.

Chapter 17

Creating international applications

This chapter discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market.

Internationalization and localization

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

- Internationalization
- Localization

Internationalization

Internationalization is the process of enabling your program to work in multiple locales. A locale is the user's environment, which includes the cultural conventions of the target country as well as the language. Linux supports many locales, each of which is described by a language and country pair (for example, en_US for English/United States). The user locale identifier can be obtained from the operating system by looking for the LANG environment variable.

Most Kylix applications do not need to determine the exact locale in which the application is running but do need to handle strings in a locale-independent manner.

Localization

Localization is the process of translating an application so that it functions in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified for the tax laws in different countries.

Internationalizing applications

You need to complete the following steps to create internationalized applications:

- Enable your code to handle strings from international character sets.
- Design your user interface to accommodate the changes that result from localization.
- Isolate all resources that need to be localized.

Enabling application code

You must make sure that the code in your application can handle the strings it will encounter in the various target locales.

Character sets

The Linux operating system uses UTF-8 to encode file names and paths, and other strings passed to the operating system kernel. The Linux operating system generally has no knowledge of locale specifics.

End user applications generally require more specific locale support than simply which character set to display on the screen. Locales and languages have different rules on how string data should be sorted, what characters are considered equivalent in comparisons, and how numbers should be formatted for display. UTF-8 provides no such support; you have to use locales and local character sets.

Multibyte character sets

The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) *char* type. These languages have too many characters to be represented using the single-byte *char*. Instead, a multibyte string can contain one or more bytes per character. AnsiStrings can contain a mix of single-byte and multibyte characters.

The lead byte of every multibyte character code is taken from a reserved range that depends on the specific character set. The second and subsequent bytes can sometimes be the same as the character code for a separate one-byte character, or it can fall in the range reserved for the first byte of multibyte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or is part of a multibyte character is to read the string, starting at the beginning, parsing it into two or more byte characters when a lead byte from the reserved range is encountered.

When writing code for Asian locales, you must be sure to handle all string manipulation using functions that are enabled to parse strings into multibyte characters.

Delphi includes many of these runtime library functions, many that are listed here:

AdjustLineBreaks	AnsiStrLower	ExtractFileDir
AnsiCompareFileName	AnsiStrPos	ExtractFileExt
AnsiExtractQuotedStr	AnsiStrRScan	ExtractFileName
AnsiLastChar	AnsiStrScan	ExtractFilePath
AnsiLowerCase	AnsiStrUpper	ExtractRelativePath
AnsiLowerCaseFileName	AnsiUpperCase	FileSearch
AnsiPos	AnsiUpperCaseFileName	IsDelimiter
AnsiQuotedStr	ByteToCharIndex	IsPathDelimiter
AnsiStrComp	ByteToCharLen	LastDelimiter
AnsiStrIComp	ByteType	StrByteType
AnsiStrLastChar	ChangeFileExt	StringReplace
AnsiStrLComp	CharToByteIndex	WrapText
AnsiStrLIComp	CharToByteLen	

For C++, see "International API" and "MBCS utilities" in the online Help for a list of the RTL functions that are enabled to work with multibyte characters.

Note If your application will be run on Linux systems using pre-2.2 versions of glibc, avoid passing large strings (greater than 50K) to multibyte-enabled routines. (There are no string limitations if using glibc version 2.2 or greater.)

Remember that the length of the strings in bytes does not necessarily correspond to the length of the string in characters. Be careful not to truncate strings by cutting a multibyte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character can't be known up front. Instead, always pass a pointer to a character or a string.

Wide characters

Display string properties such as captions, descriptions, text properties, combo boxes, and so on are all wide strings.

Note The Linux wchar_t WideChar is 32 bits per character. The 16-bit Unicode standard that Object Pascal WideChars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal WideChar data must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

The Linux kernel uses 4-byte widechars. The kernel expects strings (file names and so forth) to be encoded in UTF-8. Kylix WideChar and WideString are 2 bytes per character Unicode, which is a subset of the UCS-4 specification. To translate Unicode 2-byte characters to UCS 4-byte characters, you must add two bytes of zeros in front.

Designing the user interface

When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation.

Text

All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters.

Short strings tend to grow in translation more than long phrases. Table 17.1 provides a rough estimate of how much expansion you should plan for given the length of your English strings:

Table 17.1 Estimating string lengths

Length of English string (in characters)	Expected increase
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
over 50	10%

Graphic images

Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image.

There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not appropriate if your application is intended for countries that have different dominant religions. Even color can have different symbolic connotations in different cultures.

Formats and sort order

The date, time, number, and currency formats used in your application should be localized for the target locale. If you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized.

The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, two-character combinations are treated as a single character in the sort order. For example, in Spanish, the combination *ch* is sorted like a single unique letter between *c* and *d*. Sometimes a single character is sorted as if it were two separate characters, such as the German *eszett*.

Keyboard mappings

Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.

Isolating resources

The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. The IDE automatically creates a.xfm file that contains the resources for your menus, dialogs, and bitmaps.

In addition to these obvious user interface elements, you need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the form file. You can isolate them by declaring constants for them using the resourcestring keyword. For more information about resource string constants, see the *Delphi Language Guide*. It is best to include all resource strings in a single, separate unit. In C++, you can isolate string resources into an .rc file.

Kylix resource strings are encoded in UTF-8 (1 byte per character, usually) in the executable file.

Creating resource modules

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource module. A resource module contains all the resources and only the resource strings for a program. Resource modules allow you to create a program (no code) that supports many translations simply by swapping the resource module.

To create a resource module for your program, create a file that contains the resourcestring strings for the project and generate a project for a resource only shared object file that contains the relevant forms. The resources are compiled into a separate section of the executable file.

You should create a resource module for each translation you want to support. Each resource module should have a file name specific to the target locale, for example en_US for US English.

Using resource modules

The executable, shared object files, and packages that make up your application contain all the necessary resources. However, to replace those resources with localized versions, you need only ship your application with localized resource modules that have the same name as your executable, shared object file, or package files.

When your application starts up, it checks the locale of the local system by looking at the LANG environment variable. If it finds any resource modules with the same name as the executable file, shared object file, or package files it is using, it checks the extension of those shared object files. If the extension of the resource module matches the language and country of the system locale, your application will use the resources in that resource module instead of the resources in the executable, shared object file, or package. If no resource module matches both the language and the country, your application will try to locate a resource module that matches the language only. If no resource module file name extension matches the language, your application uses the resources compiled with the executable, shared object file, or package.

You can ship a single application that adapts itself automatically to the locale of the system it is running on, simply by providing the appropriate resource DLLsmodules.

Localizing applications

Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

Ideally, your resources have been isolated into a resource module that contains form files. You can open your forms in the IDE, translate the relevant properties, then compile the application. You can extract any resources needed using the *resbind* command line utility located in ~/kylix/bin.

Resbind extracts the Borland resources from your application and creates a shared object file that contains the resources. You can then dynamically link the resources at runtime or let the application check the environment variable on the local system on which it is running.

Resbind reads resources from an ELF file and optionally one or more resource files. The ELF file may be a Linux i386 executable or shared object file. *Resbind* accepts resource files in Windows 32-bit and 16-bit formats, and in Borland's resource file format for Linux.

Resbind can print the resources, copy them to a new resource file, or copy them to a copy of the ELF file with the resources replaced by resources read from the resource files using the following syntax:

```
resbind [option]... ELF-FILE RESOURCE-FILE...
```

The following table lists the *resbind* options.

Table 17.2 Resbind options

Option	Description
-h	Display help and exit
-V	Output version information and exit

Table 17.2 Resbind options (continued)

Option	Description
-f FMT	Control the resource file output format. FMT may be ‘w32’ (default) or ‘borland’
-l FILE	Dynamic linker to use for -s, default /lib/ld-linux.so.2
-o FILE	Write the updated ELF-FILE to FILE
-p	Print combined resources
-r FILE	Write combined resources to resource file FILE
-s FILE	Write combined resources to resource shared object FILE
-S SONAME	Define soname for -s

Resbind can be used to copy resources from a application into a resource file for editing. Later, you can copy the edited resources back into the application.

For example, the following command copies resources from an ELF file called “program” to a Windows 32-bit resource file called “program.res:”

```
$ resbind -r program.res program
```

You can then combine the edited resources located in “program.res” with the ELF file called “program” and write the result to “program.new.”

```
$ resbind -o program.new program program.res
```

Resources not included in “program.res” remain unchanged.

You can also write combined resources to a resource shared object:

```
$ resbind -s program-res.so program program.res
```

Chapter

18

Deploying applications

Once your application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. The steps required by a given application vary, depending on the type of application. The following sections describe these steps when deploying the following applications:

- Deploying general applications
- Deploying cross-platform applications
- Deploying database applications
- Deploying Web applications
- Programming for varying host environments
- Software license requirements

Deploying general applications

On Linux, applications are commonly deployed using one of the following methods:

- Using a tool such as RPM Package Manager (RPM)
- By creating tar files (tar.gz) or gnu zip files (gzip)
- Using a setup program developed for Linux such as the Setup Graphic Installer open sourced by Loki Entertainment Software

Each of these methods has its advantages and disadvantages and which one to use depends on the type of application and files required to run the application. Simple programs can be deployed using tar files. More complex programs should be deployed using either RPM or a setup program.

RPM is used to install, uninstall, upgrade, verify, and build software packages. RPM creates an archive of files and application information (including a name, version, and brief description). It is available on many versions of Linux and UNIX and is widely used for software distribution. When using RPM, the term *package* is used to mean a .rpm file or a preassembled unit containing software that is meant to be installed using RPM. RPM is a command executed from the shell:

```
rpm -i [options] [packages]
```

Other setup programs are available for deploying applications on Linux. For example, the Setup Graphic Installer is a graphic installation utility that developers can use to create an easy-to-use installation for all types of applications.

Deployment issues

Beyond the executable file, an application may require a number of supporting files, such as shared object files, initialization files, package files, and helper applications. The process of copying an application's files to a computer and making any needed settings can be automated by an installation program. Following are the main deployment concerns common to nearly all types of applications:

- Using installation programs
- Identifying application files
- Helper applications
- Shared object file locations

Database and Web applications require additional installation steps beyond those that apply to general applications. For additional information, see "Deploying database applications" on page 18-4 and "Deploying Web applications" on page 18-5

Many Kylix applications are dependent upon a certain set of shared libraries—these may include:

- libqt.so.2 (the Qt library)
- libqtintf.so.2 (the Qt interface library)
- libborunwind.so.6 (shared exception handling)
- Runtime packages (such as baseclx)
- Database dbExpress drivers (libsqldb.so, libsqlora.so and so on)

To successfully deploy an application, all required libraries need to shipped and placed in a location where they are accessible to the application at runtime. Which libraries are required will depend on the application. For example, if you are using CLX components, you will need to deploy the two Qt libraries listed above.

Accessibility of libraries is controlled by, essentially, two settings on a Linux system: An entry listing the directory which contains the required libraries in one of two places:

- /etc/ld.so.conf
  or
- LD_LIBRARY_PATH

Typically, /etc/ld.so.conf will already have a number of directories listed, so simply copying the files into an already listed directory would suffice.

In comparison with Windows 2000, the use of /etc/ld.so.conf or a directory listed there-in is equivalent to copying the shared library to the (system-wide) \WinNT\

System32 folder; using LD_LIBRARY_PATH is equivalent to specifying application-specific search paths.

If using the library path environment variable, the directory containing any shared object file that your application is loading needs to be included in the LD_LIBRARY_PATH. LD_LIBRARY_PATH is an environment variable you set to provide the runtime shared library loader an extra set of directories to look for when searching for shared libraries. You can list multiple directories, separating them with a colon (:). For example, if you are deploying an application to /usr/local/app and it uses the shared object /usr/local/app/mylib.so, you would include /usr/local/app in the LD_LIBRARY_PATH.

Using installation programs

Simple Kylix applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs.

Identifying application files

Besides the executable file, a number of other files may need to be distributed with an application.

- Application files
- Package files
- Helper applications
- DLLShared object files
- Initialization files
- Help files

To determine all the files required by your applications, type the following:

```
ldd <application_name>
```

The output lists all files that your application tries to load including operating system files. You do not need to deploy the operating system related files but you do need to include Kylix files or libraries that are referenced as well as files required by your application.

Package files

If the application uses runtime packages, those package files need to be distributed with the application. By convention, package files are typically placed in a lib directory along with other shared objects. This serves as a common location so that multiple applications can access a single instance of the files. For packages you create, it is recommended that you install them in the same directory as the application. Only the .so files need to be distributed.

Note If you are distributing packages to other developers, you need to supply both the .so and the .dcp (Delphi) or .bcp (C++) files

Helper applications

Helper applications are separate programs without which your Kylix application would be partially or completely unable to function. Helper applications may be those supplied by Borland or by third-party products.

If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the documentation provided with the helper program for specific information.

Shared object locations

You can install shared object files used only by a single application in the same directory as the application. Shared objects that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community shared object files on Linux systems is to place them in the /bin directory. Any shared object file that your application loads needs to be in the LD_LIBRARY_PATH.

Deploying database applications

Applications that access databases involve special installation considerations beyond copying the application's executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application's executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require additional handling on installation, because the files that make up the application are typically located on multiple computers.

Kylix applications use *dbExpress* to connect to a database. *dbExpress* is a set of database drivers that provide quick access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfacesclasses. When you deploy your application, you need only include a single shared object (the server-specific driver) with the application files you build. If you are supporting multiple databases, you'll need a shared object file for each database.

If you are using a client dataset (*TClientDataSet* or *TSQLClientDataSet*) or if you are using *TDataSetProvider*, you also need to distribute midas.so.

Connecting to a database

To open a database connection, you must identify both the driver to use and a set of connection parameters to be passed to that driver. The driver is identified by its *DriverName* property, which is the name of an installed *dbExpress* driver. Supported drivers include databases such as InterBase, MySQL, Oracle, Informix, PostgreSQL, or DB2. Two files are connected to the driver name:

- A *dbExpress* driver, which is a shared object file with names such as libsqlib.so, libsqlmys.so, libsqlora.so, libsqlinf.so, libsqlpg.so, or libsqldb2.so
- On the client side, the shared object file provided by the database vendor.

The relationship between these two shared object files and the database name is stored in a file called dbxdrivers. *dbExpress* also lets you define database connection names and save them in a file called dbxconnections. This allows you to deploy your database applications onto systems that access different databases. You only need to deploy the dbxdrivers and the dbxconnections files if you are loading database information at runtime. Details about database connections are covered in "Controlling connections" on page 19-2.

Install the drivers and connections files either in the same directory as your executable or in a .borland subdirectory under the home directory. If you want to share the *dbExpress* drivers and configuration files among several applications that you are developing, put them in the .borland directory. (This is the same place that Kylix places the dbxdrivers and dbxconnections files.)

Updating configuration files

If using shared configuration files, your installation program can use the MergeIniFile utility provided with the product to merge the contents of an existing drivers or connections file with a new or updated one. The syntax is:

```
MergeIniFile( SourceIniFile, TargetIniFile, FOverwrite )
```

The utility returns a count of the number of sections added. FOverwrite is false by default. If set to true, it adds the section even if it exists already and overwrites any settings for the section found in the SourceIniFile. Sections not in the SourceIniFile remain unchanged, and settings not in the configuration file are not deleted.

Deploying Web applications

Some Kylix applications are designed to be run over the World Wide Web, such as those in the form of Apache shared object files and CGI applications. CGI applications can be deployed on any Web server that supports CGI interfaces. The steps for deploying Web applications are the same as those for general applications, except that the application's files are deployed on the Web server. For information on installing general applications, see "Deploying general applications" on page 18-1.

Note To run Apache shared object files, you need to rebuild Apache with the SHARED_CORE rule enabled. This rule forces the Apache core code to be placed into a dynamic shared object (DSO) file. For more information, refer to the Apache web site: www.apache.org.

Here are some special considerations for deploying Web applications:

- For WebSnap applications, make sure the javascript shared object is installed along with the application files on the Web server.
- The correct file permissions for the directories should be set so that the appropriate users can access all needed application and database files.

- The directory containing an application must have the proper permissions assigned to it.
- Apache configuration files must specify the location of CGI or DSO applications. That directory must have read and execute attributes.
- If your application involves write activity, all users coming in from http must have read and write access to the server. Proceed with caution when allowing server access and consult with professionals concerning Linux security.
- For Web Services applications, if you have ADMIN enabled on the server, the server needs write access to the directory where the application is deployed. (This is because a configuration file containing administrative information is created and saved on the server.)

Here are some considerations for deploying Web database applications:

- If your application involves writing to a database, users permitted to access the database need to have appropriate access to it.
- The *dbExpress* driver needs to be installed along with the application files on the Web server.
- The application should not use hard-coded paths for accessing database or other files.

Deploying to Apache servers

WebBroker supports Apache version 1.3.9 and later. If using Apache, you need to be sure to set appropriate directives in the Apache configuration file, called httpd.conf. The SetEnv directive should be set to export the LD_LIBRARY_PATH variable and set it to point to the location of all shared objects required by the application at runtime. Therefore, httpd.conf must contain:

```
SetEnv LD_LIBRARY_PATH <directory>
```

where <directory> is the full path to the Kylix installation directory.

The physical directory must have the ExecCGI option set to allow execution of programs; httpd.conf should contain lines similar to the following:

```
<Directory "/<directory>/httpd/cgi-bin">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
    </Directory>
```

For more information about httpd configuration files on production servers, refer to the Apache web site: www.apache.org.

You also need to set the HOME environment variable in the httpd.conf file to your home directory (for example, /home/username). For database applications, *dbExpress* also requires a .borland directory to be located in the home directory.

Note Apache executes locally on the server within the account specified in the User directive in the httpd.conf file.

Refer also to the Mod101.txt file in the kylix/demos/internet/apachedso directory for an example including the http.conf file.

Programming for varying host environments

Due to the nature of the Linux environment, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer:

- Screen resolutions and color depths
- Fonts
- Helper applications
- DLLShared object file locations

Screen resolutions and color depths

The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware and Xserver that is installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer.

An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways:

- Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is set.
- Design using any screen resolution on the development computer and, at runtime, dynamically resize all forms and objects proportional to the difference between the screen resolutions for the development and deployment computers (a screen resolution difference ratio).
- Design using any screen resolution on the development computer and, at runtime, dynamically resize only the application's forms. Depending on the location of visual controls on the forms, this may require the forms be scrollable for the user to be able to access all controls on the forms.

Considerations when not dynamically resizing

If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution. Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen.

For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the desktop on a computer configured for a 640x480 screen resolution.

Considerations when dynamically resizing forms and controls

If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application:

- The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in pixels. Retrieve the same dimension for the user's computer at runtime using the *TScreen* object's*Height* or *Width*property. Divide the value for the development computer by the value for the user's computer to derive the difference ratio between the two computers' screen resolutions.
- Resize the visual elements of the application (forms and controls) by reducing or increasing the size of the elements and their positions on forms. This resizing is proportional to the difference between the screen resolutions on the development and user computers. Resize and reposition visual controls on forms automatically by setting the *TCustomForm* object's *Scaled* property to true and calling the *TWidgetControl* object's *ScaleBy*method. The *ScaleBy* method does not change the form's height and width, though. Do this manually by multiplying the current values for the *Height* and *Width* properties by the screen resolution difference ratio.
- The controls on a form can be resized manually, instead of automatically with the *ScaleBy* method, by referencing each visual control in a loop and setting its dimensions and position. The *Height* and *Width* property values for visual controls are multiplied by the screen resolution difference ratio. Reposition visual controls proportional to screen resolution differences by multiplying the *Top* and *Left* property values by the same ratio.
- If an application is designed on a computer configured for a higher screen resolution than that on the user's computer, font sizes will be reduced in the process of proportionally resizing visual controls. If the size of the font at design time is too small, the font as resized at runtime may be unreadable. For example, the default font size for a form is 8. If the development computer has a screen resolution of 1024x768 and the user's computer 640x480, visual control dimensions will be reduced by a factor of 0.625 (640 / 1024 = 0.625). The original font size of 8 is reduced to 5 (8 * 0.625 = 5). Text in the application appears jagged and unreadable as it is displayed in the reduced font size.
- Some visual controls, such as *TLabel* and *TEdit*, dynamically resize when the size of the font for the control changes. This can affect deployed applications when forms and controls are dynamically resized. The resizing of the control due to font size changes are in addition to size changes due to proportional resizing for screen

resolutions. This effect is offset by setting the *AutoSize* property of these controls to false.

- Avoid making use of explicit pixel coordinates, such as when drawing directly to a canvas. Instead, modify the coordinates by a ratio proportionate to the screen resolution difference ratio between the development and user computers. For example, if the application draws a rectangle to a canvas ten pixels high by twenty wide, instead multiply the ten and twenty by the screen resolution difference ratio. This ensures that the rectangle visually appears the same size under different screen resolutions.

Accommodating varying color depths

To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.

Fonts

Linux comes with a standard set of fonts, depending on the distribution. When designing an application to be deployed on other computers, realize that not all computers have fonts outside the standard sets.

Text components used in the application should all use fonts that are likely to be available on all deployment computers.

When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment computer. Distribution of third-party fonts may be subject to limitations imposed by the font creator.

Software license requirements

The distribution of some files associated with Kylix applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files:

DEPLOY

The DEPLOY document covers some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with a Kylix application. The DEPLOY document is installed in the main Kylix directory. The topics covered include:

- Executable files, shared object files, and package files
- Components and design-time packages

- Sample images
- Client datasets (*TDataSetProvider*, *TClientDataSet*, or *TSQLDataSet*)

README

The README document contains last minute information about Kylix, possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. The README document is installed in the main Kylix directory.

No-nonsense license agreement

The Kylix no-nonsense license statement (license.txt) covers other legal rights and obligations concerning Kylix.

GPL license agreement

The General Public License (GPL) agreement specifies information concerning open source licensing terms for using CLX to develop open sourced applications.

Third-party product documentation

Redistribution rights for third-party components, utilities, helper applications, database engines, and other products are governed by the vendor supplying the product. Consult the documentation for the product or the vendor for information regarding the redistribution of the product with Kylix applications prior to distribution.

Part
II

Developing database applications

The chapters in “Developing Database Applications” present concepts and skills necessary for creating database applications.

Note Database components are not available in all editions.

Chapter

19

Designing database applications

Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

The IDE provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

You have several choices available when designing a database application. This chapter describes the various architectures available. Consider each type of architecture's advantages and disadvantages to choose the approach that best suits your needs. You may want to start with a simple approach, and then scale it up later to another, more powerful architecture. The IDE's database support makes this easy because the same components are used in most of the architectures.

Using databases

The components on the dbExpress page of the Component palette let your application connect to and read from databases. These components use *dbExpress* to access database information, which they make available to other components that repackage and buffer the information, or display it to end-users.

dbExpress is a set of drivers for different types of databases. While all types of databases contain tables which store information, different types support additional features such as

- Database security
- Transactions
- Referential integrity, stored procedures, and triggers

Types of databases

Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. The IDE provides support for two types of relational database server:

- **Remote database servers** typically reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called SQL servers. (Another name is Remote Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique "dialect" of SQL. Examples of SQL servers include InterBase, Oracle, Informix, DB2, PostgreSQL, and MySQL.
- **Local databases** reside on your local drive or on a local area network. They often have proprietary APIs for accessing the data. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases. The IDE provides support for two types of local databases: Local InterBase, which is a local version of the InterBase server, and a proprietary file format for the data stored in a client dataset.

Applications that use local databases are called **single-tiered applications** because the application and the database share a single file system. Applications that use remote database servers are called **two-tiered applications** or **multi-tiered applications** because the application and the database operate on independent systems (or tiers).

Choosing the type of database to use depends on several factors. For example, your data may already be stored in an existing database. If you are creating the database tables your application uses, you may want to consider the following questions:

- How many users will be sharing these tables? Remote database servers are designed for access by several users at the same time. They provide support for multiple users through a mechanism called transactions. Some local databases (such as Local InterBase) also provide transaction support, but many only provide file-based locking mechanisms, and some (such as client dataset files) provide no multi-user support at all.
- How much data will the tables hold? Remote database servers can hold more data than local databases. Some remote database servers are designed for warehousing large quantities of data while others are optimized for other criteria (such as fast updates).
- What type of performance (speed) do you require from the database? Local databases are usually faster than remote database servers because they reside on the same system as the database application. Different remote database servers are optimized to support different types of operations, so you may want to consider performance when choosing a remote database server.

- What type of support will be available for database administration? Local databases require less support than remote database servers. Typically, they are less expensive to operate because they do not require separately installed servers or expensive site licenses.

Database security

Databases often contain sensitive information. Most SQL servers provide security at multiple levels. Typically, they require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers, see "Controlling server login" on page 21-5.

When designing database applications, you must consider what type of authentication is required by your database server. Often, applications are designed to hide the explicit database login from users, who need only log in to the application itself. If you do not want to require your users to provide a database password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can't be breached by reading the password from the application.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password which is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use a protocol such as HTTPs to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

Transactions

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Transactions ensure that

- All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial or uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**.
- Committed updates to records survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**.

Thus, transactions protect against hardware failures that occur in the middle of a database command or set of commands. Transactional logging allows you to recover the durable state after disk media failures. Transactions also form the basis of multi-user concurrency control on SQL servers. When each user interacts with the database only through transactions, one user's commands can't disrupt the unity of another user's transaction. Instead, the SQL server schedules incoming transactions, which either succeed as a whole or fail as a whole.

Transaction support is provided by most SQL servers. However, some servers, such as MySQL, provide no support for transactions. Similarly, if you are only using client datasets and storing the data in files, there is no transaction support.

For details on using transactions in your database applications, see "Managing transactions" on page 21-7.

Referential integrity, stored procedures, and triggers

All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include

- **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.
- **Stored procedures.** Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and sometimes return sets of records (datasets).
- **Triggers.** Triggers are sets of SQL statements that are automatically executed in response to a particular command.

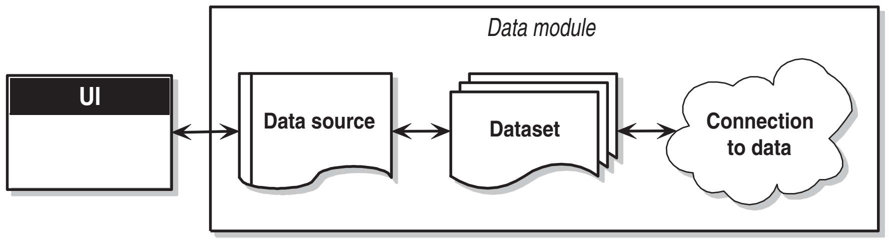
Database architecture

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application.

General structure

While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in Figure 19.1:

Figure 19.1 Generic database architecture



The user interface form

It is a good idea to isolate the user interface on a form that is completely separate from the rest of the application. This has several advantages. By isolating the user interface from the components that represent the database information itself, you introduce a greater flexibility into your design: Changes to the way you manage the database information do not require you to rewrite your user interface, and changes to the user interface do not require you to change the portion of your application that works with the database. In addition, this type of isolation lets you develop common forms that you can share between multiple applications, thereby providing a consistent user interface. By storing links to well-designed forms in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms also makes it possible for you to develop corporate standards for application interfaces. For more information about creating the user interface for a database application, see Chapter 20, “Using data controls”.

The data module

If you have isolated your user interface into its own form, you can use a data module to house the components that represent database information (datasets), and the components that connect these datasets to the other parts of your application. Like the user interface forms, you can share data modules in the Object Repository so that they can be reused or shared between applications.

The data source

The first item in the data module is a data source. The data source acts as a conduit between the user interface and a dataset that represents information from a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control.

The dataset

The heart of your database application is the dataset. This component represents a set of records from the underlying database. These records can be the data from a single database table, a subset of the fields or records in a table, or information from more than one table joined into a single view. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. When the underlying database changes, you might need to alter the way the dataset component specifies the data it contains, but the rest of your application can continue to work without alteration. For more information on the common properties and methods of datasets, see Chapter 22, "Understanding datasets".

The data connection

Different types of datasets use different mechanisms for connecting to the underlying database information. These different mechanisms, in turn, make up the major differences in the architecture of the database applications you can build. The IDE provides support for two types of datasets:

- **Client datasets**, which buffer data in memory so that you can navigate through records more easily and perform operations on the data such as filtering records or maintaining aggregate values that summarize the data. Client datasets provide support for applying the updates in the in-memory cache back to the underlying database. Because client datasets cache the records in memory, they can only hold a limited number of records. There are two types of client datasets: generic client datasets and SQL client datasets. Generic client datasets access their data by working directly with a file stored locally on disk, connecting to another dataset in the same data module, or connecting to an application server on another (server) machine. SQL client datasets can use a file stored locally on disk or connect to a database server. For more information about client datasets, see Chapter 25, "Using client datasets".
- **Unidirectional datasets**, which can read data that is described by an SQL query or that is returned by a stored procedure. Unidirectional datasets do not buffer data, so they are less flexible than client datasets. The only way to navigate through the records of a unidirectional dataset is to iterate through them in the order specified by the ORDER BY clause of the SQL query. Further, you can't use a unidirectional dataset to update data. However, unidirectional datasets provide fast access to information from a database server, and can represent far larger sets of data than client datasets. Unidirectional datasets always fetch their data using an SQL connection component. For more information about unidirectional data sets, see Chapter 24, "Using unidirectional datasets."

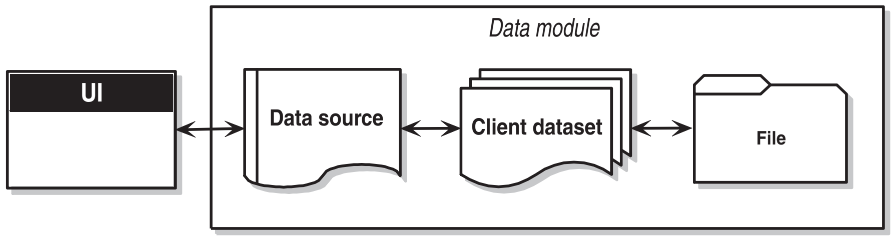
In addition, you can create your own custom datasets**.** These are descendants of *TDataSet* that represent a body of data that you create or access in code you write. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the CLX data controls to build your user interface.

The following topics describe the most common ways to use client datasets, local datsets, and unidirectional datasets in database applications.

Using a client dataset with data stored on disk

The simplest form of database application you can write does not use a database server at all. Instead, it uses the ability of client datasets to save themselves to a file and to load the data from a file. The architecture for this type of application is illustrated in Figure 19.2:

Figure 19.2 Architecture of a file-based database application



This simple file-based architecture is a single-tiered application. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

The file-based approach has the benefit of simplicity. There is no database server to install, configure, or deploy (although the client dataset does require midas.so). There is no need for site licenses or database administration.

However, there is no support for multiple users. The dataset must be dedicated entirely to the application. Data is saved to files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other’s data files.

Because there is no separate database server, you are responsible for creating the underlying table yourself. Once this table is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. When beginning a file-based database application, you may want to first create and save empty files for your datasets before writing the application itself. This way, you do not need to define the metadata for your client datasets in the final application. For more information about creating the tables for file-based applications, see “Creating a new dataset” on page 25-41.

In this file-based model, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the *MergeChangeLog* method. For more information about the change log, see “Editing data” on page 25-16.

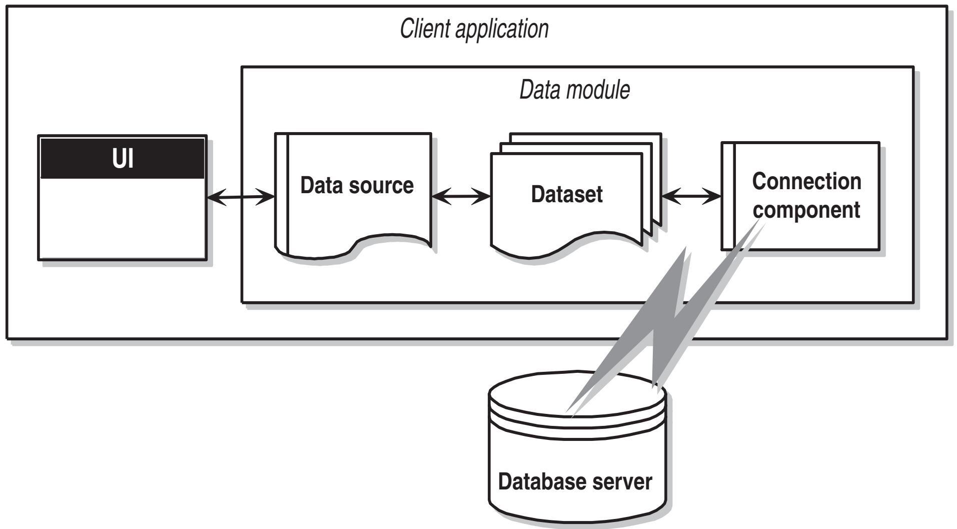
Even when you have merged changes into the data of the client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the *SaveToFile* method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table. When you want to read a table previously written using the *SaveToFile* method, use the *LoadFromFile* method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

If you always load to and save from the same file, you can use the *FileName* property instead of the *SaveToFile* and *LoadFromFile* methods. When *FileName* is set to a valid file name, the data is automatically loaded from the file when the client dataset is opened and saved to the file when the client dataset is closed.

Using a unidirectional dataset directly

The IDE requires a unidirectional dataset to connect to a database server. Thus, the simplest way to use data from an SQL server is to connect this unidirectional dataset directly to the user interface. The architecture for this type of application is illustrated in Figure 19.3.

Figure 19.3 Architecture of a unidirectional database application



This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such as Local InterBase or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

This model provides very fast access when working with a database server. There is no overhead for buffering data or managing metadata. Because no data is buffered in memory, you can use this model with arbitrarily large datasets.

However, when using this model, your application can only view one record at a time, and you can only progress through records in order. In addition, the data is read-only: there is no built-in way to post changes back to the database.

This model is best suited to applications that read the data and analyze it to present a set of summary statistics or print a report. It can, however, be used to present users with values through which to browse.

The dataset in this model is a unidirectional dataset (*TSQLDataSet*, *TSQLQuery*, *TSQLTable*, or *TSQLStoredProcedure*). Unidirectional datasets execute an SQL statement on the database server and, if the SQL statement returns data (that is, if it is a SELECT statement), obtains a unidirectional cursor for accessing the resulting data.

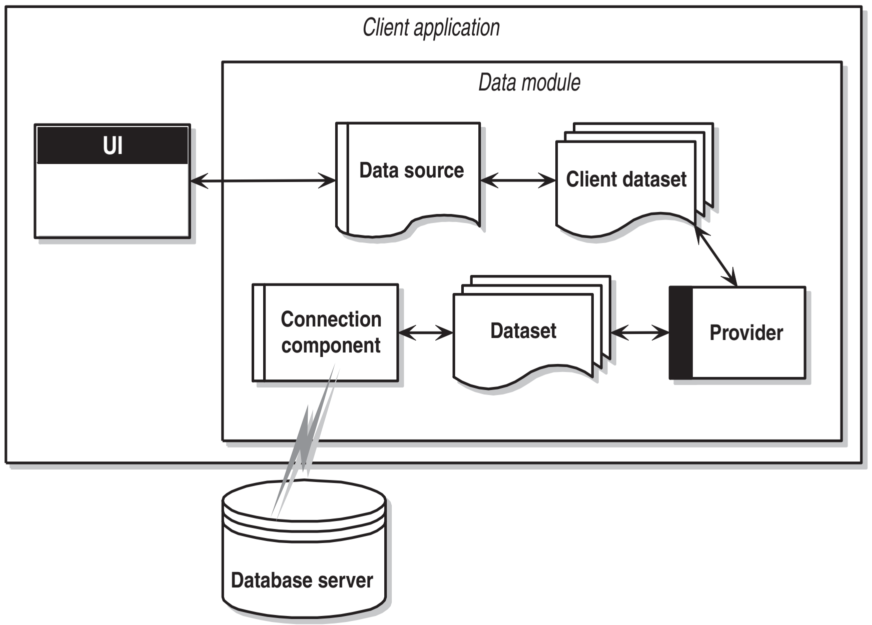
In order to connect a unidirectional dataset to a server, the data module must contain a SQL connection component (*TSQLConnection*). The unidirectional dataset is linked to this SQL connection component via its *SQLConnection* property. The SQL connection component represents the connection to the database server, and requires dbExpress. Double-click on the SQL connection component to identify the server in the Connection Editor. This editor lets you specify a connection name, which represents a named set of configuration parameters for a specific driver. Each named connection configuration is defined in a special file called connections.ini. You can use the Connection Editor to select a configuration, modify a named configuration, or define a new one.

Note You may need to purchase some of the drivers separately.

Using a client dataset to buffer records

To display multiple records, use data-aware controls to update data, move backwards through records, or move to records that meet a specific criterion, you must use a client dataset. On the other hand, you must use a unidirectional dataset to connect to a database server. You can combine these to create an application that has the flexibility of a client dataset but works with data from a database server. The architecture for this type of application is illustrated in Figure 19.4:

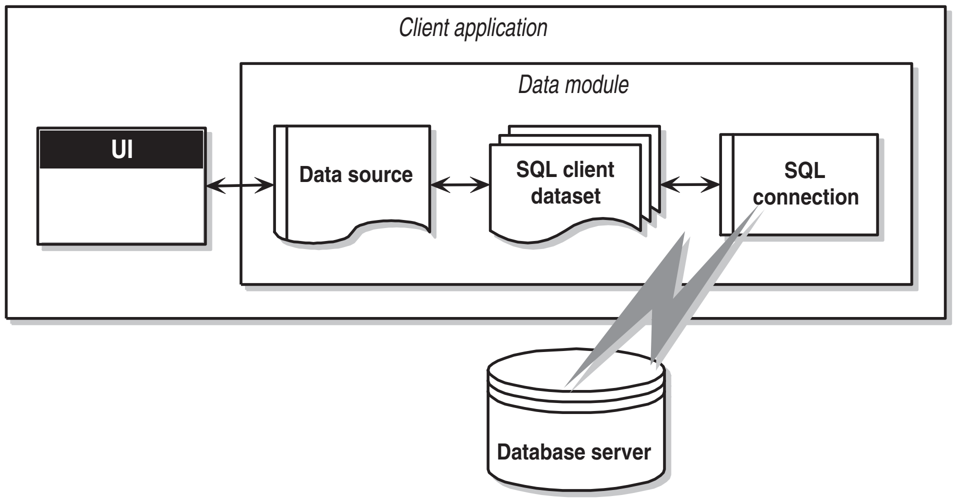
Figure 19.4 Architecture combining a client dataset and a unidirectional dataset



The connection between the client dataset and the unidirectional dataset is provided by a dataset provider. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets (which client datasets create) back to a database server. To link the client dataset to the provider, set its *ProviderName* property to the name of the provider component. The provider and the client dataset must be in the same data module. To link the provider to the unidirectional dataset, set its *DataSet* property to the unidirectional dataset.

To simplify this architecture, the IDE includes a special type of client dataset, called an SQL client dataset, that contains its own, internal provider and unidirectional dataset. By using an SQL client dataset, the preceding arrangement is simplified to look like Figure 19.5.

Figure 19.5 Architecture using a client dataset with an internal unidirectional dataset



When using an SQL client dataset, you need not explicitly add a dataset provider and unidirectional dataset to the data module: these components are internal to the SQL client dataset. This simplifies your application, but at the cost of some control:

- Because the unidirectional dataset is internal to the SQL client dataset component, you can't link two unidirectional datasets in a master/detail relationship to obtain nested detail sets. (You can, however, link two SQL client datasets into a master/ detail relationship.)
- The SQL client dataset component does not surface most of the events that occur on its internal dataset provider. However, in most cases, these events are used in multi-tiered applications, and are not needed for two-tiered applications.

Both of these models represent either a single-tiered or two-tiered application, depending on whether the database server is a local database such as Local Interbase or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

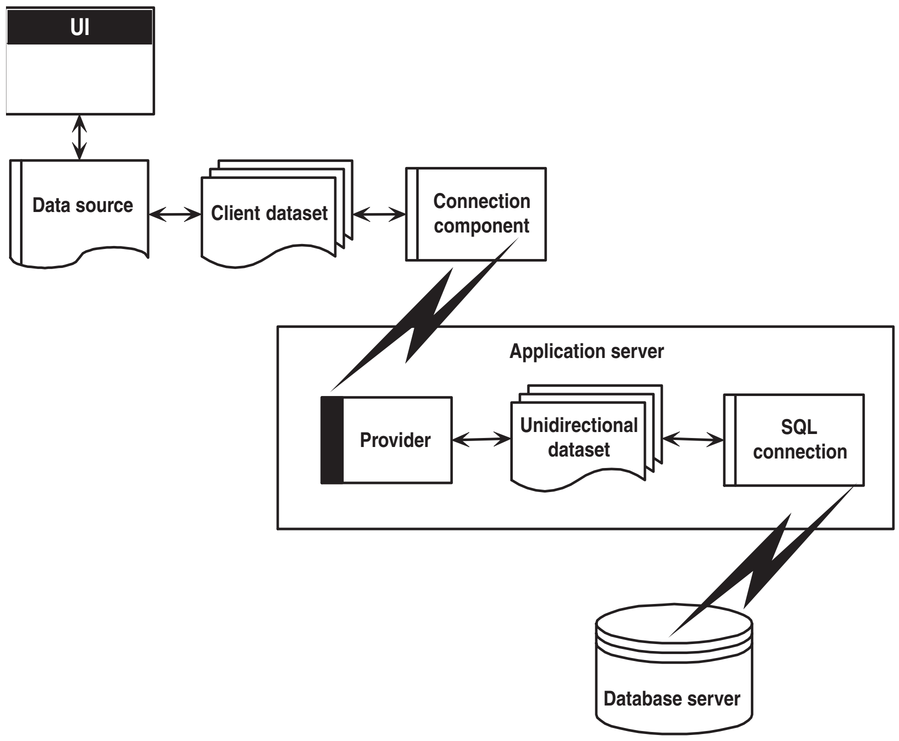
These models represent a hybrid of the two previous models. They use a (possibly internal) unidirectional dataset to access the database server and fetch records, while the client dataset buffers the records to provide more flexibility and applies updates back to the database server when the user edits the data.

Client datasets automatically handle all the details necessary for fetching, displaying, and navigating through database records. To apply user edits back to the database, you need only call the client dataset's *ApplyUpdates* method. (Note that, unlike when using a client dataset with data stored on disk, you must not call the *MergeChangeLog* method, or the client can't produce accurate delta packets for the provider.)

Using a multi-tiered architecture

When the database information includes complicated relationships between several tables, or the number of clients grows, you may want to use a multi-tiered model. Multi-tiered applications have middle tiers between the client application and the database server. This architecture is illustrated in Figure 19.6:

Figure 19.6 Multi-tiered database architecture



The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data while ensuring that the data logic is consistent. It also allows for smaller client applications because much of the processing is off-loaded onto the middle tier. These smaller client applications are easier to install, configure, and maintain. Multi-tiered applications can also improve performance by spreading the data-processing tasks over several systems.

The multi-tiered architecture is very similar to using a client dataset to buffer records from an external provider and unidirectional dataset. It differs mainly in that the unidirectional dataset that connects to the database server and the provider that acts as an intermediary between that unidirectional dataset and the client dataset have both moved to a separate application. That separate application is called the application server (or sometimes the “remote data broker”).

Because the provider has moved to a separate application, the client dataset can no longer connect to the unidirectional dataset by simply setting its *ProviderName* property. In addition, it must use some type of connection component to locate and connect to the application server. This component is a descendant of *TCustomRemoteServer* such as *TSoapConnection*. Link the client dataset to its connection component by setting the *RemoteServer* property.

Note You may need to purchase the components to connect a client dataset with a remote application server separately.

The connection component establishes a connection to the application server and returns an interface that the client dataset uses to call the provider specified by its *ProviderName* property. Each time the client dataset calls the application server, it passes the value of *ProviderName*, and the application server forwards the call to the provider.

Combining approaches

The previous sections describe several architectures you can use when writing database applications. There is no reason, however, why you can’t combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful.

For example, you can combine the disk-based architecture described in “Using a client dataset with data stored on disk” on page 19-7 with another approach such as “Using a client dataset to buffer records” on page 19-9 or “Using a multi-tiered architecture” on page 19-12. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing).

The briefcase model is useful in a situation such as the following: An on-site company database contains customer contact data that sales representatives can use and update in the field. While on-site, sales representatives download information from the database. Later, they work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales reps return on-site, they upload their data changes to the company database for everyone to use.

When operating on site, the client dataset in a briefcase model application fetches its data from a provider. The dataset is therefore connected to the database server and can, through the provider, fetch server data and send updates back to the server. Before disconnecting from the provider, the client dataset saves its snapshot of the information to a file on disk. While offsite, the client dataset loads its data from the file, and saves any changes back to that file. Finally, back on-site, the client dataset reconnects to the provider so that it can apply its updates to the database server or refresh its snapshot of the data.

Chapter

20

Using data controls

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users.

The data-aware controls you add to your user interface depend on several factors, including the following:

- The type of data you are displaying. You can choose between controls that are designed to display and edit plain text, controls that work with formatted text, controls for graphics, multimedia elements, and so on. Controls that display different types of information are described in "Displaying a single record" on page 20-8.
- How you want to organize the information. You may choose to display information from a single record on the screen, or list the information from multiple records using a grid. "Choosing how to organize the data" on page 20-8 describes some of the possibilities.
- The type of dataset that supplies data to the controls. You want to use controls that reflect the limitations of the underlying dataset. For example, you would not use a grid with a unidirectional dataset because unidirectional datasets can only supply a single record at a time.
- How (or if) you want to let users navigate through the records of datasets and add or edit data. You may want to add your own controls or mechanisms to navigate and edit, or you may want to use a built-in control such as a data navigator. For more information about using a data navigator, see "Navigating and manipulating records" on page 20-26.

Regardless of the data-aware controls you choose to add to your interface, certain common features apply. These are described below.

Using common data control features

The following tasks are common to most data controls:

- Associating a data control with a dataset
- Editing and updating data
- Disabling and enabling data display
- Refreshing data display
- Enabling mouse, keyboard, and timer events

Data controls generally let you display and edit fields of data associated with the current record in a dataset. Table 20.1 summarizes the data controls that appear on the Data Controls page of the Component palette.

Table 20.1 Data controls

Data control	Description
TDBGrid	Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records.
TDBNavigator	Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display.
TDBText	Displays data from a field as a label.
TDBEdit	Displays data from a field in an edit box.
TDBMemo	Displays data from a memo or BLOB field in a scrollable, multi-line edit box.
TDBImage	Displays graphics from a data field in a graphics box.
TDBListBox	Displays a list of items from which to update a field in the current data record.
TDBComboBox	Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
TDBCheckBox	Displays a check box that indicates the value of a Boolean field.
TDBRadioGroup	Displays a set of mutually exclusive options for a field.
TDBLookupListBox	Displays a list of items looked up from another dataset based on the value of a field.
TDBLookupComboBox	Displays a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware edit box.

Data controls are data-aware at design time. When you associate the data control with an active dataset while building an application, you can immediately see live data in the control. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For more information about the Fields editor, see "Creating persistent fields" on page 23-4.

At runtime, data controls display data and, if your application, the control, and the dataset all permit it, a user can edit data through the control.

Associating a data control with a dataset

Data controls connect to datasets by using a data source. A data source component (*TDataSource*) acts as a conduit between the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

Note Data source components are also required for linking unnested datasets in master-detail relationships.

To associate a data control with a dataset,

1 Place a dataset in a data module (or on a form), and set its properties as appropriate.

2 Place a data source in the same data module (or form). Using the Object Inspector, set its *DataSet* property to the dataset you placed in step 1.

3 Place a data control from the Data Access page of the Component palette onto a form.

4 Using the Object Inspector, set the *DataSource* property of the control to the data source component you placed in step 2.

5 Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property. This step does not apply to *TDBGrid* and *TDBNavigator* because they access all available fields in the dataset.

6 Set the *Active* property of the dataset to true to display data in the control.

Changing the associated dataset at runtime

In the preceding example, the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

D

Delphi example

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
```

C++ example

```
if (CustSource->DataSet == Customers)
  CustSource->DataSet = Orders;
else
  CustSource->DataSet = Customers;
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on two forms. For example:

Delphi example

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.DataSet := Form1.ClientDataSet1;
end;
```

C++ example

```
void __fastcall TForm2::FormCreate(TObject *Sender)
{
  DataSource1->DataSet = Form1->ClientDataSet1;
}
```

Enabling and disabling the data source

The data source has an *Enabled* property that determines if it is connected to its dataset. When *Enabled* is true, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to false. When *Enabled* is false, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to true. It is recommended, however, to control access to a dataset through a dataset component's *DisableControls* and *EnableControls* methods because they affect all attached data sources.

Responding to changes mediated by the data source

Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response.

The *OnDataChange* event occurs whenever the data in a record may change, including field edits or when the cursor moves to a new record. This event is useful for making sure the control reflects the current field values in the dataset, because it is triggered by all changes. Typically, an *OnDataChange* event handler refreshes the value of a non-data-aware control that displays field data.

The *OnUpdateData* event occurs when the data in the current record is about to be posted. For instance, an *OnUpdateData* event occurs after *Post* is called, but before the data is actually posted to the change log.

The *OnStateChange* event occurs when the state of the dataset changes. When this event occurs, you can examine the dataset's *State* property to determine its current state.

For example, the following *OnStateChange* event handler enables or disables buttons or menu items based on the current state:

Delphi example

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  ⋮
end;
```

C++ example

```
void __fastcall TForm1::DataSource1StateChange(TObject *Sender)
{
  CustTableActivateBtn->Enabled = (CustTable->State == dsInactive);
  CustTableEditBtn->Enabled = (CustTable->State == dsBrowse);
  CustTableCancelBtn->Enabled = (CustTable->State == dsInsert ||
                                 CustTable->State == dsEdit ||
                                 CustTable->State == dsSetKey);
  ⋮
}
```

Note For more information about dataset states, see "Determining and setting dataset states" on page 22-3.

Editing and updating data

All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it.

Note Unidirectional datasets never permit users to edit and update data.

Enabling editing in controls on user entry

A dataset must be in *dsEdit* state to permit editing to its data. If the data source's *AutoEdit* property is true (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data.

If *AutoEdit* is false, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a *TDBNavigator* control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see "Navigating and manipulating records" on page 20-26. Alternately, you can write code that calls the dataset's *Edit* method when you want to put the dataset into edit mode.

Editing data in a control

A data control can only post edits to its associated dataset if the dataset's *CanModify* property is true. *CanModify* is always false for unidirectional datasets. Client datasets have a *ReadOnly* property that lets you specify whether *CanModify* is true.

Note Whether a client dataset can update data does not depend on whether its source dataset permits updates, but rather, it depends on whether the underlying database table permits updates. Thus, a client dataset can set *ReadOnly* to false, even if it fetches its data from a unidirectional dataset, which is always read-only. This is because client datasets can apply updates directly to the underlying database server.

Even if the dataset's *CanModify* property is true, the *Enabled* property of the data source that connects the dataset to the control must be true as well before the control can post updates back to the database table. The *Enabled* property of the data source determines whether the control can display field values from the dataset, and therefore also whether a user can edit and post values. If *Enabled* is true (the default), controls can display field values.

Finally, you can control whether the user can even enter edits to the data that is displayed in the control. The *ReadOnly* property of the data control determines if a user can edit the data displayed by the control. If false (the default), users can edit data. Clearly, you will want to ensure that the control's *ReadOnly* property is true when the dataset's *CanModify* property is false. Otherwise, you give users the false impression that they can affect the data in the underlying database table.

In all data controls except *TDBGrid*, when you modify a field, the modification is copied to the underlying dataset when you *Tab* from the control. If you press *Esc* before you *Tab* from a field, the data control abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In *TDBGrid*, modifications are posted when you move to a different record; you can press *Esc* in any record of a field before moving to another record to cancel all changes to the record.

Client datasets cache all modifications to the data in a change log. These modifications are not applied to the underlying database table until you call the client dataset's *ApplyUpdates* method or, if you are storing data in a file on disk, until you call the client dataset's *MergeChangeLog* method and then save the client dataset.

Disabling and enabling data display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's *EnableControls* method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a **try...finally** statement (C++ **try...__finally**) so that you can re-enable controls even if an exception occurs during processing. The **finally** clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

Delphi example

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

C++ example

```
CustTable->DisableControls();
try
{
  // cycle through all records of the dataset
  for (CustTable->First(); !CustTable->EOF; CustTable->Next())
  {
  // Process each record here
  ⋮
  }
}
__finally
{
  CustTable->EnableControls();
}
```

Refreshing data display

The *Refresh* method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. However, before you refresh the dataset, be sure to apply any updates the dataset has currently cached. Client datasets raise an exception if you attempt to refresh the data before applying pending updates.

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

Enabling mouse, keyboard, and timer events

The *Enabled* property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is true.

To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to false. When *Enabled* is false, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Choosing how to organize the data

When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information.

One of the first decisions to make is whether you want to display a single record at a time, or multiple records.

In addition, you will want to add controls to navigate and manipulate records. The *TDBNavigator* control provides built-in support for many of the functions you may want to perform.

Displaying a single record

In many applications, you may only want to provide information about a single record of data at a time. For example, an order-entry application may display the information about a single order without indicating what other orders are currently logged. This information probably comes from a single record in an orders dataset. If you are connecting your user interface directly to a unidirectional dataset, a single-record user interface is the only available possibility.

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls page of the Component palette provides a wide selection of controls to represent different kinds of fields. These controls are typically data-aware versions of other controls that are available on the component palette. For example, the *TDBEdit* control is a data-aware version of the standard *TEdit* control which enables users to see and edit a text string.

Which control you use depends on the type of data (text, formatted text, graphics, boolean information, and so on) contained in the field.

Displaying data as labels

TDBText is a read-only control similar to the *TLabel* component on the Standard page of the Component palette. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could be used to display the zip code field that matches the address entered by the user.

TDBText gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

Note When you place a *TDBText* component on a form, make sure its *AutoSize* property is true (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is false, and the control is too small, data display is clipped.

Displaying and editing fields in an edit box

TDBEdit is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques.

For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open *TSQLClientDataSet* called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows:

- *DataSource*: CustomersSource
- *DataField*: CustNo

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime.

Displaying and editing text in a memo control

TDBMemo is a data-aware component—similar to the standard *TMemo* component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields.

By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the memo control to true. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to true. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for *MaxLength* is 0, meaning that there is no character limit other than that imposed by the operating system.

Several properties affect how the database memo appears and how text is entered. To prevent word wrap, set the *WordWrap* property to false. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*. To change the font of the text, use the *Font* property.

At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to false, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing graphics fields in an image control

TDBImage is a data-aware control that displays graphics contained in BLOB fields.

By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the clipboard using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods. You can, instead, supply your own editing methods attached to the event handlers for the control.

By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the *Stretch* property to true to resize the graphic to fit within an image control as it is resized.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBImage* has an *AutoDisplay* property that controls whether the accessed data should automatically displayed. If you set *AutoDisplay* to false, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing data in list and combo boxes

There are four data controls that provide the user with a set of default data values to choose from at runtime. These are data-aware versions of standard list box and combo box controls:

- *TDBListBox*, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.
- *TDBComboBox*, which combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.
- *TDBLookupListBox*, which behaves like *TDBListBox* except the list of display items is looked up in another dataset.
- *TDBLookupComboBox*, which behaves like *TDBComboBox* except the list of display items is looked up in another dataset.

Note At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing 'ROB' selects the first item in the list box beginning with the letters 'ROB'. Typing an additional 'E' selects the first item starting with 'ROBE', such as 'Robert Johnson'. The search is case-insensitive. *Backspace* and *Esc* cancel the current search string (but leave the selection intact), as does a two second pause between keystrokes.

Using TDBListBox and TDBComboBox

When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the *Items* property in the Object Inspector. Then type in the items that you want to have appear in the list. At runtime, use the methods of the *Items* property to manipulate its string list. You can specify that the items in the list should be displayed in alphabetical order by setting the *Sorted* property to true.

When a *TDBListBox* or *TDBComboBox* control is linked to a field through its *DataField* property, the field value appears selected in the list. If the current value is not in the list, no item appears selected. However, *TDBComboBox* displays the current value for the field in its edit box, regardless of whether it appears in the *Items* list.

The items in the list of *TDBListBox* or the drop-down list of *TDBComboBox* each have the height specified by *ItemHeight*. To ensure that the bottom on the list is fully visible in *TDBListBox*, you may therefore want to set the *Height* property to a multiple of *ItemHeight*. On *TDBComboBox*, this is not necessary, because the size of the drop-down list is not controlled by the *Height* property. Instead, you can set the *DropDownCount*: the maximum number of items displayed in the list. If the number of items in the list exceeds the size of the list (as determined by *Height* on *TDBListBox* or *DropDownCount* on *TDBComboBox*), the user can scroll the list.

For *TDBComboBox*, the *Style* property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The possible values of *Style* are as follows:

- *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
- *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
- *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps) or to use different fonts for individual items in the list.

Displaying and editing data in lookup list and combo boxes

Lookup list boxes and lookup combo boxes (*TDBLookupListBox* and *TDBLookupComboBox*) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

These lookup controls derive the list of display items from one of two sources:

- **A lookup field defined for a dataset.** To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field. (This process is described in "Defining a lookup field" on page 23-8). To specify the lookup field for the list box items,
  - **a** Set the *DataSource* property of the list box to the data source for the dataset containing the lookup field to use.
  - **b** Choose the lookup field to use from the drop-down list for the *DataField* property.

  When you activate a table associated with a lookup control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

- **A secondary data source, data field, and key**. If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item. To specify a secondary data source for list box items,
  - **a** Set the *DataSource* property of the list box to the data source for the control.
  - **b** Choose a field into which to insert looked-up values from the drop-down list for the *DataField* property. The field you choose cannot be a lookup field.
  - **c** Set the *ListSource* property of the list box to the data source for the dataset that contain the field whose values you want to look up.
  - **d** Choose a field to use as a lookup key from the drop-down list for the *KeyField* property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
  - **e** Choose a field whose values to return from the drop-down list for the *ListField* property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

  When you activate a table associated with a lookup control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

To specify the number of items that appear at one time in a *TDBLookupListBox* control, use the *RowCount* property. The height of the list box is adjusted to fit this row count exactly.

To specify the number of items that appear in the drop-down list of *TDBLookupComboBox*, use the *DropDownRows* property instead.

Note You can also set up a column in a data grid to act as a lookup combo box. For information on how to do this, see "Defining a lookup list column" on page 20-21.

Handling Boolean field values with check boxes

TDBCheckBox is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of *ValueChecked* and *ValueUnchecked* properties. If the field value matches the *ValueChecked* property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control is unchecked.

Note The values in *ValueChecked* and *ValueUnchecked* cannot be identical.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to "true," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

```
DBCheckBox1->ValueChecked = "true;Yes;On";
```

If the field for the current record contains values of "true," "Yes," or "On," then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to false, but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is true, and it is unchecked if the contents of the field is false. In this case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

Restricting field values with radio controls

TDBRadioGroup is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The *Items* property determines the radio buttons that appear in the group. *Items* is a string list. One radio button is displayed for each string in *Items*, and each string appears to the right of a radio button as the button's label.

If the current value of a field associated with a radio group matches one of the strings in the *Items* property, that radio button is selected. For example, if three strings, "Red," "Yellow," and "Blue," are listed for *Items*, and the field for the current record contains the value "Blue," then the third button in the group appears selected.

Note If the field does not match any strings in *Items*, a radio button may still be selected if the field matches a string in the *Values* property. If the field for the current record does not match any strings in *Items* or *Values*, no radio button is selected.

The *Values* property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose *Items* contains "Red," "Yellow," and "Blue," and *Values* contains "Magenta," "Yellow," and "Cyan." If a user selects the button labeled "Red," "Magenta" is posted to the database.

If strings for *Values* are not provided, the *Item* string for a selected radio button is returned to the database when a record is posted.

Displaying multiple records

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in "Viewing and editing data with TDBGrid" on page 20-15.

Note You can't display multiple records when using a unidirectional dataset.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

- **Master-detail forms:** You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see "Setting up master/

detail relationships" on page 24-15 and "Making the client dataset a detail of another dataset" on page 25-12.

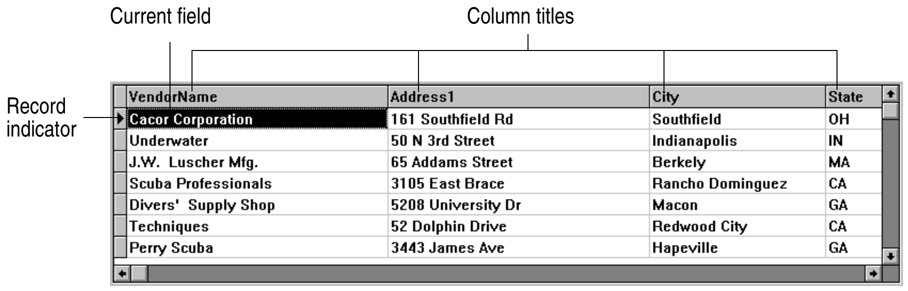
- **Drill-down forms**: In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

Tip It is generally not a good idea to combine these two approaches on a single form. While the result can sometimes be effective, it can be confusing for users to understand the data relationships.

Viewing and editing data with TDBGrid

A *TDBGrid* control lets you view and edit records from a dataset in a tabular grid format.

Figure 20.1 TDBGrid control



VendorName	Address1	City	State
Cacor Corporation	161 Southfield Rd	Southfield	OH
Underwater	50 N 3rd Street	Indianapolis	IN
J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
Divers' Supply Shop	5208 University Dr	Macon	GA
Techniques	52 Dolphin Drive	Redwood City	CA
Perry Scuba	3443 James Ave	Hapeville	GA

Three factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see "Creating a customized grid" on page 20-17.
- Creation of persistent field components for the dataset displayed in the grid. For information about creating persistent field components using the Fields editor, see Chapter 23, "Working with field components."
- The dataset's *ObjectView* property setting for grids displaying nested dataset or array fields. For information on displaying such composite fields in a grid, see "Displaying composite fields" on page 20-22.

A grid control has a *Columns* property that is itself a wrapper on a *TDBGridColumns* object. *TDBGridColumns* is a collection of *TColumn* objects representing all of the columns in a grid control. You can use the Columns editor to set up column attributes at design time, or use the *Columns* property of the grid to access the properties, events, and methods of *TDBGridColumns* at runtime.

Using a grid control in its default state

The *State* property of the grid's *Columns* property indicates whether persistent column objects exist for the grid. *Columns-.State* (Delphi) or *Columns->State* (C++) is a runtime-only property that is automatically set for a grid. The default state is *csDefault*, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid's dataset, or, if there are no persistent field components, by a default set of display characteristics.

When the grid's *Columns.State* (Delphi) or *Columns->State* (C++) property is *csDefault*, grid columns are dynamically generated from the visible fields of the dataset and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with a field component. Property changes to field components immediately show up in the grid.

Using a grid control with dynamically-generated columns is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to accommodate different datasets. A single grid with dynamically-generated columns can display a Local InterBase table at one moment, then switch to display the results of a MySQL query when the grid's *DataSource* property changes or the *DataSet* property of the data source changes.

You can change the appearance of a dynamic column at design time or runtime, but what you are actually modifying are the corresponding properties of the field component displayed in the column. Properties of dynamic columns exist only so long as a column is associated with a particular field in a single dataset. For example, changing the *Width* property of a column changes the *DisplayWidth* property of the field associated with that column. Changes made to column properties that are not based on field properties, such as *Font*, exist only for the lifetime of the column.

If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, all dynamic columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

Note Changing a grid's *Columns.State* (Delphi) or *Columns->State* (C++) property to *csDefault* at runtime deletes all column objects in the grid (even persistent columns), and rebuilds dynamic columns based on the visible fields of the grid's dataset.

Creating a customized grid

A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

Understanding persistent columns

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (the associated field or the grid itself) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes. Once you assign a value to a column property, it no longer changes when its default source changes.

For example, the default source for a column title caption is an associated field's *DisplayLabel* property. If you modify the *DisplayLabel property*, the column title reflects that change immediately. If you then assign a string to the column title's caption, the title caption is independent of the associated field's *DisplayLabel* property. Subsequent changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns need not be associated with field objects at all. If a persistent column's *FieldName* property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's *Field* property is NULL and the column is drawn with blank cells. If you override the cell's default drawing method, you can display your own custom information in the blank cells. For example, you can use a blank column to display aggregated values on the last record of a group of records that the aggregate summarizes. Another possibility is to display a bitmap or bar chart that graphically depicts some aspect of a record's data.

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

Note Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's *FieldCount* property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's *SelectedField* property is NULL and the *SelectedIndex* property is –1.

Persistent columns can be configured to display grid cells as a combo box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (…) in a cell that users can click to launch special data viewers or dialogs related to the current cell.

Creating persistent columns

To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the *State* property for a grid with persistent column objects is automatically set to *csCustomized*.

To create persistent columns for a grid control,

1 Select the grid component in the form.

2 Invoke the Columns editor by double clicking on the grid's *Columns* property in the Object Inspector.

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

You can create persistent columns for all fields in a dataset at once, or you can create persistent columns on an individual basis. To create persistent columns for all fields:

1 Right-click the grid to invoke the context menu and choose Add All Fields. Note that if the grid is not already associated with a data source, Add All Fields is disabled. Associate the grid with a data source that has an active dataset before choosing Add All Fields.

2 If the grid already contains persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set. If you choose Yes, any existing persistent column information is removed, and all fields in the current dataset are inserted by field name according to their order in the dataset. If you choose No, any existing persistent column information is retained, and new column information, based on any additional fields in the dataset, are appended to the dataset.

3 Click Close to apply the persistent columns to the grid and close the dialog box.

To create persistent columns individually:

1 Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).

2 To associate a field with this new column, set the *FieldName* property in the Object Inspector.

3 To set the title for the new column, expand the *Title* property in the Object Inspector and set its *Caption* property.

4 Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

At runtime, you can switch to persistent columns by assigning *csCustomized* to the *Columns.State* (Delphi) or *Columns::State* (C++) property. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset. You can then add a persistent column at runtime by calling the *Add* method for the column list:

```
DBGrid1.Columns.Add;
```

```
DBGrid1->Columns->Add();
```

Deleting persistent columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display. To remove a persistent column from a grid,

1. Double-click the grid to display the Columns editor.
2. Select the field to remove in the Columns list box.
3. Click Delete (you can also use the context menu or **Del** key, to remove a column).

Note If you delete all the columns from a grid, the *Columns.State* (Delphi) or *Columns->State* (C++) property reverts to its *csDefault* state and automatically build dynamic columns for each field in the dataset.

You can delete a persistent column at runtime by simply freeing the column object:

```
DBGrid1.Columns[5].Free;
```

```
delete DBGrid1->Columns->Items[5];
```

Arranging the order of persistent columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column,

1. Select the column in the Columns list box.
2. Drag it to a new location in the list box.

You can also change the column order at runtime by clicking on the column title and dragging the column to a new position.

Note Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

Important You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

At runtime, a user can use the mouse to drag a column to a new location in the grid if its *DragMode* property is set to *dmManual*. Reordering the columns of a grid with a *State* property of *csDefault* state also reorders field components in the dataset underlying the grid. The order of fields in the physical table is not affected. To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

At runtime, the grid's *OnColumnMoved* event is fired after a column has been moved.

Setting column properties at design time

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component, called the *default source*, such as a grid or an associated field component.

To set a column's properties, select the column in the Columns editor and set its properties in the Object Inspector. The following table summarizes key column properties you can set.

Table 20.2 Column properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: *TField Alignment* property.
ButtonStyle	*cbsAuto*: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's *PickList* property contains data. *cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event. *cbsNone*: The column uses only the normal edit control to edit data in the column.
Color	Specifies the background color of the cells of the column. Default Source: *TDBGrid Color* property. (For text foreground color, see the Font property.)
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7.
Expanded	Specifies whether the column is expanded. Only applies to columns representing composite fields.
FieldName	Specifies the field name associated with this column. This can be blank.
ReadOnly	`true`*:* The data in the column cannot be edited by the user. `false`*:* (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default Source: derived from *TField DisplayWidth* property.
Font	Specifies the font type, size, and color used to draw text in the column. Default Source: *TDBGrid Font* property.
PickList	Contains a list of values to display in a drop-down list in the column.
Title	Sets properties for the title of the selected column.

The following table summarizes the options you can specify for the *Title* property.

Table 20.3 Expanded TColumn Title properties

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default Source: *TField DisplayLabel* property.
Color	Specifies the background color used to draw the column title cell. Default Source: *TDBGrid FixedColor* property.
Font	Specifies the font type, size, and color used to draw text in the column title. Default Source: *TDBGrid TitleFont* property.

Defining a lookup list column

You can create a column that displays a drop-down list of values, similar to a lookup combo box control. To specify that the column acts like a combo box, set the column's *ButtonStyle* property to *cbsAuto*. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.

There are two ways to populate that list with the values for users to select:

- You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field in the dataset. For information about creating lookup fields, see "Defining a lookup field" on page 23-8. Once the lookup field is defined, set the column's *FieldName* to the lookup field name. The drop-down list is automatically populated with lookup values defined by the lookup field.
- You can specify a list of values explicitly at design time. To enter the list values at design time, double-click the *PickList* property for the column in the Object Inspector. This brings up the String List editor, where you can enter the values that populate the pick list for the column.

By default, the drop-down list displays 7 values. You can change the length of this list by setting the *DropDownRows* property.

Note To restore a column with an explicit pick list to its normal behavior, delete all the text from the pick list using the String List editor.

Putting a button in a column

A column can display an ellipsis button (…) to the right of the normal cell editor. *Ctrl+Enter* or a mouse click fires the grid's *OnEditButtonClick* event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form that displays an image.

To create an ellipsis button in a column:

1. Select the column in the *Columns* list box.
2. Set *ButtonStyle* to *cbsEllipsis*.
3. Write an *OnEditButtonClick* event handler.

Restoring default values to a column

At runtime, you can test a column's *AssignedValues* property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults.

You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component

At runtime, you can reset all default properties for a single column by calling the column's *RestoreDefaults* method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
DBGrid1.Columns.RestoreDefaults;
```

```
DBGrid1->Columns->RestoreDefaults();
```

Displaying composite fields

Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields.

There are two ways a grid can display composite fields:

- It can "flatten out" a composite field type so that each of the simpler types that make up the field appears as a separate field in the dataset. When a composite field is flattened out, its constituents appear as separate fields that reflect their common source only in that each field name is preceded by the name of the common parent field in the underlying database table.

  To display composite fields as if they were flattened out, set the dataset's *ObjectView* property to false. The dataset stores composite fields as a set of separate fields, and the grid reflects this by assigning each constituent part a separate column.
- It can display composite fields in a single column, reflecting the fact that they are a single field. When displaying composite fields in a single column, the column can be expanded and collapsed by clicking on the arrow in the title bar of the field, or by setting the *Expanded* property of the column:
  - When a column is expanded, each child field appears in its own sub-column with a title bar that appears below the title bar of the parent field. That is, the title bar for the grid increases in height, with the first row giving the name of the composite field, and the second row subdividing that for the individual parts. Fields that are not composites appear with title bars that are extra high. This expansion continues for constituents that are in turn composite fields (for example, a detail table nested in a detail table), with the title bar growing in height accordingly.
  - When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

  To display a composite field in an expanding and collapsing column, set the dataset's *ObjectView* property to true. The dataset stores the composite field as a single field component that contains a set of nested sub-fields. The grid reflects this in a column that can expand or collapse.

Table 20.4 Properties that affect the way composite fields appear

Property	Object	Purpose
Expandable	TColumn	Indicates whether the column can be expanded to show child fields in separate, editable columns. (read-only)
Expanded	TColumn	Specifies whether the column is expanded.
MaxTitleRows	TDBGrid	Specifies the maximum number of title rows that can appear in the grid
ObjectView	TDataSet	Specifies whether fields are displayed flattened out, or in object mode, where each object field can be expanded and collapsed.
ParentColumn	TColumn	Refers to the TColumn object that owns the child field's column.

Note In addition to Composite fields, some datasets include fields that refer to another dataset (dataset fields) or a record in another dataset (reference) fields. Data-aware grids display such fields as "(DataSet)" or "(Reference)", respectively. At runtime an ellipsis button appears to the right. Clicking on the ellipsis brings up a new form with a grid displaying the contents of the field. For dataset fields, this grid displays the dataset that is the field's value. For reference fields, this grid contains a single row that displays the record from another dataset.

Setting grid options

You can use the grid *Options* property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the Object Inspector is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, that collapses the list back when you click it.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

Table 20.5 Expanded TDBGrid Options properties

Option	Purpose
dgEditing	true: (Default). Enables editing, inserting, and deleting records in the grid.
	false: Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	true*:* When a field is selected, it is in Edit state.
	false*:* (Default). A field isn't automatically in Edit state when selected.
dgTitles	true*:* (Default). Displays field names across the top of the grid.
	false*:* Field name display is turned off.

Table 20.5 Expanded TDBGrid Options properties (continued)

Option	Purpose
dgIndicator	true*:* (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam.
	false: The indicator column is turned off.
dgColumnResize	true*:* (Default). Columns can be resized by dragging the column rulers in the title area. Resizing can change the corresponding width of the underlying *TField* component.
	false*:* Columns cannot be resized in the grid.
dgColLines	true*:* (Default). Displays vertical dividing lines between columns.
	false*:* Does not display dividing lines between columns.
dgRowLines	true*:* (Default). Displays horizontal dividing lines between records.
	false*:* Does not display dividing lines between records.
dgTabs	true*:* (Default). Enables tabbing between fields in records.
	false*:* Tabbing exits the grid control.
dgRowSelect	true*:* The selection bar spans the entire width of the grid.
	false*:* (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	true*:* (Default). The selection bar in the grid is always visible, even if another control has focus.
	false*:* The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	true*:* (Default). Prompt for confirmation to delete records (*Ctrl+Del*).
	false*:* Delete records without confirmation.
dgCancelOnExit	true*:* (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records.
	false*:* Permits pending inserts.
dgMultiSelect	true*:* Allows user to select noncontiguous rows in the grid using *Ctrl+Shift* or *Shift+ arrow* keys.
	false*:* (Default). Does not allow user to multi-select rows.

Editing in the grid

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The *CanModify property* of the *Dataset* is true.
- The *ReadOnly* property of grid is false.

When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus changes to another control on a form, the grid does not post changes until the cursor for the dataset moves to another record. When a record is posted, the dataset checks all associated data-aware components for a change in status. If there is a problem updating any fields that contain modified data, it raises an exception, and does not modify the record.

Note Posting record changes to a client dataset only adds them to the client dataset's internal change log. They are not posted back to the underlying database table until the client dataset applies its updates.

You can cancel all edits for a record by pressing **Esc** in any field before moving to another record.

Controlling grid drawing

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or *EditFormat* properties of the field component associated with the column.

You can augment the default grid display logic with code in a grid's *OnDrawColumnCell* event. If the grid's *DefaultDrawing* property is true, all the normal drawing is performed before your *OnDrawColumnCell* event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special values in that column's cells.

If you want to replace the drawing logic of the grid entirely, set *DefaultDrawing* to false and place your drawing code in the grid's *OnDrawColumnCell* event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the *DefaultDrawColumnCell* method inside your *OnDrawColumnCell* event handler to have the grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

Responding to user actions at runtime

You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the Object Inspector.

Table 20.6 Grid control events

Event	Purpose
OnCellClick	Occurs when a user clicks on a cell in the grid.
OnColEnter	Occurs when a user moves into a column on the grid.
OnColExit	Occurs when a user leaves a column on the grid.
OnColumnMoved	Occurs when the user moves a column to a new location.

Table 20.6 Grid control events (continued)

Event	Purpose
OnDblClick	Occurs when a user double clicks in the grid.
OnDragDrop	Occurs when a user drags and drops in the grid.
OnDragOver	Occurs when a user drags over the grid.
OnDrawColumnCell	Occurs when application needs to draw individual cells.
OnDrawDataCell	(obsolete) Occurs when application needs to draw individual cells if *State* is *csDefault*.
OnEditButtonClick	Occurs when the user clicks on an ellipsis button in a column.
OnEndDrag	Occurs when a user stops dragging on the grid.
OnEnter	Occurs when the grid gets focus.
OnExit	Occurs when the grid loses focus.
OnKeyDown	Occurs when a user presses any key or key combination on the keyboard when the grid has focus.
OnKeyPress	Occurs when a user presses a single alphanumeric key on the keyboard when the grid has focus.
OnKeyUp	Occurs when a user releases a key when in the grid.
OnMouseDown	Occurs when the user clicks the mouse in the grid.
OnMouseMove	Occurs when the user moves the mouse over the grid.
OnMouseUp	Occurs when the user releases the mouse button over the grid.
OnStartDrag	Occurs when a user starts dragging on the grid.
OnTitleClick	Occurs when a user clicks the title for a column.

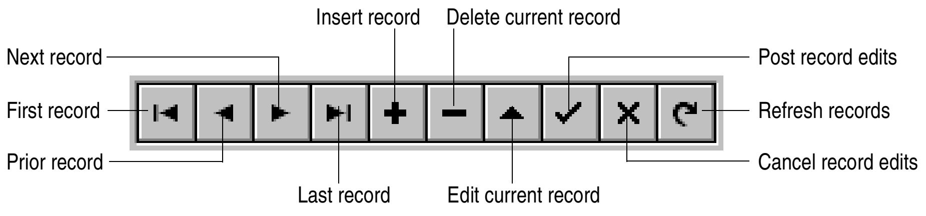
There are many uses for these events. For example, you might write a handler for the *OnDblClick* event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the *SelectedField* property to determine to current row and column.

Navigating and manipulating records

TDBNavigator provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

Figure 20.2 shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator lets you hide or show a subset of these buttons dynamically.

Figure 20.2 Buttons on the TDBNavigator control



The following table describes the buttons on the navigator.

Table 20.7 TDBNavigator buttons

Button	Purpose
First	Calls the dataset's *First* method to set the current record to the first record.
Prior	Calls the dataset's *Prior* method to set the current record to the previous record.
Next	Calls the dataset's *Next* method to set the current record to the next record.
Last	Calls the dataset's *Last* method to set the current record to the last record.
Insert	Calls the dataset's *Insert* method to insert a new record before the current record, and set the dataset in Insert state.
Delete	Deletes the current record. If the *ConfirmDelete* property is `true` it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, and returns the dataset to Browse state.
Refresh	Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

Choosing navigator buttons to display

When you first place a *TDBNavigator* on a form at design time, all its buttons are visible. You can use the *VisibleButtons* property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the First, Next, and Refresh buttons are meaningful, and you probably want to hide the others. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

Hiding and showing navigator buttons at design time

The *VisibleButtons* property in the Object Inspector is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, click on the + sign. The list of buttons that can be turned on or off appears in the Object Inspector below the *VisibleButtons* property. The + sign changes to a – (minus) sign, which you can click to collapse the list of properties.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to true, the button appears in the *TDBNavigator*. If false, the button is removed from the navigator at design time and runtime.

Note As button values are set to false, they are removed from the *TDBNavigator* on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert, Delete, Edit, Post, Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, suppose you want to prevent edits to the *OrdersTable* by hiding the *Insert, Delete, Edit, Post, Cancel,* and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The *VisibleButtons* property controls which buttons are displayed in the navigator. Here's one way you might code the *OnEnter* event handler:

Delphi example

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
  begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
  end
  else
  begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
      nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
  end;
end;
```

C++ example

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
  {
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
    DBNavigatorAll->VisibleButtons = TButtonSet() << nbFirst << nbPrior << nbNext << nbLast;
  }
  else
  {
    DBNavigatorAll->DataSource = OrderNum->DataSource;
    DBNavigatorAll->VisibleButtons = TButtonSet() << nbInsert << nbDelete << nbEdit
```

```
                                      << nbPost << nbCancel << nbRefresh;
  }
}
```

Displaying fly-over help

To display fly-over help for each navigator button at runtime, set the navigator *ShowHint* property to true. When *ShowHint* is true, the navigator displays fly-by Help Hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is false by default.

The *Hints* property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

Using a single navigator for multiple datasets

As with other data-aware controls, a navigator's *DataSource* property specifies the data source that links the control to a dataset. By changing a navigator's *DataSource* property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.

Suppose a form contains two edit controls linked to the *CustomersTable* and *OrdersTable* datasets through the *CustomersSource* and *OrdersSource* data sources respectively. When a user enters the edit control connected to *CustomersSource*, the navigator should also use *CustomersSource*, and when the user enters the edit control connected to *OrdersSource*, the navigator should switch to *OrdersSource* as well. You can code an *OnEnter* event handler for one of the edit controls, and then share that event with the other edit control. For example:

D

Delphi example

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```

C++ example

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
  else
    DBNavigatorAll->DataSource = OrderNum->DataSource;
}
```

Chapter

21

Connecting to databases

IDE applications use *dbExpress* to connect to a database. *dbExpress* is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces (Delphi) or classes (C++). When you deploy your application, you need only include a single shared object (the server-specific driver) with the application files you build.

Each *dbExpress* connection is encapsulated by a *TSQLConnection* component. *TSQLConnection* provides all the information necessary to establish a database connection using *dbExpress*, and is designed to work with unidirectional dataset components. A single SQL connection component can be shared by multiple unidirectional datasets, or the datasets can each use their own connection.

You may want to use a separate connection component for each dataset if the server does not support multiple statements per connection. Check whether the database server requires a separate connection for each dataset by reading the *MaxStmtsPerConn* property. By default, *TSQLConnection* generates connections as needed when the server limits the number of statements that can be executed over a connection. If you want to keep stricter track of the connections you are using, set the *AutoClone* property to false.

TSQLConnection lets you perform a number of tasks concerned with how you connect to or use the database server. These include

- Controlling connections
- Controlling server login
- Managing transactions
- Accessing server metadata
- Working with associated datasets
- Sending commands to the server
- Debugging database applications

Controlling connections

Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server connection. Once you have identified the server and provided login details, *TSQLConnection* can open or close a connection to the server.

Describing the server connection

In order to describe a database connection in sufficient detail for *TSQLConnection* to open a connection, you must identify both the driver to use and a set of connection parameters that are passed to that driver.

Identifying the driver

The driver is identified by the *DriverName* property, which is the name of an installed *dbExpress* driver, such as DB2, Informix, Interbase, MySQL, Oracle, or PostgreSQL. The driver name is associated with two files

- The *dbExpress* driver, which is a shared object file with a name like libsqlib.so, libsqlmys.so, libsqlora.so, libsqlinf.so, or libsqldb2.so.
- The Shared object file provided by the database vendor.

The relationship between these two shared object files and the database name is stored in a file called "dbxdrivers," which is updated when you install a *dbExpress* driver. Typically, you do not need to worry about these files because the SQL connection component looks them up in dbxdrivers when given the value of *DriverName*. When you set the *DriverName* property, *TSQLConnection* automatically sets the *LibraryName* and *VendorLib* properties to the names of the associated shared objects. Once *LibraryName* and *VendorLib* have been set (along with *GetDriverFunc*), your application does not need to rely on dbxdrivers. (That is, you do not need to deploy the dbxdrivers file with your application.)

Specifying connection parameters

The *Params* property is a string list that lists name/value pairs. Each pair has the form *Name=Value*, where *Name* is the name of the parameter, and *Value* is the value you want to assign.

The particular parameters you need depend on the database server you are using. However, one particular parameter, *Database*, is required for all servers. Its value depends on the server you are using. For example, with InterBase, *Database* is the name of the .gdb file, with Oracle, it is the entry in TNSName.ora, with DB2, it is the client-side node, while with Informix, MySQL, and PostgreSQL it is the database name.

Other typical parameters include the *User_Name* (the name to use when logging in), *Password* (the password for *User_Name*), *HostName* (the machine name or IP address of where the server is located), and *TransIsolation* (the degree to which transactions you introduce are, by default, aware of changes made by other transactions). When you specify a driver name, the *Params* property is preloaded with all the parameters you need for that driver type, initialized to default values.

Because *Params* is a string list, at design time you can double-click on the *Params* property in the Object Inspector to edit the parameters using the String List editor. At runtime, use the *ParamsValues* property to assign values to individual parameters.

Naming a connection description

Although you can always specify a connection using only the *DatabaseName* and *Params* properties, it can be more convenient to name a specific combination and then just identify the connection by name. *dbExpress* allows you to name database and parameter combinations, which are then saved in a file called "dbxconnections". The name of each combination is called a connection name.

Once you have defined the connection name, you can identify a database connection by simply setting the *ConnectionName* property to a valid connection name. Setting *ConnectionName* automatically sets the *DriverName* and *Params* properties. Once *ConnectionName* is set, you can edit the *Params* property to create temporary differences from the saved set of parameter values, but changing the *DriverName* property clears both *Params* and *ConnectionName*.

One advantage of using connection names arises when you develop your application using one database (for example Local InterBase), but deploy it for use with another (such as ORACLE). In that case, *DriverName* and *Params* will likely differ on the system where you deploy your application from the values you use during development. You can switch between the two connection descriptions easily by using two versions of the dbxconnections file. At design-time, your application loads the *DriverName* and *Params* from the design-time version of dbxconnections. Then, when you deploy your application, it loads these values from a separate version of dbxconnections that uses the "real" database. However, for this to work, you must instruct your connection component to reload the *DriverName* and *Params* properties at runtime. There are two ways to do this:

- Set the *LoadParamsOnConnect* property to true. This causes *TSQLConnection* to automatically set *DriverName* and *Params* to the values associated with *ConnectionName* in the dbxconnections file when the connection is opened.
- Call the *LoadParamsFromIniFile* method. This method sets *DriverName* and *Params* to the values associated with *ConnectionName* in the dbxconnections file. You might choose to use this method if you want to then override certain parameter values before opening the connection.

Using the Connection Editor

The relationships between connection names and their associated driver and connection parameters is stored in the dbxconnections file. You can create or modify these associations using the Connection Editor.

To display the Connection Editor, double-click on the *TSQLConnection* component. The Connection Editor appears, with a drop-down list containing all available drivers, a list of connection names for the currently selected driver, and a table listing the connection parameters for the currently selected connection name.

You can use this dialog to indicate the connection to use by selecting a driver and connection name, and editing any parameter values you want to temporarily override. Once you have chosen the configuration you want, click the Test Connection button to check that you have not made any mistakes.

You can use this dialog to indicate the connection to use by selecting a driver and connection name. Once you have chosen the configuration you want, click the Test Connection button to check that you have chosen a valid configuration.

In addition, you can use this dialog to edit the named connections in dbxconnections:

- Edit the parameter values in the parameter table to change the currently selected named connection. When you exit the dialog by clicking OK, the new parameter values are saved to the dbxconnections file.
- Click the Add Connection button to define a new named connection. A dialog appears where you specify the driver to use and the name of the new connection. Once the connection is named, edit the parameters to specify the connection you want and click the OK button to save the new connection to dbxconnections.
- Click the Delete Connection button to delete the currently selected named connection from dbxconnections.
- Click the Rename Connection button to change the name of the currently selected named connection. Note that any edits you have made to the parameters are saved with the new name when you click the OK button.

Opening and closing server connections

TSQLConnection generates events when it opens or closes a connection to the server. This lets you customize the behavior of your application in response to changes in the database connection.

Opening a connection

There are two ways to connect to a database server using *TSQLConnection*:

- Call the *Open* method.
- Set the *Connected* property to true.

Calling the *Open* method sets *Connected* to true.

When you set *Connected* to true, *TSQLConnection* first generates a *BeforeConnect* event, where you can perform any initialization. For example, you can use this event to alter any properties that specify the server to which it will then connect. If you are altering any parameter values, however, be sure that the *LoadParamsOnConnect* property is false, otherwise, after the event handler exits, *TSQLConnection* replaces all the connection parameters with the values associated with *ConnectionName* in the dbxconnections file.

After the *BeforeConnect* event, *TSQLConnection* may display a default login dialog, depending on how you choose to control server login. It then passes the user name and password to the driver, opening a connection.

Once the connection is open, *TSQLConnection* generates an *AfterConnect* event, where you can perform any tasks that require an open connection, such as fetching metadata from the server.

Once a connection is established, it is maintained as long as there is at least one active dataset using it. When there are no more active datasets, *TSQLConnection* may drop the connection, depending on the value of its *KeepConnection* property.

KeepConnection determines if your application maintains a connection to a database even when all datasets associated with that database are closed. If true, a connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, setting *KeepConnection* to true reduces network traffic and speeds up your application. If *KeepConnection* is false, the connection is dropped when there are no active datasets using the database. If a dataset that uses the database is later opened, the connection must be reestablished and initialized.

Disconnecting from a database server

There are two ways to disconnect from a server using *TSQLConnection*:

- Set the *Connected* property to false.
- Call the *Close* method.

Calling *Close* sets *Connected* to false.

When *Connected* is set to false, *TSQLConnection* generates a *BeforeDisconnect* event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed.

After the *BeforeConnect* event, *TSQLConnection* closes all open datasets and disconnects from the server.

Finally, *TSQLConnection* generates an *AfterDisconnect* event, where you can respond to the change in connection status, such as enabling a Connect button in your user interface.

Note Calling *Close* or setting *Connected* to false disconnects from a database server even if *KeepConnection* is true.

Controlling server login

Most remote database servers include security features to prohibit unauthorized access. Generally, the server requires a user name and password login before permitting database access.

At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are several ways you can handle a server's request for a login:

- Let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt* property of the *TSQLConnection* object to true (the default). In Delphi, add DBLogDlg to the **uses** clause of the unit that declares the connection component. In C++, include DBLogDlg.hpp in the unit that declares the connection component. Your application displays the standard login dialog box when the server requests a user name and password.

- Provide the values for User_Name and Password in the dbxconnections file. You can save these parameter values from the Connection Editor under a particular *ConnectionName*. Set *LoginPrompt* to false to prevent the default login dialog from appearing. Typically, you only want to use this approach to avoid the need to log in during development. It is a serious breach of security to leave an unencrypted password in the dbxconnections file, where it is available for anyone to read.
- Use the *Params* property to supply login information before the attempt to log in. You can assign values to the *User_Name* and *Password* parameters at design-time through the Object Inspector or programmatically at runtime. Set *LoginPrompt* to false to prevent the default login dialog from appearing. For example, the following code sets the user name and password in the *BeforeConnect* event handler, decrypting an encrypted password that is stored in the dbxconnections file:

Delphi example

```
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
  with Sender as TSQLConnection do
  begin
    if LoginPrompt = False then
    begin
      Params.Values['User_Name'] := 'SYSDBA';
      Params.Values['Password'] := Decrypt(Params.Values['Password']);
    end;
  end;
end;
```

C++ example

```
void __fastcall TForm1::SQLConnectionBeforeConnect(TObject *Sender)
{
  if (SQLConnection1->LoginPrompt == false)
  {
    SQLConnection1->Params->Values["User_Name"] = "SYSDBA";
    SQLConnection1->Params->Values["Password"] =
        Decrypt(SQLConnection1->Params->Values["Password"]);
  }
}
```

Note that setting the values in the *Params* property at design-time or using hard-coded strings in code causes the values to be embedded in the application's executable file. While not as freely available as in the dbxconnections file, this still leaves them easy to find, compromising server security.

- Write an event handler for the *OnLogin* event. Set the *LoginPrompt* property to true and in the *OnLogin* event handler, set the login parameters. A copy of the *User_Name*, *Password*, and *Database* parameters is passed to the event handler in its

LoginParams parameter. Assign values to these parameters using this string list, providing whichever values are needed. On exit, the values returned in *LoginParams* are used to form the connection. The following example assigns values for the *User_Name* and *Password* database parameters using a global variable (*UserName*) and a method that returns a password given a user name (*PasswordSearch*):

Delphi example

```
procedure TForm1.SQLConnection1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['User_Name'] := UserName;
  LoginParams.Values['Password'] := PasswordSearch(UserName);
end;
```

C++ example

```
void __fastcall TForm1::SQLConnection1Login(TDatabase *Database, TStrings *LoginParams)
{
  LoginParams->Values["User_Name"] = UserName;
  LoginParams->Values["Password"] = PasswordSearch(UserName);
}
```

As with the other methods of providing login parameters, when writing an *OnLogin* event handler, avoid hard coding the password in your application code. It should appear only as an encrypted value, an entry in a secure database your application uses to look up the value, or be dynamically obtained from the user.

Warning The *OnLogin* event does not occur unless the *LoginPrompt* property is true. Having a *LoginPrompt* value of false and providing login information only in an *OnLogin* event handler creates a situation where it is impossible to log in to the database: The default dialog does not appear and the *OnLogin* event handler never executes.

Managing transactions

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

It is always possible to manage transactions by sending SQL commands directly to the database. Each database provides its own transaction management model, although some, such as MySQL, have no transaction support at all. For servers that support it, you may want to code your own transaction management directly, taking advantage of advanced transaction management capabilities on a particular database server, such as schema caching.

If you do not need to use any advanced transaction management capabilities, *TSQLConnection* provides a set of methods and properties you can use to manage transactions without explicitly sending any SQL commands. Using these properties and methods has the advantage that you do not need to customize your application for each type of database server you use, as long as the server supports transactions. (Trying to use transactions on a database that does not support them — such as MySQL— causes *TSQLConnection* to raise an exception.)

Warning When a client dataset edits the data accessed through a *TSQLConnection* component (either directly or when it is linked, through a provider, to a unidirectional dataset that uses the *TSQLConnection* component), the provider implicitly provides transaction support for any updates. Be careful that any transactions you explicitly start do not conflict with those generated by the provider.

Starting a transaction

Start a transaction by calling the *StartTransaction* method. *StartTransaction* takes a single parameter, a transaction descriptor that lets you manage multiple simultaneous transactions and specify the transaction isolation level on a per-transaction basis. (For more information on transaction levels, see "Specifying the transaction isolation level" on page 21-10.)

Delphi example

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
```

C++ example

```
TTransactionDesc TD;
TD.TransactionID = 1;
TD.IsolationLevel = xilREADCOMMITTED;
SQLConnection1->StartTransaction(TD);
```

In order to manage multiple simultaneous transactions, set the *TransactionID* field of the transaction descriptor to a unique value. *TransactionID* can be any value you choose, as long as it is unique (does not conflict with any other transaction currently underway). Depending on the server, transactions started by *TSQLConnection* can be nested or they can be overlapped. Before you create multiple simultaneous transactions, be sure they are supported by the database server.

By default, when you start a transaction, all subsequent statements that read from or write to the database occur in the context of that transaction, until the transaction is explicitly terminated or, in the case of overlapped transactions, until another transaction starts. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

With overlapped transactions, a first transaction becomes inactive when the second transaction starts, although you can postpone committing or rolling back the first transaction until later. However, if you are using an InterBase database, you can identify each dataset in your application with a particular active transaction, by setting its *TransactionLevel* property. That is, after starting a second transaction, you can continue to work with both transactions simultaneously, simply by associating a dataset with the transaction you want.

You can determine whether a transaction is in process by checking the *InTransaction* property.

Ending a transaction

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit any changes.

Ending a successful transaction

When the actions that make up the transaction have all succeeded, you can make the database changes permanent by using the *Commit* method. *Commit* takes a single parameter: the transaction descriptor you supplied to the *StartTransaction* method:

```
SQLConnection1.Commit(TD);
```

```
SQLConnection1->Commit(TD);
```

Note If you are working with nested transactions, it is possible to commit a secondary (nested) transaction, only to have it rolled back later when the primary transaction is rolled back.

Commit is usually attempted in a try...except (Delphi) or catch (C++) statement. That way, if the transaction cannot commit successfully, you can use the except or catch block to handle the error and retry the operation or to roll back the transaction.

Ending an unsuccessful transaction

If an error occurs when making the changes that are part of the transaction or when trying to commit the transaction, you will want to discard all changes that make up the transaction. To discard these changes, use the *Rollback* method. *Rollback* takes a single parameter: the transaction descriptor you supplied to the *StartTransaction* method:

```
SQLConnection1.Rollback(TD);
```

```
SQLConnection1->Rollback(TD);
```

Rollback usually occurs in

- Exception handling code when you can't recover from a database error.

- Button or menu event code, such as when a user clicks a Cancel button.

Specifying the transaction isolation level

The transaction descriptor you provide when you start a transaction includes an *IsolationLevel* field, which allows you to control how the transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction “sees” of other transactions’ changes to a table.

Each server type supports a different set of possible values for *IsolationLevel*, and some do not support different isolation levels at all. If the server supports different isolation levels, the set of allowable values includes two or more of the following, depending on the server:

- *DirtyRead*: When *TransIsolation* is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is not available for many database servers (such as Oracle or InterBase).
- *ReadCommitted*: When *TransIsolation* is *ReadCommitted*, only committed changes made by other transactions are visible. Although this setting protects your transaction from seeing uncommitted changes that may be rolled back, you may still receive an inconsistent view of the database state if another transaction is committed while you are in the process of reading.
- *RepeatableRead*: When *TransIsolation* is *RepeatableRead*, your transaction is guaranteed to see a consistent state of the database data. Your transaction sees a single snapshot of the data. It cannot see any subsequent changes to data by other simultaneous transactions, even if they are committed. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions.

Note In addition, the transaction descriptor allows you to specify a custom isolation level that is defined by the database driver. None of the dbExpress drivers provided with the IDE support custom isolation levels.

For a detailed description of how each isolation level is implemented, see your server documentation.

Accessing server metadata

Once you have an open connection, you can use *TSQLConnection* to obtain information about the entities available on the database server. This information, called metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

TSQLConnection has a number of methods you can call to fill a list with this metadata:

- *GetTableNames* fills a *TStrings* descendant (such a *TStringList*) with the names of all available tables on the server. Which tables are returned depends on two things: the *TableScope* property, and a boolean parameter that indicates whether you are only interested in system tables. If you only want system tables, you can use the *SystemTables* parameter, and the list will contain only system tables, regardless of the value of *TableScope*. (Note that not all servers use system tables to store metadata, so asking for system tables may result in an empty list.) If you do not request only system tables, *GetTableNames* returns the names of any tables that match the types specified by the *TableScope* property. These can include system tables, data tables, views, and synonyms.
- *GetFieldNames* fills a *TStrings* descendant with the names of all fields (columns) in a specified table.
- *GetIndexNames* fills a *TStrings* descendant with the names of all indexes defined on a specified table.
- *GetProcedureNames* fills a *TStrings* descendant with the names of all available stored procedures (if any).
- *GetProcedureParams* fills a *TList* object with pointers to parameter description records (Delphi) or structures (C++), where each record or structure describes a parameter of a specified stored procedure, including its name, index, parameter type, field type, and so on. You can convert these parameter descriptions to the more familiar *TParams* object by calling the global *LoadParamListItems* procedure. Because *GetProcedureParams* dynamically allocates the individual records (Delphi) or structures (C++), your application must free them when it is finished with the information.
- *GetPackageNames* fills a *TStrings* descendant with the names of all packages (if any) defined on the server. This method is only relevant to Oracle servers.

Note You can also access server metadata using *TSQLDataSet*. *TSQLDataSet* provides more detailed information and greater control over the information returned (for example, you can provide a pattern mask to limit the items returned. For more information on how to access server metadata with *TSQLDataSet*, see "Accessing schema information" on page 24-17.

Working with associated datasets

TSQLConnection maintains a list of all active datasets that use it to connect to a database. It uses this list, for example, to close all of the datasets when it closes the database connection.

You can use this list as well, to perform actions on all the datasets that use a specific *TSQLConnection* to connect to a particular database.

Closing datasets without disconnecting from the server

There may be times when you want to close all datasets without disconnecting from the database server. To close all open datasets without disconnecting from a server, follow these steps:

1 Set the *TSQLConnection* component's *KeepConnection* property to true.

2 Call the *TSQLConnection* component's *CloseDataSets* method.

Iterating through the associated datasets

To perform any actions (other than closing them all) on all the datasets that use a *TSQLConnection* instance, use the *DataSets* and *DataSetCount* properties. *DataSets* is an indexed array of all active datasets that are linked to the SQL connection component via their *SQLConnection* property. *DataSetCount* is the number of datasets in this array.

Note When using an SQL client dataset (*TSQLClientDataSet*), the connection component's *DataSets* property does not list the client dataset itself. Rather, it lists the internal unidirectional dataset that the client dataset uses to access the data.

DataSets lists only the active (open) datasets. If a dataset is closed, it does not appear in the list.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets and disables any controls that use the data they provide:

Delphi example

```
var
  I: Integer;
begin
  with SQLConnection1 do
  begin
    for I := 0 to DataSetCount - 1 do
      DataSets[I].DisableControls;
  end;
end;
```

C++ example

```
for (int i = 0; i < SQLConnection1->DataSetCount; i++)
  SQLConnection1->DataSets[i]->DisableControls();
```

Sending commands to the server

Simple Data Definition Language (DDL) SQL statements such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN statements can be executed directly from a *TSQLConnection* component using its *Execute* method. These statements do not return result sets and only operate on or create a database's metadata. The *Execute* method can also be used to execute Data Manipulation Language (DML) SQL statements, such as INSERT, DELETE, and UPDATE statements.

Execute takes three parameters, a string (Delphi) or an AnsiString (C++) that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, and a pointer that can receive a dynamically created *TCustomSQLDataSet* object that is populated with the result set from executing the statement. Although you can use this third parameter to receive records that are returned as a result of executing the statement, the recommended approach is to use an SQL dataset instead. If you do use the third parameter, your application is responsible for freeing the dataset returned in the third parameter.

Note *Execute* can only execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, as you can with SQL scripting utilities. To execute more than one statement, call *Execute* repeatedly, each time with new parameters.

It is relatively easy to execute a statement that does not include any parameters. For example, the following code executes a CREATE TABLE statement (DDL) without any parameters:

Delphi example

```
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( ' +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
```

C++ example

```
void __fastcall TDataForm::CreateTableButtonClick(TObject *Sender)
{
  SQLConnection1->Connected = true;
  AnsiString SQLstmt = "CREATE TABLE NewCusts " +
    "( " +
    " CustNo INTEGER, " +
    " Company CHAR(40), " +
    " State CHAR(2), " +
    " PRIMARY KEY (CustNo) " +
    ")";
  SQLConnection1->Execute(SQLstmt, NULL, NULL);
}
```

To use parameters, you must create a *TParams* object. For each parameter value, use the *TParams CreateParam* method to add a *TParam* object. Then use properties of *TParam* to describe the parameter and set its value.

This process is illustrated in the following example, which executes an INSERT statement. The INSERT statement includes a single parameter named :*StateParam*. A *TParams* object (called *stmtParams*) is created to supply a value of "CA" for that parameter.

Delphi example

```
procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    SQLConnection1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom.db" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    SQLConnection1.Execute(SQLstmt, stmtParams, nil);
  finally
    stmtParams.Free;
  end;
end;
```

C++ example

```
void __fastcall TForm1::INSERT_WithParamsButtonClick(TObject *Sender)
{
  AnsiString SQLstmt;
  TParams *stmtParams = new TParams;
  try
  {
    SQLConnection1->Connected = true;
    stmtParams->CreateParam(ftString, "StateParam", ptInput);
    stmtParams->ParamByName("StateParam")->AsString = "CA";
    SQLstmt = "INSERT INTO 'Custom.db' ";
    SQLstmt += "(CustNo, Company, State) ";
    SQLstmt += "VALUES (7777, 'Robin Dabank Consulting', :StateParam)";
    SQLConnection1->Execute(SQLstmt, stmtParams, NULL);
  }
  __finally
```

```
    {
      delete stmtParams;
    }
  }
```

If the SQL statement includes a parameter but you do not supply a *TParam* object to supply a value for it, the SQL statement may cause an error when executed (this depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

Debugging database applications

While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).

Using TSQLMonitor to monitor SQL commands

TSQLConnection uses a companion component, *TSQLMonitor*, to intercept these messages and save them in a string list. To use *TSQLMonitor*,

1 Add a *TSQLMonitor* component to the form or data module containing the *TSQLConnection* component whose SQL commands you want to monitor.
2 Set its *SQLConnection* property to the *TSQLConnection* component.
3 Set the SQL monitor's *Active* property to true.

As SQL commands are sent to the server, the SQL monitor's *TraceList* property is automatically updated to list all the SQL commands that are intercepted.

You can save this list to a file by specifying a value for the *FileName* property and then setting the *AutoSave* property to true. *AutoSave* causes the SQL monitor to save the contents of the *TraceList* property to a file every time is logs a new message.

If you do not want the overhead of saving a file every time a message is logged, you can use the *OnLogTrace* event handler to only save files after a number of messages have been logged. For example, the following event handler saves the contents of *TraceList* every 10th message, clearing the log after saving it so that the list never gets too long:

D **Delphi example**

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: Pointer);
var
  LogFileName: string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
```

```
      begin
        LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
        Tag := Tag + 1; {ensure next log file has a different name }
        SaveToFile(LogFileName);
        TraceList.Clear; { clear list }
      end;
    end;
  end;
```

C++ example

```
void __fastcall TForm1::SQLMonitor1LogTrace(TObject *Sender, void *CBInfo)
{
  TSQLMonitor *pMonitor = dynamic_cast<TSQLMonitor *>(Sender);
  if (pMonitor->TraceCount == 10)
  {
    // build unique file name
    AnsiString LogFileName = "c:\\log";
    LogFileName = LogFileName + IntToStr(pMonitor->Tag);
    LogFileName = LogFileName + ".txt"
    pMonitor->Tag = pMonitor->Tag + 1;
    // Save contents of log and clear the list
    pMonitor->SaveToFile(LogFileName);
    pMonitor->TraceList->Clear();

}
```

Note If you were to use the previous event handler, you would also want to save any partial list (fewer than 10 entries) when the application shuts down.

Using a callback to monitor SQL commands

Instead of using *TSQLMonitor*, you can customize the way your application traces SQL commands by using the SQL connection component's *SetTraceCallbackEvent* method. *SetTraceCallbackEvent* takes two parameters: a callback of type *TSQLCallbackEvent*, and a user-defined value that is passed to the callback function.

The callback function takes two parameters: *CallType* and *CBInfo*:

- *CallType* is reserved for future use.
- *CBInfo* is a pointer to a record (Delphi) or structure (C++) that includes the category (the same as *CallType*), the text of the SQL command, and the user-defined value that is passed to the *SetTraceCallbackEvent* method.

The callback returns a value of type *CBRType*, typically *cbrUSEDEF*.

The *dbExpress* driver calls your callback every time the SQL connection component passes a command to the server or the server returns an error message.

Warning Do not call *SetTraceCallbackEvent* if the *TSQLConnection* object has an associated *TSQLMonitor* component. *TSQLMonitor* uses the callback mechanism to work, and *TSQLConnection* can only support one callback at a time.

Chapter

22

Understanding datasets

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may come from a query or stored procedure that accesses a database, or from another dataset.

All dataset objects that you use in your database applications descend from the virtualized dataset object, *TDataSet*, and they inherit data fields, properties, events, and methods from *TDataSet*. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you will use in your database applications. You need to understand this shared functionality to use any dataset object.

TDataSet is a virtualized dataset, meaning that many of its properties and methods are virtual or abstract (in C++ terminology, pure virtual). A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An *abstract* or *pure virtual method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains abstract (pure virtual) methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of *TDataSet*'s descendants, such as *TClientDataSet*, *TSQLClientDataSet*, *TSQLDataSet*, *TSQLQuery*, *TSQLTable*, and *TSQLStoredProc*, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its abstract (pure virtual) methods.

TDataSet defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of *TField* components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For more information about *TField* components, see Chapter 23, "Working with field components."

This chapter describes how to use the common database functionality introduced by *TDataSet*. Bear in mind, however, that although *TDataSet* introduces the methods for this functionality, not all *TDataSet* dependants implement them in a meaningful way.

Types of datasets

TDataSet has two immediate descendants, *TCustomClientDataset* and *TCustomSQLDataSet*. In addition you can create your own custom *TDataSet* descendants — for example to supply data from a process other than a database server.

TCustomClientDataSet is the base class for all client datasets. It is not used directly in an application because many crucial properties are protected. Client datasets buffer data and updates in memory, and can be used when you need to navigate through a dataset or edit values and apply them back to the database server. Client datasets implement most of the features introduced by *TDataSet*, as well as introducing additional features such as maintained aggregates. For information about the features introduced by client datasets, see Chapter 25, “Using client datasets.”

TCustomSQLDataSet is the base class for all unidirectional datasets. This class can’t be used directly in an application because the properties that specify what data to access are all protected. Unidirectional datasets raise exceptions when you attempt any navigation other than moving to the next record. They do not provide support for any data buffering, or the functions that require data buffering (such as updating data, filters, and lookup fields). For information about the *TCustomSQLDataSet* descendants that you can use in your applications, see Chapter 24, “Using unidirectional datasets.”

Each of these *TDataset* descendants has advantages and disadvantages. For an overview of how these types of datasets can be built into your database application, see “Database architecture” on page 19-4.

Opening and closing datasets

To read or edit the data in a dataset, an application must first open it. You can open a dataset in two ways,

- Set the *Active* property of the dataset to true, either at design time in the Object Inspector, or in code at runtime:

```
CustTable.Active := True;
```

```
CustTable->Active = true;
```

- Call the *Open* method for the dataset at runtime,

```
CustQuery.Open;
```

```
CustQuery->Open();
```

You can close a dataset in two ways,

- Set the *Active* property of the dataset to false, either at design time in the Object Inspector, or in code at runtime,

D
```
CustQuery.Active := False;
```
C++
```
CustQuery->Active = false;
```

- Call the *Close* method for the dataset at runtime,

D
```
CustTable.Close;
```
C++
```
CustTable->Close();
```

You may need to close a dataset when you change certain of its properties, such as *CommandText* on a *TSQLDataSet* component. When you reopen the dataset, the new property value takes effect.

Determining and setting dataset states

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset's read-only *State* property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

Table 22.1 Values for the dataset State property

Value	State	Meaning
dsInactive	Inactive	DataSet closed. Its data is unavailable.
dsBrowse	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Edit	DataSet open. The current row can be modified. (not supported on unidirectional datasets)
dsInsert	Insert	DataSet open. A new row is inserted or appended. (not supported on unidirectional datasets)
dsSetKey	SetKey	*TClientDataSet* only. DataSet open. Enables setting of ranges and key values used for ranges and *GotoKey* operations.
dsCalcFields	CalcFields	DataSet open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated.
dsCurValue	CurValue	Internal use only.
dsNewValue	NewValue	Internal use only.
dsOldValue	OldValue	Internal use only.
dsFilter	Filter	DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed. (not supported on unidirectional datasets)
dsBlockRead	Block Read	DataSet open. Data-aware controls are not updated and events are not triggered when the current record changes.
dsInternalCalc	Internal Calc	DataSet open. An *OnCalcFields* event is underway for calculated values that are stored with the record. (client datasets only)

When an application opens a dataset, it appears by default in *dsBrowse* mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the

- code in your application, or
- built-in behavior of data-related components.

To put a dataset into *dsBrowse*, *dsEdit*, *dsInsert*, *dsSetKey,* or *dsBlockRead* states, call the method or set the property that corresponds to the name of the state. For example, the following code puts *CustTable* into *dsInsert* state, accepts user input for a new record, and writes the new record to the change log:

Delphi example

```
CustTable.Insert; { Your application explicitly sets dataset state to Insert }
AddressPromptDialog.ShowModal;
if (AddressPromptDialog.ModalResult = mrOK) then
  CustTable.Post { Kylix sets dataset state to Browse on successful completion }
else
  CustTable.Cancel; {Kylix sets dataset state to Browse on cancel }
```

C++ example

```
CustTable->Insert(); // explicitly set dataset state to Insert
AddressPromptDialog->ShowModal();
if (AddressPromptDialog->ModalResult == mrOk)
  CustTable->Post(); // dataset state changes to Browse on successful completion
else
  CustTable->Cancel(); // dataset state changes to Browse on cancel
```

This example also illustrates that the state of a dataset automatically changes to *dsBrowse* when

- The *Post* method successfully writes a record to the change log. (If *Post* fails, the dataset state remains unchanged.)
- The *Cancel* method is called.

Some states cannot be set directly. For example, to put a dataset into *dsInactive* state, set its *Active* property to false, or call the *Close* method for the dataset. The following statements are equivalent:

```
CustTable.Active := False;
```

```
CustTable.Close;
```

```
CustTable->Active = false;
```

```
CustTable->Close();
```

The remaining states (*dsCalcFields*, *dsCurValue*, *dsNewValue*, *dsOldValue*, *dsFilter*, and *dsInternalCalc*) cannot be set by your application. Instead, the state of the dataset changes automatically to these values as necessary. For example, *dsCalcFields* is set when a dataset's *OnCalcFields* event occurs. When the *OnCalcFields* event finishes, the dataset is restored to its previous state.

Note Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see "Responding to changes mediated by the data source" on page 20-4.

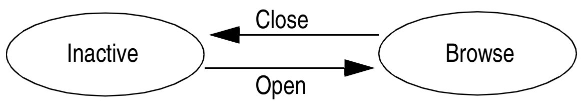
The following sections provide overviews of the most common states, how and when they are set, how states relate to one another, and where to go for related information, if applicable.

Inactivating a dataset

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time, a dataset is closed until you set its *Active* property to true. At runtime, a dataset is initially closed until an application opens it by calling the *Open* method, or by setting the *Active* property to true.

When you open an inactive dataset, its state automatically changes to the *dsBrowse* state. The following diagram illustrates the relationship between these states and the methods that set them.

Figure 22.1 Relationship of Inactive and Browse states



To make a dataset inactive, call its *Close* method. You can write *BeforeClose* and *AfterClose* event handlers that respond to the *Close* method for a dataset. For example, if a dataset is in *dsEdit* or *dsInsert* modes when an application calls *Close*, you can prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

Delphi example

```
procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation,
      mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { save the changes }
      mrNo:     CustTable.Cancel; { abandon the changes}
      mrCancel: Abort;            { abort closing the dataset }
    end;
  end;
end;
```

C++ example

```
void __fastcall TForm1::CustTableVerifyBeforeClose(TDataSet *DataSet)
{
  if (CustTable->State == dsEdit || CustTable->State == dsInsert)
  {
    TMsgDlgButtons btns;
    btns << mbYes << mbNo;
    if (MessageDlg("Post changes before closing?", mtConfirmation, btns, 0) == mrYes)
      CustTable->Post();
    else
      CustTable->Cancel();
  }
}
```

To associate a procedure with the *BeforeClose* event for a dataset at design time:

1 Select the table in the data module (or form).

2 Click the Events page in the Object Inspector.

3 Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

Browsing a dataset

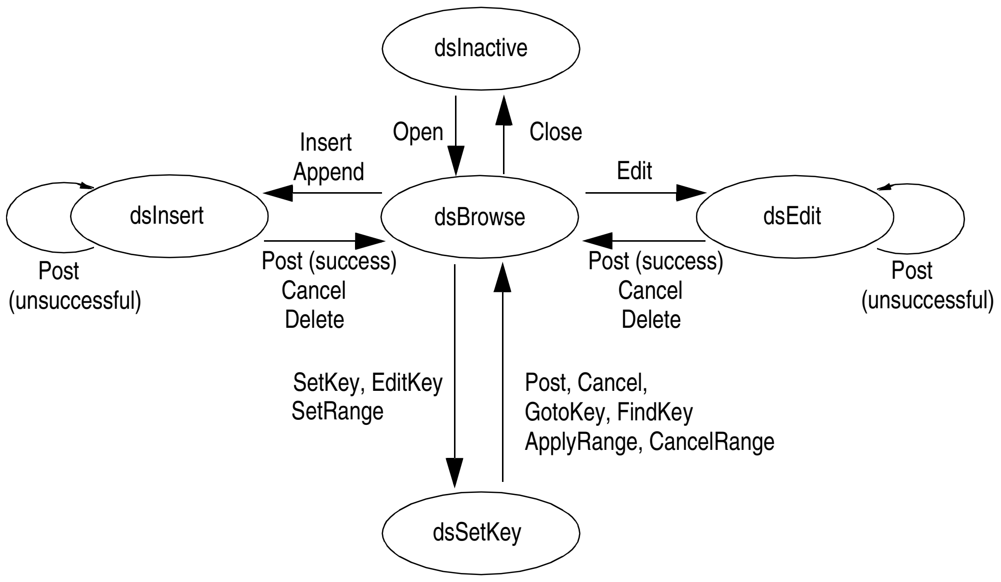
When an application opens a dataset, the dataset automatically enters *dsBrowse* state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use *dsBrowse* to scroll from record to record in a dataset. For more information about scrolling from record to record, see "Navigating datasets" on page 22-9.

From *dsBrowse* all other dataset states can be set, as long as the dataset supports them. For example, calling the *Insert* or *Append* methods for a dataset changes its state from *dsBrowse* to *dsInsert* (unless other factors and dataset properties, such as *CanModify* prevent this change). Calling *SetKey* to search for records puts a client dataset in *dsSetKey* mode. For more information about inserting and appending records in a dataset, see "Modifying data" on page 22-25.

Two methods associated with all datasets can return a dataset to *dsBrowse* state. *Cancel* ends the current edit, insert, or search task, and always returns a dataset to *dsBrowse* state. *Post* attempts to write changes to the database, and if successful, also returns a dataset to *dsBrowse* state. If *Post* fails, the current state remains unchanged.

The following diagram illustrates the relationship of *dsBrowse* both to the other dataset modes you can set in your applications, and the methods that set those modes.

Figure 22.2 Relationship of Browse to other dataset states



Enabling dataset editing

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is true. *CanModify* is true unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor (such as a client dataset's *ReadOnly* property or a dataset provider's *poReadOnly* option) intervenes.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if all of the following conditions apply:

- The control's *ReadOnly* property is false (the default)
- The *AutoEdit* property of the data source for the control is true
- *CanModify* is true for the dataset.

Note Even if a dataset is in *dsEdit* state, editing records may not succeed if your application user does not have proper SQL access privileges.

You can return a dataset from *dsEdit* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Cancel* discards edits to the current field or record. *Post* attempts to write a modified record to the change log, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write changes, the dataset remains in *dsEdit* state. *Delete* tries to remove the current record from the dataset, and if it succeeds, returns the dataset to *dsBrowse* state. If *Delete* fails, the dataset remains in *dsEdit* state.

Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid) or that causes the control to lose focus (such as moving to a different control on the form).

For a complete discussion of editing fields and records in a dataset, see "Modifying data" on page 22-25.

Enabling insertion of new records

A dataset must be in *dsInsert* mode before an application can add new records. In your code you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is true. *CanModify* is true unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor (such as a client dataset's *ReadOnly* property or a dataset provider's *poReadOnly* option) intervenes.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if all of the following conditions apply:

- The control's *ReadOnly* property is false (the default)
- The *AutoEdit* property of the data source for the control is true
- *CanModify* is true for the dataset.

Note Even if a dataset is in *dsInsert* state, inserting records may not succeed if your application user does not have proper SQL access privileges.

You can return a dataset from *dsInsert* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Delete* and *Cancel* discard the new record. *Post* attempts to write the new record to the change log, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write the record, the dataset remains in *dsInsert* state.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

For more discussion of inserting and appending records in a dataset, see "Modifying data" on page 22-25.

Enabling index-based operations

There are no methods defined in *TDataSet* that put the dataset into the *dsSetKey* state. This state exists for client datasets, which uses it when setting up information for index-based operations such as indexed-based searches or limiting records to a specified range. You can put a client dataset into *dsSetKey* mode with the *SetKey* or *EditKey* method at runtime.

While the dataset is in the *dsSetKey* state, assigning values to its fields changes an internal buffer containing the criteria for the indexed-based operation, rather than the values of the fields on the current record.

The dataset remains in the *dsSetKey* state until you call a method to perform the index-based operation, or call the *Post* method.

For more information about indexed-based operations in client datasets, see "Navigating data in client datasets" on page 25-2 and "Limiting what records appear" on page 25-6.

Calculating fields

A dataset enters *dsCalcFields* or *dsInternalCalc* mode whenever its *OnCalcFields* event occurs. These states prevent modifications or additions to the records in a dataset except for the calculated fields the handler is designed to modify. The reason all other modifications are prevented is because *OnCalcFields* uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the *OnCalcFields* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about creating calculated fields and *OnCalcFields* event handlers, see "Using OnCalcFields" on page 22-31.

Filtering records

If a dataset is not unidirectional, it enters *dsFilter* mode whenever an application calls the dataset's *OnFilterRecord* event handler. (Unidirectional datasets do not support filters). This state prevents modifications or additions to the records in a dataset during the filtering process so that the filter request is not invalidated.

When the *OnFilterRecord* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about filtering, see "Displaying and editing a subset of data using filters" on page 22-20.

Applying updates

When applying the updates in its change log back to the database table, client datasets may enter the *dsNewValue*, *dsOldValue*, or *dsCurValue* states temporarily. These states indicate that the corresponding properties of a field component (*NewValue*, *OldValue*, and *CurValue*, respectively) are being accessed, usually in an *OnReconcileError* event handler. Your applications cannot see or set these states.

Navigating datasets

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as *TDBEdit*, *TDBLabel*, and *TDBMemo*. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

Table 22.2 Navigational methods of datasets

Method	Moves the cursor to
First	The first row in a dataset. (all datasets)
Last	The last row in a dataset. (not available for unidirectional datasets).
Next	The next row in a dataset. (all datasets)
Prior	The previous row in a dataset. (not available for unidirectional datasets).
MoveBy	A specified number of rows forward or back in a dataset. (unidirectional datasets raise an exception if you try to move backward)

The data-aware, visual component *TDBNavigator* encapsulates these methods as buttons that users can click to move among records at runtime. For more information about the navigator component, see "Navigating and manipulating records" on page 20-26.

In addition to these methods, *TDataSet* defines two Boolean properties that provide useful information when iterating through the records in a dataset:.

Table 22.3 Navigational properties of datasets

Property	Description
Bof (Beginning-of-file)	true: the cursor is at the first row in the dataset.
	false: the cursor is not known to be at the first row in the dataset.
Eof (End-of-file)	true: the cursor is at the last row in the dataset.
	false: the cursor is not known to be at the last row in the dataset.

Using the First and Last methods

The *First* method moves the cursor to the first row in a dataset and sets the *Bof* property to true. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
CustTable.First;
```

```
CustTable->First();
```

The *Last* method moves the cursor to the last row in a dataset and sets the *Eof* property to true. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
CustTable.Last;
```

```
CustTable->Last();
```

Note The *Last* method raises an exception in unidirectional datasets.

Tip While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you can also enable your users to navigate from record to record using the *TDBNavigator* component. The navigator component contains buttons that, when active and visible, enable a user to move to the first and last rows of an active dataset. The *OnClick* events for these buttons occur immediately after the navigator calls the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see "Navigating and manipulating records" on page 20-26.

Using the Next and Prior methods

The *Next* method moves the cursor forward one row in the dataset and sets the *Bof* property to false if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
CustTable.Next;
```

```
CustTable->Next();
```

The *Prior* method moves the cursor back one row in the dataset, and sets *Eof* to false if the dataset is not empty. If the cursor is already at the first row in the dataset when you call *Prior*, *Prior* does nothing.

For example, the following code moves to the previous record in *CustTable*:

```
CustTable.Prior;
```

```
CustTable->Prior();
```

Note The *Prior* method raises an exception in unidirectional datasets.

Using the MoveBy method

MoveBy lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the *Bof* and *Eof* properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

Note *MoveBy* raises an exception in unidirectional datasets if you use a negative argument.

MoveBy returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
CustTable.MoveBy(-2);
```

```
CustTable->MoveBy(-2);
```

Using the Eof and Bof properties

Two read-only, runtime properties, *Eof* (End-of-file) and *Bof* (Beginning-of-file), are useful for controlling dataset navigation, particularly when you want to iterate through all records in a dataset.

Eof

When *Eof* is true, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to true when an application

- Opens an empty dataset.
- Successfully calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset.
- Calls *SetRange* on an empty range or dataset. (client datasets only)

Eof is set to false in all other cases; you should assume *Eof* is false unless one of the conditions above is met *and* you test the property directly.

Eof is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call *First*), *Eof* is false. To iterate through the dataset a record at a time, create a loop that terminates when *Eof* is true. Inside the loop, call *Next* for each record in the dataset. *Eof* remains false until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

Delphi example

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

C++ example

```
CustTable->DisableControls();
try
{
  for (CustTable->First();!CustTable->Eof;CustTable->Next()) // Cycle until Eof is true
  (
    // Process each record here
```

```
      ⋮
    }
  }
  __finally
  {
    CustTable->EnableControls();
  }
```

Tip This example also demonstrates how to disable and enable data-aware visual controls tied to a dataset. If you disable visual controls during dataset iteration, it speeds processing because your application does not need to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the **finally** clause of a **try...finally** statement (C++ **try...__finally**). This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

Bof

When *Bof* is true, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to true when an application

- Opens a dataset.
- Calls a dataset's *First* method.
- Calls a dataset's *Prior* method, and the method fails because the cursor is currently at the first row in the dataset. (Note that this condition does not apply to unidirectional datasets.)
- Calls *SetRange* on an empty range or dataset. (client datasets only)

Bof is set to false in all other cases; you should assume *Bof* is false unless one of the conditions above is met *and* you test the property directly.

Like *Eof*, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

D

Delphi example

```
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.BOF do { Cycle until BOF is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Prior; { BOF False on success; BOF True when Prior fails on first record }
  end;
finally
  CustTable.EnableControls; { Display new current row in controls }
end;
```

C++ example

```
CustTable->DisableControls(); // Speed up processing; prevent screen flicker
try
{
  while (!CustTable->Bof) // Cycle until Bof is true
  (
    // Process each record here
    ⋮
    CustTable->Prior();
    // Bof false on success; Bof true when Prior fails on first record
  }
}
__finally
{
  CustTable->EnableControls();
}
```

Marking and returning to records

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. *TDataSet* introduces a bookmarking feature that lets you tag records and return to them later. The bookmarking feature consists of a *Bookmark* property and five bookmark methods.

The *Bookmark* property indicates which bookmark among any number of bookmarks in your application is current. *Bookmark* is a string that identifies the current bookmark. Each time you add another bookmark, it becomes the current bookmark.

TDataSet implements virtual bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. Unidirectional datasets do not add support for bookmarks. *TCustomClientDataSet*, however, reimplements the bookmark methods to return meaningful values as described in the following list:

- *BookmarkValid*, for determining if a specified bookmark is in use.
- *CompareBookmarks*, to test two bookmarks to see if they are the same.
- *GetBookmark*, to allocate a bookmark for your current position in the dataset.
- *GotoBookmark*, to return to a bookmark previously created by *GetBookmark.*
- *FreeBookmark*, to free a bookmark previously allocated by *GetBookmark*.

To create a bookmark, you must declare a variable of type *TBookmark* in your application. The *TBookmark* type is a pointer (in C++, void *). When you call *GetBookmark*, the dataset allocates storage for the bookmark and sets your variable to point to that storage. The bookmark contains information that identifies a particular location in the dataset.

Before calling *GotoBookmark* to move to a specific record, you can call *BookmarkValid* to determine if the bookmark points to a record. *BookmarkValid* returns true if a specified bookmark points to a record. In *TDataSet*, *BookmarkValid* is a virtual method that always returns false, indicating that the bookmark is not valid. Descendants that support bookmarks reimplement this method to provide a meaningful return value.

You can also call *CompareBookmarks* to see if a bookmark to which you want to move is different from another (or the current) bookmark. *CompareBookmarks* always returns 0, indicating that the bookmarks are identical. Descendants that support bookmarks reimplement this method to provide a meaningful return value.

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. *TDataSet* generates an error when you call *GotoBookmark*. Unidirectional datasets do nothing. Descendants that support bookmarks reimplement this method to provide a meaningful return value.

FreeBookmark frees the memory allocated for a specified bookmark when you no longer need it. You should also call *FreeBookmark* before reusing an existing bookmark.

The following code illustrates one use of bookmarking:

Delphi example

```
procedure DoSomething (const MyDS: TSQLClientDataSet)
var
  Bookmark: TBookmark;
begin
  Bookmark := MyDS.GetBookmark; { Allocate memory and assign a value }
  MyDS.DisableControls; { Turn off display of records in data controls }
  try
    MyDS.First; { Go to first record in table }
    while not MyDS.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      ⋮
      MyDs.Next;
    end;
  finally
    MyDs.GotoBookmark(Bookmark);
    MyDs.EnableControls; { Turn on display of records in data controls, if necessary }
    MyDs.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;
```

C++ example

```
void DoSomething (const TSQLClientDataSet *MyDS)
{
  TBookmark Bookmark = MyDS->GetBookmark(); // Allocate memory and assign a value
  MyDS->DisableControls(); // Turn off display of records in data controls
  try
  {
    for (MyDS->First(); !MyDS->Eof; MyDS->Next()) // Iterate through each record in table
    {
      // Do your processing here
      ⋮
    }
  }
  __finally
  {
    MyDS->GotoBookmark(Bookmark);
    MyDS->EnableControls(); // Turn on display of records in data controls
    MyDS->FreeBookmark(Bookmark); // Deallocate memory for the bookmark
  }
}
```

Before iterating through records, controls are disabled. Should an error occur during iteration through records, the **finally** clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

Searching datasets

If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods of *TDataSet*. These methods enable you to search on any type of columns in any dataset.

Note Client datasets introduce an additional family of methods that let you search for records based on an index. For information about searching a client dataset based on its index, see “Navigating data in client datasets” on page 25-2.

Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is “Professional Divers, Ltd.”:

Delphi example

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',
    SearchOptions);
end;
```

C++ example

```
TLocateOptions SearchOptions;
SearchOptions.Clear();
SearchOptions << loPartialKey;
bool LocateSuccess = CustTable->Locate("Company", "Professional Divers, Ltd.",
    SearchOptions);
```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns true if it finds a matching record, false if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are Variants, which means you can specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array type as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array on the fly using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

Delphi example

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

C++ example

```
TLocateOptions Opts;
Opts.Clear();
Opts << loPartialKey;
Variant locvalues[2];
locvalues[0] = Variant("Sight Diver");
locvalues[1] = Variant("P");
CustTable->Locate("Company;Contact", VarArrayOf(locvalues, 1), Opts);
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index.

Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of the *Company* field is "Professional

Divers, Ltd.”, and returns the company name, a contact person, and a phone number for the company:

Delphi example

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company;
    Contact; Phone');
end;
```

C++ example

```
Variant LookupResults = CustTable->Lookup(“Company”, “Professional Divers, Ltd”,
   “Company;Contact;Phone”);
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as Variants. If more than one return value is requested, *Lookup* returns a Variant array. If there are no matching records, *Lookup* returns a Null Variant. For more information about Variant arrays, see the online help.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual fields in the string items with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array type as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array on the fly using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

Delphi example

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
    'Company; Addr1; Addr2; State; Zip');
end;
```

C++ example

```
Variant LookupResults;
Variant locvalues[2];
Variant v;

locvalues[0] = Variant(“Sight Diver”);
locvalues[1] = Variant(“Kato Paphos”);
LookupResults = CustTable->Lookup(“Company;City”, VarArrayOf(locvalues, 1),
                “Company;Addr1;Addr2;State;Zip”);
// now put the results in a global stringlist (created elsewhere)
```

```
pFieldValues->Clear();
for (int i = 0; i < 5; i++) // Lookup call requested 5 fields
{
  v = LookupResults.GetElement(i);
  if (v.IsNull())
    pFieldValues->Add("");
  else
    pFieldValues->Add(v);
}
```

Lookup uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index.

Displaying and editing a subset of data using filters

An application is frequently interested in only a subset of records within a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values.

With unidirectional datasets, you can only limit the records in the dataset when you specify the SQL command that fetches the records. With other *TDataSet* descendants, however, you can define a subset of the data that has already been fetched. To restrict an application's access to a subset of all records in the dataset, you can use filters.

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's *Filter* property or coded into its *OnFilterRecord* event handler. Filter conditions are based on the values in any specified number of fields in a dataset whether or not those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

Note Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to restrict record retrieval using the SQL command that fetches the data, or to set a range on an indexed client dataset rather than using filters.

Enabling and disabling filtering

Enabling filters on a dataset is a three-step process:

1. Create a filter.
2. Set filter options for string-based filter tests, if necessary.
3. Set the *Filtered* property to true.

When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the *Filtered* property to false.

Creating filters

There are two ways to create a filter for a dataset:

- Specify simple filter conditions in the *Filter* property. *Filter* is especially useful for creating and applying filters at runtime.
- Write an *OnFilterRecord* event handler for simple or complex filter conditions. With *OnFilterRecord*, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions.

The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for example, in response to user input). Its main disadvantages are that filter conditions must be expressible in a single text string, cannot make use of branching and looping constructs, and cannot test or compare its values against values not already in the dataset.

The strengths of the *OnFilterRecord* event are that a filter can be complex and variable, can be based on multiple lines of code that use branching and looping constructs, and can test dataset values against values outside the dataset, such as the text in an edit box. The main weakness of using *OnFilterRecord* is that you set the filter at design time and it cannot be modified in response to user input. (You can, however, create several filter handlers and switch among them in response to general application conditions.)

The following sections describe how to create filters using the *Filter* property and the *OnFilterRecord* event handler.

Setting the Filter property

To create a filter using the *Filter* property, set the value of the property to a string that contains the filter conditions. The string contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

```
Dataset1->Filter = "State = 'CA'";
```

You can also supply a value for *Filter* based on the text entered in a control. For example, the following statement assigns the text in an edit box to *Filter*:

```
Dataset1.Filter := Edit1.Text;
```

```
Dataset1->Filter = Edit1->Text;
```

You can, of course, create a string based on both hard-coded text and data entered by a user in a control:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

```
Dataset1->Filter = AnsiString("State = '") + Edit1->Text + "'";
```

Blank records do not appear unless they are explicitly included in the filter:

```
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
```

```
Dataset1->Filter = "State <> 'CA' or State = BLANK";
```

After you specify a value for *Filter*, to apply the filter to the dataset, set the *Filtered* property to true.

You can also compare field values to literals and to constants or calculate values using the following operators:

Table 22.4 Operators that can appear in a filter

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to

Table 22.4 Operators that can appear in a filter (continued)

Operator	Meaning
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both true
NOT	Tests that the following statement is not true
OR	Tests that at least one of two statements is true
IS NULL	Tests that a field value is null.
IS NOT NULL	Tests that a field value is not null.
+	Adds numbers, concatenates strings, adds numbers to date/time values
-	Subtracts numbers, subtracts dates, or subtracts a number from a date
*	Multiplies two numbers
/	Divides two numbers
Upper	Upper-cases a string
Lower	Lower-cases a string
Substring	Returns the substring starting at a specified position.
Trim	Trims spaces or a specified character from front and back of a string.
TrimLeft	Trims spaces or a specified character from front of a string.
TrimRight	Trims spaces or a specified character from back of a string.
Year	Returns the year from a date/time value
Month	Returns the month from a date/time value
Day	Returns the day from a date/time value
Hour	Returns the hour from a time value
Minute	Returns the minute from a time value
Second	Returns the seconds from a time value
GetDate	Returns the current date
Date	Returns the date part of a date/time value
Time	Returns the time part of a date/time value
Like	Provides pattern matching in string comparisons.
In	Tests for set inclusion.
*	Wildcard for partial comparisons.

By using combinations of these operators, you can create fairly sophisticated filters. For example, the following statement checks to make sure that two test conditions are met before accepting a record for display:

```
(Custno > 1400) AND (Custno < 1500);
```

Note When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

Writing an OnFilterRecord event handler

A filter for a dataset is an event handler that responds to *OnFilterRecord* events generated by the dataset for each record it retrieves. At the heart of every filter handler is a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your filter handler must set an *Accept* parameter to true to include a record, or false to exclude it. For example, the following filter displays only those records with the State field set to “CA”:

Delphi example

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'].AsString = 'CA';
end;
```

C++ example

```
void __fastcall TForm1::Table1FilterRecord(TDataSet *DataSet; bool &Accept)
{
  Accept = DataSet->FieldByName[“State”]->AsString == “CA”;
}
```

When filtering is enabled, an *OnFilterRecord* event is generated for each record retrieved. The event handler tests each record, and only those that meet the filter’s conditions are displayed. Because the *OnFilterRecord* event is generated for every record in a dataset, you should keep the event handler as tightly-coded as possible to avoid adversely affecting the performance of your application.

Switching filter event handlers at runtime

You can code any number of filter event handlers and switch among them at runtime. To switch to a different filter event handler at runtime, assign the new event handler to the dataset’s *OnFilterRecord* property.

For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

```
DataSet1->OnFilterRecord = NewYorkFilter;
Refresh();
```

Setting filter options

The *FilterOptions* property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

Table 22.5 FilterOptions values

Value	Meaning
foCaseInsensitive	Ignore case when comparing strings.
foNoPartialCompare	Disable partial string matching (i.e., do not match strings ending with an asterisk (*)).

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

Delphi example

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

C++ example

```
TFilterOptions FilterOptions;
FilterOptions->Clear();
FilterOptions << foCaseInsensitive;
Table1->FilterOptions = FilterOptions;
Table1->Filter = "State = 'CA'";
```

Navigating records in a filtered dataset

There are four dataset methods that enable you to navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

Table 22.6 Filtered dataset navigational methods

Method	Purpose
FindFirst	Move to the first record in the dataset that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset.
FindLast	Move to the last record in the dataset that matches the current filter criteria.
FindNext	Moves from the current record in the filtered dataset to the next one.
FindPrior	Move from the current record in the filtered dataset to the previous one.

For example, the following statement finds the first filtered record in a dataset:

```
DataSet1.FindFirst;
```

```
DataSet1->FindFirst();
```

Provided that you set the *Filter* property or create an *OnFilterRecord* event handler for your application, these methods position the cursor on the specified record regardless of whether filtering is currently enabled. If you call these methods when filtering is not enabled, then they

- Temporarily enable filtering.
- Position the cursor on a matching record if one is found.
- Disable filtering.

Note If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*, *Last*, *Next*, and *Prior*.

All navigational filter methods position the cursor on a matching record (if one is found), make that record the current one, and return true. If a matching record is not found, the cursor position is unchanged, and these methods return false. You can check the status of the *Found* property to wrap these calls, and only take action when *Found* is true. For example, if the cursor is already on the last matching record in the dataset, and you call *FindNext*, the method returns false, and the current record is unchanged.

Modifying data

You can use the dataset methods listed below to insert, update, and delete data if the read-only *CanModify* property for the dataset is true. *CanModify* is true unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor (such as a client dataset's *ReadOnly* property or a dataset provider's *poReadOnly* option) intervenes:

Table 22.7 Dataset methods for inserting, updating, and deleting data

Method	Description
Edit	Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state.
Insert	Posts any pending data, and puts the dataset in *dsInsert* state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset in *dsBrowse* state.
Delete	Deletes the current record and puts the dataset in *dsBrowse* state.

Editing records

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is true.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is false (the default),
- The *AutoEdit* property of the data source for the control is true, and
- *CanModify* is true for the dataset.

Note Even if a dataset is in *dsEdit* state, editing records may not succeed if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you provide a navigator component on your forms, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling *Post*. You cancel them by calling *Cancel*. In code, *Edit* and *Post* are often used together. For example,

Delphi example

```
with CustTable do
begin
  Edit;
  FieldValues['CustNo'] := 1234;
  Post;
end;
```

C++ example

```
ClientDataSet1->Edit();
ClientDataSet1->FieldValues["CustNo"] = 1234;
CleintDataSet1->Post();
```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record to the change log.

Note When you want to write the edits in the change log back to the database, you must call the *ApplyUpdates* method of a client dataset. For more information about editing in client datasets, see "Editing data" on page 25-16.

Adding new records

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is true.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is false (the default), and
- *CanModify* is true for the dataset.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the *Next* and *Last* buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

Post writes the new record to the change log. To write inserts and appends from the change log to the database, call the *ApplyUpdates* method of a client dataset.

Inserting records

Insert opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post*, a newly inserted record is written to the change log as follows:

- For indexed datasets, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is inserted into the dataset at its current position.

Appending records

Append opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post*, a newly appended record is written to the change log as follows:

- For indexed datasets, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is added to the end of the dataset.

Note When the dataset applies the updates in the change log, the physical location of inserted and appended records in the underlying SQL database is implementation-specific. If the table is indexed, the index is updated with the new record information.

Deleting records

A dataset must be active before an application can delete records. *Delete* removes the current record from the dataset into the change log and puts the dataset in *dsBrowse* mode. The record that followed the deleted record becomes the current record. A deleted record is not removed from the change log until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

Posting data to the database

The *Post* method is central to a database application's ability to edit. *Post* writes changes to the current record to the change log, but it behaves differently depending on a dataset's state.

- In *dsEdit* state, *Post* writes a modified record to the change log.
- In *dsInsert* state, *Post* writes a new record to the change log.
- In *dsSetKey* state, *Post* returns the dataset to *dsBrowse* state.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, *Post* is called implicitly. Calls to the *First*, *Next*, *Prior*, and *Last* methods perform a *Post* if the table is in *dsEdit* or *dsInsert* modes. The *Append* and *Insert* methods also implicitly post any pending data.

Warning The *Close* method does not call *Post* implicitly. Use the *BeforeClose* event to post any pending edits explicitly.

Canceling changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called *Post*. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the *Cancel* method for the dataset. A call to *Cancel* always returns a dataset to *dsBrowse* state.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

Client datasets introduce additional methods to cancel edits on non-current records. See "Undoing changes" on page 25-17 for details.

Modifying entire records

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

Table 22.8 Methods that work with entire records

Method	Description
AppendRecord([array of values])	Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit *Post*.
InsertRecord([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*.
SetFields([array of values])	Sets the values of the corresponding fields; analogous to assigning values to *TField*s. Application must perform an explicit *Post*.

These methods each take an array of values as an argument, where each value corresponds to a column in the underlying dataset. In C++, use the ARRAYOFCONST macro to create these arrays. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, *AppendRecord* adds a record to the end of the dataset and *InsertRecord* inserts a record after the current cursor position. For indexed tables, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

SetFields assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call *Edit* to put the dataset in *dsEdit* mode. To write out the changes to the current record, it must perform a *Post*.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, SetFields assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, consider a client dataset called CountryTable with columns for Name, Capital, Continent, Area, and Population. The following statement would insert a record into the client dataset:

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

```
CountryTable->InsertRecord(ARRAYOFCONST((“Japan”, “Tokyo”, “Asia”)));
```

This statement does not specify values for Area and Population, so NULL values are inserted for them. The dataset is indexed on Name, so the statement would insert the record based on the alphabetic collation of “Japan”.

To update the record, an application could use the following code:

Delphi example

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
```

```
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

C++ example

```
TLocateOptions SearchOptions;
SearchOptions->Clear();
SearchOptions << loCaseInsensitive;
if (CountryTable->Locate("Name", "Japan", SearchOptions))
{
  CountryTable->Edit();
  CountryTable->SetFields(ARRAYOFCONST(((void *)NULL, (void *)NULL, (void *)NULL,
        344567, 164700000)));
  CountryTable->Post();
}
```

This code assigns values to the Area and Population fields and then posts them to the change log. The three nil arguments (Delphi) or NULL pointers (C++) act as place holders for the first three columns to preserve their current contents.

Warning In C++, when using NULL pointers with *SetFields* to leave some field values untouched, be sure to cast the NULL to a void *. If you use NULL as a parameter without the cast, you will set the field to a blank value.

Using dataset events

TDataSet defines a number of events that enable an application to perform validation, compute totals, and perform other tasks. The events are listed in the following table.

Table 22.9 Dataset events

Event	Description
BeforeOpen, AfterOpen	Called before/after a dataset is opened.
BeforeClose, AfterClose	Called before/after a dataset is closed.
BeforeInsert, AfterInsert	Called before/after a dataset enters Insert state.
BeforeEdit, AfterEdit	Called before/after a dataset enters Edit state.
BeforePost, AfterPost	Called before/after changes to a table are posted.
BeforeCancel, AfterCancel	Called before/after the previous state is canceled.
BeforeDelete, AfterDelete	Called before/after a record is deleted.
OnNewRecord	Called when a new record is created; used to set default values.
OnCalcFields	Called when calculated fields are calculated.

Aborting a method

To abort a method such as an *Open* or *Insert*, call the *Abort* procedure in any of the *Before* event handlers (*BeforeOpen*, *BeforeInsert*, and so on). For example, the following code requests a user to confirm a delete operation or else it aborts the call to *Delete*:

Delphi example

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)
begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

C++ example

```
void __fastcall TForm1::TableBeforeDelete (TDataSet *Dataset)
{
  TMsgDlgButtons btns;
  btns << mbYes << mbNo;
  if (MessageDlg("Delete this record?", mtConfirmation, btns, 0) == mrYes)
    Abort();
}
```

Using OnCalcFields

The *OnCalcFields* event is used to set the values of calculated fields. The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is true, *OnCalcFields* is called when

- A dataset is opened.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.
- A record is retrieved from the database.

OnCalcFields is always called whenever a value in a non-calculated field changes, regardless of the setting of *AutoCalcFields*.

Caution *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is true, *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is true, then *OnCalcFields* is called again, leading to another *Post*, and so on.

If *AutoCalcFields* is false, then *OnCalcFields* is not called when individual fields within a single record are modified.

When *OnCalcFields* executes, a dataset is in *dsCalcFields* mode, so you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

Chapter

23

Working with field components

This chapter describes the properties, events, and methods common to the *TField* object and its descendants. Field components represent individual fields (columns) in datasets. This chapter also describes how to use field components to control the display and editing of data in your applications.

Field components are always associated with a dataset. You never use a *TField* object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as *TDBEdit* and *TDBGrid* access to the data in a particular column of the associated dataset.

Generally speaking, a single field component represents the characteristics of a single column, or field, in a dataset, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. For example, a *TFloatField* component has four properties that directly affect the appearance of its data:

Table 23.1 TFloatField properties that affect data display

Property	Purpose
Alignment	Specifies whether data is displayed left-aligned, centered, or right-aligned.
DisplayWidth	Specifies the number of digits to display in a control at one time.
DisplayFormat	Specifies data formatting for display (such as how many decimal places to show).
EditFormat	Specifies how to display a value during editing.

As you scroll from record to record in a dataset, a field component lets you view and change the value for that field in the current record.

Field components have many properties in common with one another (such as *DisplayWidth* and *Alignment*), and they have properties specific to their data types (such as *Precision* for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications. The following sections describe dynamic and persistent fields in more detail and offer advice on choosing between them.

Dynamic field components

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact *TField* descendant created for each column is determined by field type information received from the database or (for client datasets) from a provider component.

Dynamic fields are temporary. They exist only as long as a dataset is open. Each time you reopen a dataset that uses dynamic fields, it rebuilds a completely new set of dynamic field components based on the current structure of the data underlying the dataset. If the columns in the underlying data change, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database browsing tool, you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the datasets used by the application change frequently.

To use dynamic fields in an application:

1. Place datasets and data sources in a data module.
2. Associate the datasets with data. This involves using a connection component or provider to connect to the source of the data and setting any properties that specify what data the dataset represents.
3. Associate the data sources with the datasets.
4. Place data-aware controls in the application's forms, in Delphi add the data module to each uses clause for each form's unit, in C++ include the data module's header in each form's unit, and then associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one. Note that because you are using dynamic field components, there is no guarantee that any fieldname you specify will exist when the dataset is opened.
5. Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

Persistent field components

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you

- Set or change the field's display or edit characteristics at design time or runtime.
- Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset.
- Validate data entry.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields to replace existing fields, based on columns in the table or query underlying a dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

Note When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always use the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see "Dynamic field components" on page 23-2.

Note One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control the appearance of columns in data-aware grids. To learn about controlling column appearance in grids, see "Creating a customized grid" on page 20-17.

Creating persistent fields

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying data changes. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, your application raises an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset:

1 Place a dataset in a data module.

2 Bind the dataset to its underlying data. This typically involves associating the dataset with a connection component or provider and specifying any properties to describe the data. For example, If you are using *TSQLDataSet*, you can set the *SQLConnection* property to a properly configured *TSQLConnection* component and set the *CommandText* property to a valid query.

3 Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box.

   The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module (Delphi), the title bar displays 'CustomerData.Customers,' or as much of the name as fits.

   Below the title bar is a set of navigation buttons that let you scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty. If the dataset is unidirectional, the buttons for moving to the last record and the previous record are always dimmed.

   The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.

4 Choose Add Fields from the Fields editor context menu.

5 Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and (if the dataset is not unidirectional) Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, it no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, it verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, the dataset raises an exception warning you that the field is not valid, and does not open.

Arranging persistent fields

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields:

1 Select the fields. You can select and order one or more fields at a time.

2 Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use *Ctrl+Up* and *Ctrl+Dn* to change an individual field's order in the list.

Defining new persistent fields

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements for the other persistent fields in a dataset.

New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

- The Field properties group box lets you enter general field component information. Enter the field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName* property. The New Field dialog uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's *Name* property and is only provided for informational purposes (*Name* is the identifier by which you refer to the field component in your source code). The dialog discards anything you enter directly in the Component edit box.

- The Type combo box in the Field properties group lets you specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. Use the Size edit box to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.
- The Field type radio group lets you specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you are working with a client dataset, you can create InternalCalc fields or Aggregate fields. The following table describes these types of fields you can create:

Table 23.2 Special persistent field kinds

Field kind	Purpose
Data	Replaces an existing field (for example to change its data type).
Calculated	Displays values calculated at runtime by a dataset's *OnCalcFields* event handler.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify (not supported for unidirectional datasets).
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Aggregate	Displays a value summarizing the data in a set of records from a client dataset.

The Lookup definition group box is only used to create lookup fields. This is described more fully in "Defining a lookup field" on page 23-8.

Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallIntField* with a *TIntegerField*. Because you cannot change a field's data type directly, you must define a new field to replace it.

Important Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset, follow these steps:

1. Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.
2. In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.
3. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.

4 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

5 Select Data in the Field type radio group if it is not already selected.

6 Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's type declaration (Delphi) or header (C++) is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 23-11.

Defining a calculated field

A calculated field displays values calculated at runtime by a dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1 Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.

2 Choose a data type for the field from the Type combo box.

3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

4 Select Calculated or InternalCalc in the Field type radio group. InternalCalc is only available if you are working with a client dataset. The significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.

5 Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's or data module's type declaration (Delphi) or header (C++).

6 Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field" on page 23-7.

Note To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 23-11.

Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field:

1. Select the dataset component from the Object Inspector drop-down list.
2. Choose the Object Inspector Events page.
3. Double-click the *OnCalcFields* property to bring up or create a *CalcFields* event handler for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
  + ' ' + CustomersZip.Value;
```

```
CustomersCityStateZip->Value = CustomersCity->Value + AnsiString(“, ”) +
    CustomersState->Value + AnsiString(“ ”) + CustomersZip->Value;
```

Note When writing the *OnCalcFields* event handler for an internally calculated field, you can improve performance by checking the client dataset's *State* property and only recomputing the value when *State* is *dsInternalCalc*. See "Using internally calculated fields in client datasets" on page 25-22 for details.

Defining a lookup field

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. In Delphi, the column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

Note Unidirectional datasets do not support lookup fields.

To create a lookup field in the New Field dialog box:

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.

3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

4 Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.

5 Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.

6 Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.

7 Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.

8 Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

You can use the *LookupCache* property to hone the way lookup fields are determined. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes. Set *LookupCache* to true to cache the values of a lookup field when the *LookupDataSet* is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of *LookupKeyFields* values are preloaded when the *DataSet* is opened. When the current record in the *DataSet* changes, the field object can locate its *Value* in the cache, rather than accessing the *LookupDataSet*. This performance improvement is especially dramatic if the *LookupDataSet* is on a network where access is slow.

Tip You can use a lookup cache to provide lookup values programmatically rather than from a secondary dataset. Be sure that the *LookupDataSet* property is nil (Delphi) or NULL (C++). Then, use the *LookupList* property's *Add* method to fill it with lookup values. Set the *LookupCache* property to true. The field will use the supplied lookup list without overwriting it with values from a lookup dataset.

If every record of *DataSet* has different values for *KeyFields*, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by *KeyFields*.

If *LookupDataSet* is volatile, caching lookup values can lead to inaccurate results. Call *RefreshLookupList* to update the values in the lookup cache. *RefreshLookupList* regenerates the *LookupList* property, which contains the value of the *LookupResultField* for every set of *LookupKeyFields* values.

When setting *LookupCache* at runtime, call *RefreshLookupList* to initialize the cache.

Defining an aggregate field

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See "Using maintained aggregates" on page 25-22 for details about maintained aggregates.

To create an aggregate field in the New Field dialog box:

1 Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.

2 Choose aggregate data type for the field from the Type combo box.

3 Select Aggregate in the Field type radio group.

4 Choose OK. The newly defined aggregate field is automatically added and the client dataset's *Aggregates* property is automatically updated to include the appropriate aggregate specification.

5 Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see "Specifying aggregates" on page 25-22.

Once a persistent *TAggregateField* is created, a *TDBText* control can be bound to the aggregate field. The *TDBText* control will then display the value of the aggregate field that is relevant to the current record of the client data set.

Deleting persistent field components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table. To remove one or more persistent field components for a dataset:

1 Select the field(s) to remove in the Fields editor list box.

2 Press *Del*.

Note You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate a persistent field component that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see "Creating persistent fields" on page 23-4.

Note If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

Setting persistent field properties and events

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a *TDBGrid*, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the Object Inspector.

Setting display and edit properties at design time

To edit the display properties of a selected field component, switch to the Properties page on the Object Inspector window. The following table summarizes display properties that can be edited.

Table 23.3 Field component properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers a field contents within a data-aware component.
ConstraintErrorMessage	Specifies the text to display when edits clash with a constraint condition.
CustomConstraint	Specifies a local constraint to apply to data during editing.
Currency	Numeric fields only. true: displays monetary values. false (default): does not display monetary values.
DisplayFormat	Specifies the format of data displayed in a data-aware control.
DisplayLabel	Specifies the column name for a field in a data-aware grid.
DisplayWidth	Specifies the width, in characters, of a grid column that displays this field.
EditFormat	Specifies the edit format of data in a data-aware control.
EditMask	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on).
FieldKind	Specifies the type of field (data, lookup, calculated, or aggregate).
FieldName	Specifies the name of a column in the table from which the field derives its value and data type.
HasConstraints	Indicates whether there are constraint conditions imposed on a field.
ImportedConstraint	Specifies an SQL constraint imported from the database server. This is only used when the field gets its value from an application server running on another platform.
Index	Specifies the order of the field in a dataset.
LookupDataSet	Specifies the dataset used to look up field values when *Lookup* is true.
LookupKeyFields	Specifies the field(s) in the lookup dataset to match when doing a lookup.
LookupResultField	Specifies the field in the lookup dataset from which to copy values into this field.
MaxValue	Numeric fields only. Specifies the maximum value a user can enter for the field.

Table 23.3 Field component properties (continued)

Property	Purpose
MinValue	Numeric fields only. Specifies the minimum value a user can enter for the field.
Name	Specifies the name used to refer to the field component in code.
Origin	Specifies the name of the field as it appears in the underlying database.
Precision	Numeric fields only. Specifies the number of significant digits.
ReadOnly	true: Displays field values in data-aware components, but prevents editing. false (the default): Permits display and editing of field values.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField* fields.
Tag	Long integer available for programmer use in every component as needed.
Transliterate	true (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. false: Locale translation does not occur.
Visible	true (the default): Permits display of field in a data-aware grid. false: Prevents display of field in a data-aware grid component. User-defined components can make display decisions based on this property.

Not all properties are available for all field components. For example, a field component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type *TFloatField* does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see "Controlling and masking user input" on page 23-13.

Setting field component properties at runtime

You can use and manipulate the properties of field component at runtime. For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to true:

```
CustomersCityStateZip.ReadOnly := True;
```

```
CustomersCityStateZip->ReadOnly = true;
```

And this statement changes field ordering by setting the *Index* property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip.Index := 3;
```

```
CustomersCityStateZip->Index = 3;
```

Controlling and masking user input

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with *TStringField*, *TDateField*, *TTimeField*, *TDateTimeField*, and *TSQLTimeStampField* components. You can use existing masks, or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the Object Inspector.

Note For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component:

1 Select the component in the Fields editor or Object Inspector.

2 Click the Properties page in the Object Inspector.

3 Double-click the values column for the EditMask field in the Object Inspector, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box lets you create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button lets you load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

Using default formatting for numeric, date, and time fields

The IDE provides built-in display and edit format routines and intelligent default formatting for *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, *TTimeField*, and *TSQLTimeStampField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Table 23.4 Field component formatting routines

Routine	Used by . . .
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField, TTimeField, TDateTimeField, TSQLTimeStampField*
SQLTimeStampToString	*TSQLTimeStampField*
FormatCurr	*TCurrencyField, TBCDField*
BcdToStrF	*TFMTBcdField*

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the system locale. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to true sets the *DisplayFormat* property for the value 1234.56 to $1234.56, while the *EditFormat* is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and *EditFormat* properties of a field component to override the default display settings for that field. You can also write *OnGetText* and *OnSetText* event handlers to do custom formatting for field components at runtime.

Handling events

Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components:

Table 23.5 Field component events

Event	Purpose
OnChange	Called when the value for a field changes.
OnGetText	Called when the value for a field component is retrieved for display or editing.
OnSetText	Called when the value for a field component is set.
OnValidate	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

OnGetText and *OnSetText* events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. *OnValidate* is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component:

1. Select the component.
2. Select the Events page in the Object Inspector.
3. Double-click the Value field for the event handler to display its source code window.
4. Create or edit the handler code.

Working with field component methods at runtime

Field components methods available at runtime let you convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as *BeforePost*). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's *FocusControl* method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler can call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
CustomersCompany.FocusControl;
```

```
CustomersCompany->FocusControl();
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see the entries for *TField* and its descendants in the online *CLX Reference*.

Table 23.6 Selected field component methods

Method	Purpose
AssignValue	Sets a field value to a specified value using an automatic conversion function based on the field's type.
Clear	Clears the field and sets its value to NULL.
GetData	Retrieves unformatted data from the field.
IsValidChar	Determines if a character entered by a user in a data-aware control to set a value is allowed for this field.
SetData	Assigns unformatted data to this field.

Displaying, converting, and accessing field values

Data-aware controls such as *TDBEdit* and *TDBGrid* automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Chapter 20, "Using data controls."

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part. For example, when using standard controls, your application is responsible for tracking when to update controls because field values change. If the dataset has a datasource component, you can use its events to help you do this. In particular, the *OnDataChange* event lets you know when you may need to update a control's value and the *OnStateChange* event can help you determine when to enable or disable controls. For more information on these events, see "Responding to changes mediated by the data source" on page 20-4.

The following topics discuss how to work with field values so that you can display them in standard controls.

Displaying field component values in standard controls

An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a *TEdit* control because the value of the *CustomersCompany* field may have changed:

Delphi example

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
  Edit3.Text := CustomersCompany.Value;
end;
```

C++ example

```
void __fastcall TForm1::Table1DataChange(TObject *Sender, TField *Field)
{
  Edit3->Text = CustomersCompany->Value;
}
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in functions for handling conversions.

Note You can also use Variants to access and set field values. For more information about using variants to access and set field values, see "Accessing field values with the default dataset property" on page 23-18.

Converting field values

Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations. The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField	yes	NA	yes	yes	yes	yes
TWideStringField	yes	yes	yes	yes	yes	yes
TIntegerField	yes	yes	NA	yes		
TSmallIntField	yes	yes	yes	yes		
TWordField	yes	yes	yes	yes		
TLargeintField	yes	yes	yes	yes		
TFloatField	yes	yes	yes	yes		
TCurrencyField	yes	yes	yes	yes		
TBCDField	yes	yes	yes	yes		

TFMTBCDField	yes	yes	yes	yes		
TDateTimeField	yes	yes		yes	yes	
TDateField	yes	yes		yes	yes	
TTimeField	yes	yes		yes	yes	
TSQLTimeStampField	yes	yes		yes	yes	
TBooleanField	yes	yes				
TBytesField	yes	yes				
TVarBytesField	yes	yes				
TBlobField	yes	yes				
TMemoField	yes	yes				
TGraphicField	yes	yes				
TVariantField	NA	yes	yes	yes	yes	yes
TAggregateField	yes	yes				

Note that some columns in the table refer to more than one conversion property (such as *AsFloat*, *AsCurrency*, and *AsBCD*). This is because all field data types that support one of those properties always support the others as well.

Note also that the *AsVariant* property can translate among all data types. For any datatypes not listed above, *AsVariant* is also available (and is, in fact, the only option). When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. Table 23.7 lists permissible conversions that produce special results:

Table 23.7 Special conversion results

Conversion	Result
String to Boolean	Converts True, False, Yes, and No to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime or SQLTimeStamp to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to True or False.

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

Conversion always occurs before an actual assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
Edit1.Text := CustomersCustNo.AsString;
```

```
Edit1->Text = CustomersCustNo->AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

```
MyTableMyField->AsInteger = StrToInt(Edit1->Text);
```

Accessing field values with the default dataset property

The most general method for accessing a field's value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

```
Customers->FieldValues["CustNo"] = Edit2->Text;
```

Because the *FieldValues* property is of type Variant, it automatically converts other datatypes into a Variant value.

For more information about Variants, see the online help.

Accessing field values with a dataset's Fields property

You can access the value of a field with the *Fields* property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using each field component's conversion properties. For more information about field component conversion properties, see "Converting field values" on page 23-16.

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

```
Edit1->Text = CustTable->Fields->Fields[6]->AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

Delphi example

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

C++ example

```
Customers->Edit();
Customers->Insert();
Customers->Fields->Fields[6]->AsString = Edit1->Text;
Customers->Post();
```

Accessing field values with a dataset's FieldByName method

You can also access the value of a field with a dataset's *FieldByName* method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

```
Edit2->Text = Customers->FieldByName("CustNo")->AsString;
```

Conversely, you can assign a value to a field:

Delphi example

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

C++ example

```
Customers->Edit();
Customers->FieldByName("CustNo")->AsString = Edit2->Text;
Customers->Post();
```

Checking a field's current value

If your application uses a client datasetto update data from a database server, and you encounter difficulties when updating records, you can use the *CurValue* property to examine the field value in the record causing problems. *CurValue* represents the current value of the field on the database server, reflecting any changes made by other users of the database.

CurValue is only available inside an *OnUpdateError* or *OnReconcileError* event handler. In these event handlers, you can compare *CurValue* (the value on the server) to *NewValue* (the unposted value that caused the problem) and *OldValue* (the value that was originally assigned to the field before any edits were made). *CurValue* differs from *OldValue* only if another user changed the value of the field after *OldValue* was read.

Setting a default value for a field

You can specify how a default value for a field in a client dataset should be calculated at runtime using the *DefaultExpression* property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be:

```
‘12:00:00’
```

including the quotes around the literal value.

Note If the underlying database table defines a default value for the field, the default you specify in *DefaultExpression* takes precedence. That is because *DefaultExpression* is applied when the client dataset posts the record containing the field, which occurs before the edited record is applied to the database server.

Specifying constraints

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. These constraints are enforced when your application applies updates to the database server.

In addition to these server-enforced constraints, you can create and use custom constraints that are applied locally to the fields in client datasets. Custom constraints can duplicate the server constraints (so that you detect errors immediately when posting edits to the change log rather than later when you attempt to apply updates), or they can impose additional, application-defined limits. Custom constraints provide validation of data entry, but they cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

CustomConstraint is an SQL string that specifies any application-specific constraints imposed on the field’s value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as:

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

Using object fields

Object fields are fields that represent a composite of other, simpler datatypes. These include ADT (Abstract Data Type) fields, array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets.

ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type.

Dataset and reference fields map to fields that access other data sets. A dataset field provides access to a nested (detail) dataset and a reference field stores a pointer (reference) to another persistent object (ADT).

Table 23.8 Types of object field components

Component name	Purpose
TADTField	Represents an ADT (Abstract Data Type) field.
TArrayField	Represents an array field.
TDataSetField	Represents a field that contains a nested data set reference.
TReferenceField	Represents a REF field, a pointer to an ADT.

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's *ObjectView* property to true, which instructs the dataset to store these fields hierarchically, rather than flattening them out as if the constituent child fields were separate, independent fields.

The following properties are common to all object fields and provide the functionality to handle child fields and datasets.

Table 23.9 Common object field descendant properties

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classifies the object field.
FieldCount	Number of child fields that comprise the object field.
FieldValues	Provides access to the values of the child fields.

Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls.

Data-aware controls such as *TDBEdit* that represent a single field value display child field values in an uneditable comma delimited string. In addition, if you set the control's *DataField* property to the child field instead of the object field itself, the child field can be viewed an edited just like any other normal data field.

A *TDBGrid* control displays ADT and array field data differently, depending on the value of the dataset's *ObjectView* property. When *ObjectView* is false, each child field appears in a single column. When *ObjectView* is true, an ADT or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

Working with ADT fields

ADTs are user-defined types created on the server, and are similar to the record type (Delphi) or structures (C++). An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called *CityEdit*, and use the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

Using persistent field components

The easiest way to access ADT field values is to use persistent field components. For the ADT structure above, the following persistent fields can be added to the *Customer* table using the Fields editor:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

Given these persistent fields, you can simply access the child fields of an ADT field by name:

```
CityEdit.Text := CustomerAddrCity.AsString;
```

```
CityEdit->Text = CustomerAddrCity->AsString;
```

Although persistent fields are the easiest way to access ADT child fields, it is not possible to use them if the structure of the dataset is not known at design time. When accessing ADT child fields without using persistent fields, you must set the dataset's *ObjectView* property to true.

Using the dataset's FieldByName method

You can access the children of an ADT field using the dataset's *FieldByName* method by qualifying the name of the child field with the ADT field's name:

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

```
CityEdit->Text = Customer->FieldByName("Address.City")->AsString;
```

Using the dateset's FieldValues property

You can also use qualified field names with a dataset's *FieldValues* property:

```
CityEdit.Text := Customer['Address.City'];
```

```
CityEdit->Text = Customer->FieldValues["Address.City"];
```

Note in Delphi that you can omit the property name (FieldValues) because FieldValues is the dataset's default property.

Note Unlike other runtime methods for accessing ADT child field values, the *FieldValues* property works even if the dataset's *ObjectView* property is false.

Using the ADT field's FieldValues property

You can access the value of a child field with the TADTField's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert fields of any type. The index parameter is an integer value that specifies the offset of the field.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->FieldValues[1];
```

In Delphi, because FieldValues is the default property of *TADTField*, the property name (FieldValues) can be omitted. Thus, the following statement is equivalent to the Delphi example preceding:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

Using the ADT field's Fields property

Each ADT field has a *Fields* property that is analogous to the *Fields* property of a dataset. Like the *Fields* property of a dataset, you can use it to access child fields by position:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->Fields->
Fields[1]->AsString;
```

or by name:

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->
Fields->FieldByName("City")->AsString;
```

Working with array fields

Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The *SparseArrays* property of *TDataSet* determines whether a unique *TField* object is created for each element of the array field.

There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's *ObjectView* property must be set to true before you can access the elements of an array field.

Using persistent fields

You can map persistent fields to the individual array elements in an array field. For example, consider an array field *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the *Customer* table component represent the *TelNos_Array* field and its six elements:

Delphi example

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

C++ example

```
CustomerTELNOS_ARRAY: TArrayField;
CustomerTELNOS_ARRAY0: TStringField;
CustomerTELNOS_ARRAY1: TStringField;
CustomerTELNOS_ARRAY2: TStringField;
CustomerTELNOS_ARRAY3: TStringField;
CustomerTELNOS_ARRAY4: TStringField;
CustomerTELNOS_ARRAY5: TStringField;
```

Given these persistent fields, the following code uses a persistent field to assign an array element value to an edit box named *TelEdit*.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

```
TelEdit->Text = CustomerTELNOS_ARRAY0->AsString;
```

Using the array field's FieldValues property

You can access the value of a child field with the array field's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert child fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

```
TelEdit->Text = ((TArrayField*)Customer->FieldByName("TelNos_Array"))->FieldValues[1];
```

In Delphi, because *FieldValues* is the default property of *TArrayField*, this can also be written:

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

Using the array field's Fields property

TArrayField has a *Fields* property that you can use to access individual sub-fields. This is illustrated below, where an array field (*OrderDates*) is used to populate a list box with all non-null array elements:

Delphi example

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

C++ example

```
for (int i = 0; i < OrderDates->Size; ++i)
  if (!OrderDates->Fields->Fields[i]->IsNull)
    OrderDateListBox->Items->Add(OrderDates->Fields->Fields[i]->AsString);
```

Working with dataset fields

Dataset fields provide access to data stored in a nested dataset. The *NestedDataSet* property references the nested dataset. The data in the nested dataset is then accessed through the fields of the nested dataset.

Displaying dataset fields

TDBGrid controls enable the display of data stored in data set fields. In a *TDBGrid* control, a dataset field is indicated in each cell of a dataset column with a "(DataSet)", and at runtime an ellipsis button also exists to the right. Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a dataset field, the following code displays the dataset associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

```
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested dataset is another dataset, you use another dataset component to access its data. The type of dataset you use is determined by the parent dataset (the one with the dataset field.) For example, a dataset field in a *TClientDataSet* object must use another *TClientDataSet* to represent its value.

To access the data in a dataset field,

1. Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the values in that dataset field.
3. Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the nested dataset field for the current record has a value, the detail dataset component will contain records with the nested data; otherwise, the detail dataset will be empty.

Working with reference fields

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is returned in a nested dataset, but can also be accessed via the *Fields* property on the *TReferenceField*.

Displaying reference fields

In a *TDBGrid* control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

```
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

Accessing data in a reference field

You can access the data in a reference field in the same way you access a nested dataset:

1. Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the value of that dataset field.
3. Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the reference is assigned, the reference dataset will contain a single record with the referenced data. If the reference is null, the reference dataset will be empty.

You can also use the reference field's Fields property to access the data in a reference field. For example, the following lines assign data from the reference field

CustomerRefCity to an edit box called *CityEdit* (in the Delphi example, to the two lines are equivalent):

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

```
CityEdit->Text = CustomerADDRESS_REF->NestedDataSet->Fields->Fields[1]->AsString;
```

Chapter

24

Using unidirectional datasets

Unidirectional datasets provide the mechanism by which an application reads data from an SQL database table. They are designed for quick lightweight access to database information, with minimal overhead. Unidirectional datasets send an SQL command to the database server, and if the command returns a set of records, obtain a unidirectional cursor for accessing those records. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets. Many of the capabilities introduced by *TDataSet* are either unimplemented in unidirectional datasets, or cause them to raise exceptions. For example:

- The only supported navigation methods are the *First* and *Next* methods. Most others raise exceptions. Some, such as the methods involved in bookmark support, simply do nothing.
- There is no built-in support for editing because editing requires a buffer to hold the edits. The *CanModify* property is always false, so attempts to put the dataset into edit mode always fail. You can, however, use them to update data using an SQL UPDATE command or provide conventional editing support by connecting them to a client dataset (as described in "Using a client dataset to buffer records" on page 19-9).
- There is no support for filters, because they need to work with multiple records, which requires buffering. If you try to filter a unidirectional dataset, it raises an exception. Instead, all limits on what data appears must be imposed using the SQL command that defines the data for the dataset.
- There is no support for lookup fields, which require buffering to hold multiple records containing lookup values. If you define a lookup field on a unidirectional dataset, it does not work properly.

Despite these limitations, unidirectional datasets are a powerful way to access data. They are fast, and very simple to use and deploy.

Types of unidirectional datasets

The *dbExpress* page of the component palette contains four types of unidirectional dataset: *TSQLDataSet*, *TSQLQuery*, *TSQLTable*, and *TSQLStoredProc*.

TSQLDataSet is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications.

You can use *TSQLQuery* for almost everything you can accomplish with *TSQLDataSet*. *TSQLQuery* differs primarily in that it can't be used for stored procedures. (Many servers support an extension to SQL that lets you create queries to execute stored procedures, but there is no standard syntax for this) *TSQLQuery* is intended for compatibility with Windows applications, so that it is easier to port applications that use query components to Linux.

TSQLTable is a special-purpose dataset when you want to represent all of the fields and all of the records in a single database table. *TSQLTable* is intended for compatibility with Windows applications so that it is easier to port applications that use table components to Linux.

TSQLStoredProc is a special-purpose dataset for executing a stored procedure. You can also use *TSQLDataSet* to execute stored procedures: *TSQLStoredProc* is intended for compatibility with Windows applications so that it is easier to port applications that use stored procedure components to Linux.

Connecting to the Server

The first step when working with a unidirectional dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the Object Inspector can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the Object Inspector can list what stored procedures are available on the server.

The connection to a database server is represented by a separate *TSQLConnection* component. *TSQLConnection* identifies the database server and several connection parameters (including which database to use on the server, the host name of the machine running the server, the username, password, and so on). For information about setting up a connection using *TSQLConnection*, see Chapter 21, "Connecting to databases."

To connect a unidirectional dataset, you must specify the *TSQLConnection* that forms the connection. Do this using the *SQLConnection* property. At design time, you can choose the SQL connection component from a drop-down list in the Object Inspector. If you make this assignment at runtime, be sure that the connection is active:

D

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

```
SQLDataSet1->SQLConnection = SQLConnection1;
SQLConnection1->Connected = true;
```

Typically, all unidirectional datasets in an application share the same connection component, unless you are working with data from multiple database servers or with a database server such as MySQL that does not support multiple statements.

Specifying what data to display

There are a number of ways to specify what data a unidirectional dataset represents. Which method you choose depends on the type of unidirectional dataset you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure.

When you work with a *TSQLDataSet* component, use the *CommandType* property to indicate where the dataset gets its data. *CommandType* can take any of the following values:

- *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting set of records.
- *ctTable*: When *CommandType* is *ctTable*, *TSQLDataSet* retrieves all of the records from a specified table.
- *ctStoredProc*: When *CommandType* is *ctStoredProc*, *TSQLDataSet* executes a stored procedure. If the stored procedure returns a cursor, the dataset contains the returned records.

Note You can also populate the unidirectional dataset with metadata about what is available on the server. For information on how to do this, see "Accessing schema information" on page 24-17.

Representing the results of a query

Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. They need not return a set of records; SQL defines queries such as UPDATE queries that perform actions on the server. Queries that do not return records are discussed in greater detail in "Executing commands that do not return records" on page 24-12.

Most queries that return records are SELECT commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query.

Specifying a query using TSQLDataSet

When using *TSQLDataSet*, assign the text of the query statement to the *CommandText* property:

```
SQLDataSet1.CommandText := 'SELECT CustName, Address FROM Customer';
```

```
SQLDataSet1->CommandText = "SELECT CustName, Address FROM Customer";
```

At design time, you can type the query directly into the Object Inspector, or, if the SQL dataset already has an active connection to the database, you can click the ellipsis button by the *CommandText* property to display the Command Text editor. The Command Text editor lists the available tables, and the fields in those tables, to make it easier to compose your queries.

Specifying a query using TSQLQuery

When using *TSQLQuery*, assign the query to the *SQL* property instead. Unlike the *CommandText* property of *TSQLDataSet*, which is a string (Delphi) or an AnsiString (C++), the *SQL* property is a *TStrings* object. Each separate string in this *TStrings* object is a separate line of the query. Using multiple lines does not affect the way the query executes on the server, but can make it easier to assemble a query from multiple sources:

Delphi example

```
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add('SELECT ' + Edit1.Text + ' FROM ' + Edit2.Text);
if Length(Edit3.Text) <> 0 then
  SQLQuery1.SQL.Add('ORDER BY ' + Edit3.Text)
```

C++ example

```
SQLQuery1->SQL->Clear();
SQLQuery1->SQL->Add("SELECT " + Edit1->Text + " FROM " + Edit2.Text);
if (!Edit3->Text->IsEmpty())
  SQLQuery1->SQL->Add("ORDER BY " + Edit3.Text);
```

At design time, use the String List editor to specify the query. Click the ellipsis button by the *SQL* property in the Object Inspector to display the String List editor.

One advantage of using *TSQLQuery* is that, because the *SQL* property is a *TStrings* object, you can load the text of the query from a file by calling the *TStrings LoadFromFile* method:

```
SQLQuery1.SQL.LoadFromFile('/usr/queries/custquery.sql');
```

```
SQLQuery1->SQL->LoadFromFile("//usr//queries//custquery.sql");
```

Using parameters in queries

The previous topic showed how to build a query dynamically at runtime. However, you can accomplish the same type of thing for queries written at design time by using parameters.

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. For example, in the following SELECT statement, a parameter is used to specify a selection criterion:

```
SELECT CustNo, OrderNo, SaleDate FROM Orders
WHERE CustNo = :CustNumber
```

In this SQL statement, *:CustNumber* is a placeholder for the actual value supplied to the statement at runtime by your application. Note that the name, *:CustNumber*, begins with a colon. Parameter names must start with a colon (:). You can also include unnamed parameters by adding a question mark (?) to your query. Unnamed parameters are identified by position, because they do not have unique names.

Before the dataset can execute the query, you must supply values for any parameters in the query text. The dataset uses its *Params* property to store these values. *Params* is a collection of *TParam* objects, where each *TParam* object represents a single parameter. When you specify the text for the query (using the *CommandText* or *SQL* property), the dataset generates this set of *TParam* objects, and initializes any of their properties that it can deduce from the query.

Note You can suppress the automatic generation of *TParam* objects in response to changing the query text by setting the *ParamCheck* property to false. See "Creating and modifying server metadata" on page 24-13 for an example where this is useful.

Note You do not need to explicitly assign parameter values if the dataset uses a datasource to obtain parameter values from another dataset. This process is described in "Setting up master/detail relationships" on page 24-15.

Setting up parameters at design time

At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* property in the Object Inspector. If the SQL statement does not contain any parameters, no *TParam* objects are listed in the collection editor.

Note The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for dataset parameters. The only way to add or delete parameters is in the SQL statement itself.

For each parameter, select it in the parameter collection editor. Then use the Object Inspector to modify its properties:

- The *DataType* property lists the data type for the parameter's value. This value may be correctly initialized, if the dataset could deduce the value type from the query. If the dataset could not deduce the type, *DataType* is *ftUnknown*, and you must change it to indicate the type of the parameter value.
- The *ParamType* property lists the type of the selected parameter. For queries, this is always initialized to *ptInput*, because queries can only contain input parameters.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

Setting parameters at runtime

To create parameters at runtime, you can use the:

- *ParamByName* method to assign values to a parameter based on its name.
- *Params* (Delphi) or *Params::Items* (C++) property to assign values to a parameter based on the parameter's ordinal position in the SQL statement.
- *Params.ParamValues* (Delphi) or *Params::ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set.

The following code uses *ParamByName* to assign the text of an edit box to the :Capital parameter:

```
SQLDataSet1.ParamByName('Capital').AsString := Edit1.Text;
```

```
SQLDataSet1->ParamByName("Capital")->AsString = Edit1->Text;
```

The same code can be rewritten using the *Params* property, using an index of 0 (assuming the :Capital parameter is the first parameter in the SQL statement):

```
SQLDataSet1.Params[0].AsString := Edit1.Text;
```

```
SQLDataSet1->Params->Items[0]->AsString = Edit1->Text;
```

The command line below sets three parameters at once, using the *Params ParamValues* property:

```
SQLDataSet1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

```
SQLDataSet1->Params->ParamValues["Name;Capital;Continent"] =
  VarArrayOf(OPENARRAY(Variant, (Edit1->Text, Edit2->Text, Edit3->Text)));
```

Note that *ParamValues* uses Variants, avoiding the need to cast values.

Representing the records in a table

When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself.

Note If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This results in slightly slower performance on the server. The wildcard (*) in automatically-generated queries is more robust to changes in the fields on the server.

Representing a table using TSQLDataSet

To make *TSQLDataSet* generate a query to fetch all fields and all records of a single database table, set the *CommandType* property to *ctTable*.

When *CommandType* is *ctTable*, *TSQLDataSet* generates a query based on the values of two properties:

- *CommandText* specifies the name of the database table that the *TSQLDataSet* object should represent.
- *SortFieldNames* lists the names of any fields to use to sort the data, in the order of significance.

For example, if you specify the following:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

```
SQLDataSet1->CommandType = ctTable;
SQLDataSet1->CommandText = "Employee";
SQLDataSet1->SortFieldNames = "HireDate,Salary"
```

TSQLDataSet generates the following query, which lists all the records in the Employee table, sorted by HireDate and, within HireDate, by Salary:

```
select * from Employee order by HireDate, Salary
```

Representing a table using TSQLTable

When using *TSQLTable*, specify the table you want using the *TableName* property.

To specify the order of fields in the dataset, you must specify an index. There are two ways to do this:

- Set the *IndexName* property to the name of an index defined on the server that imposes the order you want.
- Set the *IndexFieldNames* property to a semicolon-delimited list of field names on which to sort. *IndexFieldNames* works like the *SortFieldNames* property of *TSQLDataSet*, except that it uses a semicolon instead of a comma as a delimiter.

Representing the results of a stored procedure

Stored procedures are sets of SQL statements that are named and stored on an SQL server. They typically handle frequently-repeated database-related tasks. They are especially useful for operations that act upon large numbers of rows in database tables or that use aggregate or mathematical functions. Using stored procedures typically improves the performance of a database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing network traffic by moving processing to the server.

Stored procedures may or may not return data. Those that return data may return it as a cursor (similar to the results of a SELECT query), as multiple cursors (effectively returning multiple datasets), or they may return data in output parameters. These differences depend in part on the server: Some servers do not allow stored procedures to return data, or only allow output parameters. Some servers do not support stored procedures at all. See your server documentation to determine what is available.

How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using.

Specifying a stored procedure using TSQLDataSet

When using *TSQLDataSet*, to specify a stored procedure:

- Set the *CommandType* property to *ctStoredProc*.
- Specify the name of the stored procedure as the value of the *CommandText* property:

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

```
SQLDataSet1->CommandType = ctStoredProc;
SQLDataSet1->CommandText = "MyStoredProcName";
```

Specifying a stored procedure using TSQLStoredProc

When using *TSQLStoredProc*, you need only specify the name of the stored procedure as the value of the *StoredProcName* property.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

```
SQLStoredProc1->StoredProcName = "MyStoredProcName";
```

Working with stored procedure parameters

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, used to pass values to a stored procedure for processing.
- *Output parameters*, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- A *result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For any server, individual stored procedures may or may not use input parameters. On the other hand, some uses of parameters are server-specific. For example, the InterBase implementation of stored procedures never returns a result parameter.

Access to stored procedure parameters is provided by *TParam* objects in the *Params* property. As with query parameters, *TSQLDataSet* and *TSQLStoredProc* automatically generate the *TParam* objects for each parameter when you set the *CommandText* (*TSQLDataSet*) or *StoredProcName* (*TSQLStoredProc*) property.

Setting up parameters at design time

As with query parameters, you can specify stored procedure parameter values at design time using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* property in the Object Inspector.

Important You can assign values to input parameters by selecting them in the parameter collection editor and using the Object Inspector to set the *Value* property. However, do not change the names or data types for input parameters reported by the server. Otherwise, when you execute the stored procedure an exception is raised.

Some servers do not report parameter names or data types. In these cases, you must set up the parameters manually using the parameter collection editor. Right click and choose Add to add parameters. For each parameter you add,

- Assign the name (defined by the procedure on the server) as the value of the *Name* property.
- Indicate whether it is an input, output, input/output, or result parameter by setting the *ParamType* property.
- Indicate the data type of the parameter's value by setting the *DataType* property.

Some servers provide parameter names, but do not return all parameter information. Check the *DataType* and *ParamType* properties for each parameter and fix any that are set to *ftUnknown* or *ptUnknown*.

Note You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

Using parameters at runtime

At runtime, you can access individual parameter objects to assign values to input parameters and to retrieve values from output parameters. As with query parameters, you can use the *ParamByName* method to access individual parameters based on their names. For example, the following code sets the value of an input/output parameter, executes the stored procedure, and retrieves the returned value:

Delphi example

```
with SQLDataSet1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

C++ example

```
SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger = 103;
SQLDataSet1->ExecProc();
int Result = SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger;
```

Because some servers do not report parameter names or data types, you may need to set up parameters in code if you do not specify the name of the stored procedure until runtime. To check whether this is necessary, try specifying a stored procedure on the target database at design time, and use the parameter collection editor to check whether the dataset automatically sets up parameters.

The following example illustrates how to set up parameters at runtime if they are not supplied automatically:

Delphi example

```
var
  P1, P2: TParam;
begin
  with SQLDataSet1 do
  begin
    CommandType := ctStoredProc;
    CommandText := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := ‘EMP_NO’;
      Params[0].DataType := ftSmallint;
      Params[1].Name := ‘PROJ_ID’;
      Params[1].DataType := ftString;
      ParamByname(‘EMP_NO’).AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname(‘PROJ_ID’).AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
end;
```

C++ example

```
SQLDataSet1->CommandType = ctStoredProc;
SQLDataSet1->CommandText = "GET_EMP_PROJ";
SQLDataSet1->Params->Clear();
TParam *P1 = new TParam(SQLDataSet1->Params, ptInput);
TParam *P2 = new TParam(SQLDataSet1->Params, ptOutput);
try
{
  SQLDataSet1->Params->Items[0]->Name = "EMP_NO";
  SQLDataSet1->Params->Items[0]->DataType = ftSmallint;
  SQLDataSet1->Params->Items[1]->Name = "PROJ_ID";
  SQLDataSet1->Params->Items[1]->DataType = ftString;
  SQLDataSet1->ParamByName("EMP_NO")->AsSmallInt = 52;
  SQLDataSet1->ExecProc();
  Edit1->Text = SQLDataSet1->ParamByName("PROJ_ID")->AsString;
}
__finally
{
  delete P1;
  delete P2;
}
```

Fetching the data

Once you have specified the source of the data, you must fetch the data before your application can access it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records.

As with any dataset, there are two ways to fetch the data for a unidirectional dataset:

- Set the *Active* property to true, either at design time in the Object Inspector, or in code at runtime:

```
  CustQuery.Active := True;
```

```
CustQuery->Active = true;
```

- Call the *Open* method at runtime,

```
  CustQuery.Open;
```

```
CustQuery->Open();
```

Use the *Active* property or the *Open* method with any unidirectional dataset that obtains records from the server. It does not matter whether these records come from a SELECT query (including automatically-generated queries when the *CommandType* is *ctTable*) or a stored procedure.

Preparing the dataset

Before a query or stored procedure can execute on the server, it must first be "prepared". Preparing the dataset means that *dbExpress* and the server allocate resources for the statement and its parameters. If *CommandType* is *ctTable*, this is when the dataset generates its SELECT query. Any parameters that are not bound by the server are folded into a query at this point.

Unidirectional datasets are automatically prepared when you set *Active* to true or call the *Open* method. When you close the dataset, the resources allocated for executing the statement are freed. If you intend to execute the query or stored procedure more than once, you can improve performance by explicitly preparing the dataset before you open it the first time. To explicitly prepare a dataset, set its *Prepared* property to true.

```
CustQuery.Prepared := True;
```

```
CustQuery->Prepared = true;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to false.

Set the *Prepared* property to false if you want to ensure that the dataset is re-prepared before it executes (for example, if you change a parameter value or the *SortFieldNames* property).

Fetching multiple datasets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. In order to access the other sets of records, call the *NextRecordSet* method:

Delphi example

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

C++ example

```
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet();
```

NextRecordSet returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. That is, the first time you call *NextRecordSet*, it returns a dataset for the second set of records. Calling *NextRecordSet* again returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, *NextRecordSet* returns nil (Delphi) or NULL (C++).

Executing commands that do not return records

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution.

Note If the command does not return any records, you do not need to use a unidirectional dataset at all, because there is no need for the dataset methods that provide access to a set of records. The SQL connection component that connects to the database server can be used directly to execute a command on the server. See “Sending commands to the server” on page 21-13 for details.

Specifying the command to execute

With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is:

When using *TSQLDataSet*, use the *CommandType* and *CommandText* properties to specify the command:

- If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server.
- If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute.

When using *TSQLQuery*, use the *SQL* property to specify the SQL statement to pass to the server.

When using *TSQLStoredProc*, use the *StoredProcName* property to specify the name of the stored procedure to execute.

Just as you specify the command in the same way as when you are retrieving records, you work with query parameters or stored procedure parameters the same way as with queries and stored procedures that return records. See "Using parameters in queries" on page 24-4 and "Working with stored procedure parameters" on page 24-8 for details.

Executing the command

To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use

- The *ExecSQL* method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
FixTicket.ExecSQL;
```

```
FixTicket->CommandText = "DELETE FROM TrafficViolations WHERE (TicketID = 1099)";
FixTicket->ExecSQL();
```

- The *ExecProc* method if the dataset is an instance of *TSQLStoredProc*.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
SQLStoredProc1.ExecProc;
```

```
SQLStoredProc1->StoredProcName = "MyCommandWithNoResults";
SQLStoredProc1->ExecProc();
```

Tip If you are executing the query or stored procedure multiple times, it is a good idea to set the *Prepared* property to true.

Creating and modifying server metadata

Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures.

If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset and let it handle all the generation of all SQL commands concerned with editing (see "Using a client dataset to buffer records" on page 19-9). In fact, this is the recommended approach because data-aware controls are designed to perform edits through a client dataset (or custom dataset that enables editing).

The only way your application can create or modify metadata on the server, however, is to send a command. Not all database drivers support the same SQL syntax. It is beyond the scope of this topic to describe the SQL syntax supported by each database type and the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that system.

In general, use the CREATE TABLE statement to create tables in a database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE.

For each of the CREATE statements, there is a corresponding DROP statement to delete the metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

For example, the following statement creates a stored procedure called GET_EMP_PROJ on an InterBase database:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The following code uses a *TSQLDataSet* to create this stored procedure. Note the use of the *ParamCheck* property to prevent the dataset from confusing the parameters in the stored procedure definition (:EMP_NO and :PROJ_ID) with a parameter of the query that creates the stored procedure.

D **Delphi example**

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
      'RETURNS (PROJ_ID CHAR(5)) AS ' +
      'BEGIN ' +
        'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
        'WHERE EMP_NO = :EMP_NO ' +
        'INTO :PROJ_ID ' +
          'DO SUSPEND; ' +
       END';
  ExecSQL;
end;
```

C++ example

```
SQLDataSet1->ParamCheck = false;
SQLDataSet1->CommandType = ctQuery;
SQLDataSet1->CommandText = "CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) RETURNS (PROJ_ID
CHAR(5)) AS BEGIN FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT WHERE EMP_NO = :EMP_NO INTO
:PROJ_ID DO SUSPEND; END";
SQLDataSet1->ExecSQL();
```

Setting up master/detail relationships

There are two ways to set up a master/detail relationship that uses a unidirectional dataset as the detail set. Which method you use depends on the type of unidirectional dataset you are using. Once you have set up such a relationship, the unidirectional dataset (the "many" in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the "one" in the one-to-many relationship).

Setting up master/detail relationships with TSQLDataSet or TSQLQuery

To set up a master/detail relationship where the detail set is an instance of *TSQLDataSet* or *TSQLQuery*, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set's parameters every time the master record changes. Although you could write code to do this using an event handler, *TSQLDataSet* and *TSQLQuery* provide an easier mechanism using the *DataSource* property.

If parameter values for a parameterized query are not bound at design time or specified at runtime, *TSQLDataSet* and *TSQLQuery* attempt to supply values for them based on the *DataSource* property. *DataSource* identifies a different dataset that is searched for field names that match the names of unbound parameters. This search dataset can be any type of dataset, it need not be another unidirectional dataset. The search dataset must be created and populated before you create the detail dataset that uses it. If matches are found in the search dataset, the detail dataset binds the parameter values to the values of the fields in the current record pointed to by the data source.

To illustrate how this works, consider two tables: a customer table and an orders table. For every customer, the orders table contains a set of orders that the customer made. The Customer table includes an ID field that specifies a unique customer ID. The orders table includes a CustID field that specifies the ID of the customer who made an order.

The first step is to set up the Customer dataset:

1. Add a *TSQLDataSet* component to your application. Set its *CommandType* property to *ctTable* and its *CommandText* property to the name of the Customer table.
2. Add a *TDataSource* component named *CustomerSource*. Set its *DataSet* property to the dataset added in step 1. This data source now represents the Customer dataset.
3. Add another *TSQLDataSet* component and set its *CommandType* property to *ctQuery*. Set its *CommandText* property to

   ```
   SELECT CustID, OrderNo, SaleDate
   FROM Orders
   WHERE CustID = :ID
   ```

   Note that the name of the parameter is the same as the name of the field in the master (Customer) table.
4. Set the detail dataset's *DataSource* property to *CustomerSource*. Setting this property makes the detail dataset a linked query.

At runtime the *:ID* parameter in the SQL statement for the detail dataset is not assigned a value, so the dataset tries to match the parameter by name against a column in the dataset identified by *CustomersSource*. *CustomersSource* gets its data from the master dataset, which, in turn, derives its data from the Customer table. Because the Customer table contains a column called "ID," the value from the *ID* field in the current record of the master dataset is assigned to the *:ID* parameter for the detail dataset's SQL statement. The datasets are linked in a master-detail relationship. Each time the current record changes in the Customers dataset, the detail dataset's SELECT statement executes to retrieve all orders based on the current customer id.

Setting up master/detail relationships with TSQLTable

To set up a master/detail relationship where the detail set is an instance of *TSQLTable*, use the *MasterSource* and *MasterFields* properties.

The *MasterSource* property specifies a data source bound to the master table (like the *DataSource* property you use when the detail is a query). The *MasterFields* property lists the fields in the master table that correspond to the fields in the *TSQLTable* object's index.

Before you set the *MasterFields* property, first specify the index that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you have specified the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the indexed fields in the (detail) SQL table. To link datasets based on multiple column names, use a semicolon delimited list:

```
SQLTable1.MasterFields := 'OrderNo;ItemNo';
```

```
SQLTable1->MasterFields = "OrderNo;ItemNo";
```

Tip If you double click on the *MasterFields* property in the Object Inspector after you have assigned a *MasterSource*, the Field Link editor appears to help you select link fields in the master dataset and in *TSQLTable*.

Accessing schema information

You can populate a unidirectional dataset with information about what is available on the server instead of the results of a query or stored procedure. This information, called metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

To fetch metadata from the database server, you must first indicate what data you want to see, using the *SetSchemaInfo* method. *SetSchemaInfo* takes four parameters:

- The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of Oracle packages (*stPackages*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses a different set of fields to describe the items in the list. For details on the structures of these datasets, see "The structure of metadata datasets" on page 24-18.
- If you are fetching information about fields, indexes, or stored procedure parameters, the name of the table or stored procedure to which they apply. If you are fetching any other type of schema information, this parameter is nil (Delphi) or NULL (C++).
- A pattern that must be matched for every name returned. This pattern is an SQL pattern such as 'Cust%', which uses the wildcards '%' (to match a string of arbitrary characters of any length) and '_' (to match a single arbitrary character). To use a literal percent or underscore in a pattern, the character is doubled (%% or __). If you do not want to use a pattern, this parameter can be nil (Delphi) or NULL (C++).
- If you are fetching information from an Oracle server about stored procedures or the parameters of a specific stored procedure, the fourth parameter optionally gives the name of the Oracle package to which the stored procedure or procedures belong. If the server is not an Oracle server or you have no need to specify the package that contains a stored procedure, you can set this fourth parameter to an empty string.

Note If you are fetching schema information about tables (*stTables*), the resulting schema information can describe ordinary tables, system tables, views, and/or synonyms, depending on the value of the SQL connection's *TableScope* property.

The following call requests a table listing all system tables (server tables that contain metadata):

```
SQLDataSet1.SetSchemaInfo(stSysTable, '', '');
```

```
SQLDataSet1->SetSchemaInfo(stSysTable, "", "");
```

When you open the dataset after this call to *SetSchemaInfo*, the resulting dataset has a record for each table, with columns giving the table name, type, schema name, and so on. If the server does not use system tables to store metadata (for example MySQL), when you open the dataset it contains no records.

The previous example used only the first parameter. Suppose, Instead, you want to obtain a list of input parameters for a stored procedure named 'MyProc'. Suppose, further, that the person who wrote that stored procedure named all parameters using a prefix to indicate whether they were input or output parameters ('inName', 'outValue' and so on). You would call *SetSchemaInfo* as follows:

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

```
SQLDataSet1->SetSchemaInfo(stProcedureParams, "MyProc", "in%");
```

The resulting dataset is a table of input parameters with columns to describe the properties of each parameter.

Note You can also fetch metadata from the database server into a list object rather than using a dataset. This alternate approach, which uses *TSQLConnection* rather than a dataset, provides less information and has no support for pattern matching strings. For details, see "Accessing server metadata" on page 21-10.

Fetching data after using the dataset for metadata

There are two ways to return to executing queries or stored procedures with the dataset after a call to *SetSchemaInfo*:

- Change the *CommandText* property, specifying the query, table, or stored procedure from which you want to fetch data.
- Call *SetSchemaInfo*, setting the first parameter to *stNoSchema*. In this case, the dataset reverts to fetching the data specified by the current value of *CommandText*.

The structure of metadata datasets

For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

Information about tables

When you request information about tables (*stTables* or *stSysTables*), the resulting dataset includes a record for each table. It has the following columns:

Table 24.1 Columns in tables of metadata listing tables

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table. This is the same as the *Database* parameter on an SQL connection component.

Table 24.1 Columns in tables of metadata listing tables (continued)

Column Name	Field type	Contents
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the table.
TABLE_NAME	ftString	The name of the table. This field determines the sort order of the dataset.
TABLE_TYPE	ftInteger	Identifies the type of table. It is a sum of one or more of the following values: 1: Table 2: View 4: System table 8: Synonym 16: Temporary table 32: Local table.

Information about stored procedures

When you request information about stored procedures (*stProcedures*), the resulting dataset includes a record for each stored procedure. It has following columns:

Table 24.2 Columns in tables of metadata listing stored procedures

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure. This field determines the sort order of the dataset.
PROC_TYPE	ftInteger	Identifies the type of stored procedure. It is a sum of one or more of the following values: 1: Procedure (in C++ no return value) 2: Function (in C++ returns a value) 4: Package 8: System procedure
IN_PARAMS	ftSmallint	The number of input parameters
OUT_PARAMS	ftSmallint	The number of output parameters.

Information about fields

When you request information about the fields in a specified table (*stColumns*), the resulting dataset includes a record for each field. It includes the following columns:

Table 24.3 Columns in tables of metadata listing fields

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table whose fields you listing. This is the same as the *Database* parameter on an SQL connection component.

Table 24.3 Columns in tables of metadata listing fields (continued)

Column Name	Field type	Contents
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the field.
TABLE_NAME	ftString	The name of the table that contains the fields.
COLUMN_NAME	ftString	The name of the field. This value determines the sort order of the dataset.
COLUMN_POSITION	ftSmallint	The position of the column in its table.
COLUMN_TYPE	ftInteger	Identifies the type of value in the field. It is a sum of one or more of the following: 1: Row ID 2: Row Version 4: Auto increment field 8: Field with a default value
COLUMN_DATATYPE	ftSmallint	The datatype of the column. This is one of the logical field type constants defined in sqllinks.pas (Delphi) or sqllinks.h (C++).
COLUMN_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in COLUMN_DATATYPE and COLUMN_SUBTYPE, but in a form used in some DDL statements.
COLUMN_SUBTYPE	ftSmallint	A subtype for the column's datatype. This is one of the logical subtype constants defined in sqllinks.pas (Delphi) or sqllinks.h (C++).
COLUMN_PRECISION	ftInteger	The size of the field type (number of characters in a string, bytes in a bytes field, significant digits in a BCD value, members of an ADT field, and so on)
COLUMN_SCALE	ftSmallint	The number of digits to the right of the decimal on BCD values, or descendants on ADT and array fields.
COLUMN_LENGTH	ftInteger	The number of bytes required to store field values
COLUMN_NULLABLE	ftSmallint	A Boolean that indicates whether the field can be left blank. (0 means the field requires a value)

Information about indexes

When you request information about the indexes on a table (stIndexes), the resulting dataset includes a record for each field in each record. (Multi-record indexes are described using multiple records) The dataset has the following columns:

Table 24.4 Columns in tables of metadata listing indexes

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the index. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the index.
TABLE_NAME	ftString	The name of the table for which the index is defined.

Table 24.4 Columns in tables of metadata listing indexes (continued)

Column Name	Field type	Contents
INDEX_NAME	ftString	The name of the index. This field determines the sort order of the dataset.
PKEY_NAME	ftString	Indicates the name of the primary key.
COLUMN_NAME	ftString	The name of the field (column) in the index.
COLUMN_POSITION	ftSmallint	The position of this field in the index.
INDEX_TYPE	ftSmallint	Identifies the type of index. It is a sum of one or more of the following values: 1: Non-unique 2: Unique 4: Primary key
SORT_ORDER	ftString	Indicates that the index is ascending (a) or descending (d) on this field.
FILTER	ftString	Describes a filter condition that limits the indexed records.

Information about stored procedure parameters

When you request information about the parameters of a stored procedure (*stProcedureParams*), the resulting dataset includes a record for each parameter. It has the following columns:

Table 24.5 Columns in tables of metadata listing parameters

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure that contains the parameter.
PARAM_NAME	ftString	The name of the parameter. This field determines the sort order of the dataset.
PARAM_TYPE	ftSmallint	Identifies the type of parameter. This is the same as a *TParam* object's *ParamType* property.
PARAM_DATATYPE	ftSmallint	The datatype of the parameter. This is one of the logical field type constants defined in sqllinks.pas (Delphi) or sqllinks.h (C++).
PARAM_SUBTYPE	ftSmallint	A subtype for the parameter's datatype. This is one of the logical subtype constants defined in sqllinks.pas (Delphi) or sqllinks.h (C++).
PARAM_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in PARAM_DATATYPE and PARAM_SUBTYPE, but in a form used in some DDL statements.
PARAM_PRECISION	ftInteger	The maximum number of digits in floating-point values or bytes (for strings and Bytes fields).

Table 24.5 Columns in tables of metadata listing parameters (continued)

Column Name	Field type	Contents
PARAM_SCALE	ftSmallint	The number of digits to the right of the decimal on floating-point values.
PARAM_LENGTH	ftInteger	The number of bytes required to store parameter values.
PARAM_NULLABLE	ftSmallint	A Boolean that indicates whether the parameter can be left blank. (0 means the parameter requires a value)

Information about Oracle packages

When you request information about the packages on an Oracle server (*stPackages*), the resulting dataset includes a record for each package. It has following columns:

Table 24.6 Columns in tables of metadata listing stored procedures

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the package. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the package.
OBJECT_NAME	ftString	The name of the package. This field determines the sort order of the dataset.

Chapter

25

Using client datasets

Client datasets provide all the data access, editing, navigation, data constraint, and filtering support introduced by *TDataSet*. They cache all data in memory and manipulate the in-memory cache. The support for manipulating the data they store in memory is provided by a shared object file, midas.so, which must be deployed with any application that uses client datasets.

The IDE provides two types of client datasets:

- *TClientDataSet*, which is designed to work without a specific type of database connectivity support (such as *dbExpress)*.
- *TSQLClientDataSet*, which uses *dbExpress* to fetch the data for the in-memory cache.

Both types of client dataset add to the properties and methods inherited from *TDataSet* to provide an expanded set of features for working with data. They differ primarily in how they obtain the data that is cached in memory.

Both types of client dataset support the following mechanisms for reading data and writing updates:

- Reading from and writing to a file accessed directly from the client dataset. When using only this mechanism, an application should use *TClientDataSet*, which has less overhead. However, when using the briefcase model, this mechanism is equally appropriate for both types of client dataset. Properties and methods supporting this mechanism are described in "Using a client dataset with file-based data" on page 25-40.
- Reading from another dataset. Client datasets provide a variety of mechanisms for copying data from other datasets. These are described in "Copying data from another dataset" on page 25-25.

In addition, *TClientDataSet* can also fetch data from a provider component. Provider components link *TClientDataSet* to other datasets. The provider forwards data from the source dataset to the client dataset, and sends edits from the client dataset back to the source dataset. Optionally, the provider generates an SQL command to apply updates back to the server, which the source dataset forwards to the server. Properties and methods for working with a provider are described in “Using a client dataset with a provider” on page 25-27.

TSQLClientDataSet can’t be linked to an external provider. Instead, it fetches data from a database server. The client dataset still caches updates in its in-memory cache, and includes a method to apply those cached updates back to the database server. Properties and methods for using *TSQLClientDataSet* to connect to a database server are described in “Using an SQL client dataset” on page 25-37.

Working with data using a client dataset

Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Chapter 20, “Using data controls”for information on how to display database information in data-aware controls.

As descendants of *TDataSet*, client datasets inherit the power and usefulness of the properties, methods, and events defined for all dataset components. For a complete introduction to this generic dataset behavior, see Chapter 22, “Understanding datasets.”

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for common database functions can involve additional capabilities or considerations.

Navigating data in client datasets

If an application uses standard data-aware controls, then a user can navigate through a client dataset’s records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see “Navigating datasets” on page 22-9.

Also inherited from *TDataSet* are the *Locate* and *Lookup* methods, which search for a particular record based on the values of specified fields. These methods are described in “Searching datasets” on page 22-16.

Client datasets implement the standard bookmark capabilities introduced in *TDataSet* for marking and navigating to specific records. For more information about bookmarking, see “Marking and returning to records” on page 22-14.

In addition to these methods introduced by *TDataSet*, client datasets introduce a number of additional navigation methods that take advantage of the way they store and index in-memory records.

For example, instead of using bookmarks, with the overhead of allocating and freeing the memory to store them, client datasets can position the cursor at any specific record using the *RecNo* property. While other datasets may use *RecNo* to determine the record number of the current record, client datasets can also set *RecNo* to a particular record number to make that record the current one.

However, the most powerful method introduced by *TClientDataSet* is the *Goto* and *Find* search methods that search for a record based on indexed fields. By explicitly using indexes that you define for the client dataset, client datasets can improve over the searching performance provided by the *Locate* and *Lookup* methods.

The following table summarizes the six related *Goto* and *Find* methods an application can use to search for a record:

Table 25.1 Index-based search methods

Method	Purpose
EditKey	Preserves the current contents of the search key buffer and puts the dataset into *dsSetKey* state so your application can modify existing search criteria prior to executing a search.
FindKey	Combines the *SetKey* and *GotoKey* methods in a single method.
FindNearest	Combines the *SetKey* and *GotoNearest* methods in a single method.
GotoKey	Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found.
GotoNearest	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
SetKey	Clears the search key buffer and puts the dataset into *dsSetKey* state so your application can specify new search criteria prior to executing a search.

GotoKey and *FindKey* are boolean functions that, if successful, move the cursor to a matching record and return true. If the search is unsuccessful, the cursor is not moved, and these functions return false.

GotoNearest and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

Specifying the index to use for searching

Before using the *Goto* and *Find* search methods, you must define the index, or key, that is used to speed the search. If the index has already been created for your client dataset, you must make that index current using the *IndexName* property. For example, if a client dataset has an index named “CityIndex” on which you want to search for a value, you must set the *IndexName* property to “CityIndex”:

```
ClientDataSet1.Close;
ClientDataSet1.IndexName := 'CityIndex';
ClientDataSet1.Open;
ClientDataSet1.SetKey;
ClientDataSet1['City'] := Edit1.Text;
ClientDataSet1.GotoNearest;
```

```
ClientDataSet1->Close();
ClientDataSet1->IndexName = “CityIndex”;
ClientDataSet1->Open();
ClientDataSet1->SetKey();
ClientDataSet1->FieldValues[“City”] = Edit1->Text;
ClientDataSet1->GotoNearest();
```

Instead of specifying an index name, you can list fields to use as a key in the *IndexFieldNames* property.

For more information on creating and using indexes in client datasets, see "Sorting and indexing" on page 25-18.

Executing a search with Goto methods

To execute a search using *Goto* methods, follow these general steps:

1. Specify the index to use for the search (as described above).
2. Open the client dataset.
3. Put the dataset in *dsSetKey* state with *SetKey*.
4. Specify the value(s) to search on in the *Fields* property. *Fields* is a *TFields* object, which maintains an indexed list of field components you can access by specifying ordinal numbers corresponding to columns. The first column number in a dataset is 0.
5. Search for and move to the first matching record found with: *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button's *OnClick* event, moves to the first record where the first field in the index has a value that exactly matches the text in an edit box:

Delphi example

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

C++ example

```
void __fastcall TSearchDemo::SearchExactClick(TObject *Sender)
{
  ClientDataSet1->SetKey();
  ClientDataSet1->Fields->Fields[0]->AsString = Edit1->Text;
  if (!ClientDataSet1->GotoKey())
    ShowMessage("Record not found");
}
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
ClientDataSet1.SetKey;
ClientDataSet1.Fields[0].AsString := 'Sm';
ClientDataSet1.GotoNearest;
```

```
ClientDataSet1->SetKey();
ClientDataSet1->Fields->Fields[0]->AsString = "Sm";
ClientDataSet1->GotoNearest();
```

If a record exists with “Sm” as the first two characters of the first indexed field’s value, the cursor is positioned on that record. Otherwise, the position of the cursor does not change and *GotoNearest* returns false.

Executing a search with Find methods

The Find methods do the same thing as the *Goto* methods, except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search. To execute a search using *Find* methods, follow these general steps:

1. Specify the index to use for the search (as described above).
2. Open the client dataset.
3. Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

Note *FindNearest* can only be used for string fields.

Specifying the current record after a successful search

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to true to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is false, meaning that successful searches position the cursor on the first matching record.

Searching on partial keys

If a client dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the client dataset’s current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1.

For client datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

Repeating or extending a search

Each time you call *SetKey* or *FindKey* it clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*. For example, suppose you have already executed a search based on the City field of the “CityIndex” index. Suppose further that “CityIndex” includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:

D **Delphi example**

```
ClientDataSet1.KeyFieldCount := 2;
ClientDataSet1.EditKey;
```

```
ClientDataSet1['Company'] := Edit2.Text;
ClientDataSet1.GotoNearest;
```

C++ example

```
ClientDataSet1->KeyFieldCount = 2;
ClientDataSet1->EditKey();
ClientDataSet1->FieldValues["Company"] = Variant(Edit2->Text);
ClientDataSet1->GotoNearest();
```

Limiting what records appear

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions.

Filters are introduced by *TDataSet* class, and so potentially apply to custom datasets as well as client datasets. (They are not implemented for unidirectional datasets.) For more information about using filters, see "Displaying and editing a subset of data using filters" on page 22-20.

Ranges only apply to *TClientDataSet* components. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges.

Understanding the differences between ranges and filters

Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than "Jones" and less than "Smith". Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to use for locating records, you can assign the index on which to define a range using either the *IndexName* or the *IndexFieldNames* property.

A filter, on the other hand, is any set of records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters can make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use the WHERE clause of a query to select data before it ever fills the client dataset's in-memory cache.

Specifying ranges

There are two mutually exclusive ways to specify a range for a client dataset:

- Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*.
- Specify both endpoints at once using *SetRange*.

Setting the beginning of a range

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the *Fields* property are treated as starting index values to use when applying the range. Fields specified must apply to the current index.

For example, suppose your application uses a client dataset named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. The following code can be used to create and apply a range:

Delphi example

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

C++ example

```
Customers->SetRangeStart();
Customers->FieldValues["CustNo"] = StrToInt(StartVal->Text);
Customers->SetRangeEnd();
if (!EndVal->Text.IsEmpty())
  Customers->FieldValues["CustNo"] = StrToInt(EndVal->Text);
Customers->ApplyRange();
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal* is null, then all records from the beginning of the dataset are included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records are included, since none are less than null.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you try to set a value for a field that is not in the index, the dataset raises an exception.

Tip To start at the beginning of the dataset, omit the call to *SetRangeStart*.

To finish specifying the start of a range, call *SetRangeEnd* or apply or cancel the range. For information about applying and canceling ranges, see “Applying or canceling a range” on page 25-11.

Setting the end of a range

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range. Fields specified must apply to the current index.

Warning Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the IDE assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the *FieldByName* method. For example,

Delphi example

```
with ClientDataSet1 do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

C++ example

```
ClientDataSet1->SetRangeStart();
ClientDataSet1->FieldByName(“LastName”)->Value = Edit1->Text;
ClientDataSet1->SetRangeEnd();
ClientDataSet1->FieldByName(“LastName”)->Value = Edit2->Text;
ClientDataSet1->ApplyRange();
```

As with specifying start of range values, if you try to set a value for a field that is not in the index, the dataset raises an exception.

To finish specifying the end of a range, apply or cancel the range. For information about applying and canceling ranges, see “Applying or canceling a range” on page 25-11.

Setting start- and end-range values

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

SetRange takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statements establish a range based on a two-column index:

Delphi example

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

C++ example

```
TVarRec StartVals[2];
TVarRec EndVals[2];
StartVals[0] = Edit1->Text;
StartVals[1] = Edit2->Text;
EndVals[0] = Edit3->Text;
EndVals[1] = Edit4->Text;

Table1->SetRange(StartVals, 1, EndVals, 1);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field.

Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the dataset assumes the ending value of the range is a null value. A range with null ending values is always empty because the starting range is greater than or equal to the ending range.

Specifying a range based on partial keys

If a key is composed of one or more string fields, the *SetRange* methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

Delphi example

```
ClientDataSet1.SetRangeStart;
ClientDataSet1['LastName'] := 'Smith';
ClientDataSet1.SetRangeEnd;
ClientDataSet1['LastName'] := 'Zzzzzz';
ClientDataSet1.ApplyRange;
```

C++ example

```
ClientDataSet1->SetRangeStart();
ClientDataSet1->FieldValues["LastName"] = "Smith";
ClientDataSet1->SetRangeEnd();
ClientDataSet1->FieldValues["LastName"] = "Zzzzzz";
ClientDataSet1->ApplyRange();
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith.” The value specification could also be:

```
ClientDataSet1['LastName'] := 'Sm';
```

```
ClientDataSet1->FieldValues["LastName"] = "Sm";
```

This statement includes records that have *LastName* greater than or equal to “Sm.”

Including or excluding records that match boundary values

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is false by default.

If you prefer, you can set the *KeyExclusive* property for a client dataset to true to exclude records equal to ending range. For example,

Delphi example

```
ClientDataSet1.KeyExclusive := True;
ClientDataSet1.SetRangeStart;
ClientDataSet1['LastName'] := 'Smith';
ClientDataSet1.SetRangeEnd;
ClientDataSet1['LastName'] := 'Tyler';
ClientDataSet1.ApplyRange;
```

C++ example

```
ClientDataSet1->SetRangeStart();
ClientDataSet1->KeyExclusive = true;
ClientDataSet1->FieldValues["LastName"] = "Smith";
ClientDataSet1->SetRangeEnd();
ClientDataSet1->FieldValues["LastName"] = "Tyler";
ClientDataSet1->ApplyRange();
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith” and less than “Tyler”.

Modifying a range

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

The process for editing and applying a range involves these general steps:

1. Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
2. Modifying the ending index value for the range.
3. Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

Editing the start of a range

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the *Fields* property overwrite the current index values to use when applying the range.

Tip If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range. For more information about ranges based on partial keys, see “Specifying a range based on partial keys” on page 25-9.

Editing the end of a range

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range.

Applying or canceling a range

When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

Applying a range

When you specify a range, the boundary conditions you define are not put into effect until you apply the range. To make a range take effect, call the *ApplyRange* procedure. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

Canceling a range

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

Delphi example

```
⋮
ClientDataSet1.CancelRange;
⋮
{later on, use the same range again. No need to call SetRangeStart, etc.}
ClientDataSet1.ApplyRange;
⋮
```

C++ example

```
⋮
ClientDataSet1->CancelRange();
⋮
// later on, use the same range again. No need to call SetRangeStart, etc.
ClientDataSet1->ApplyRange();
⋮
```

Representing master/detail relationships

Client datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master).

Client datasets support master/detail relationships in two very distinct ways:

- Make the client dataset the detail of another dataset by linking cursors. This process is described in "Making the client dataset a detail of another dataset" below.
- Make the client dataset the master in a master/detail relationship using nested detail tables. This process is described in "Using nested detail tables" on page 25-14.

Each of these approaches has its unique advantages. Linking cursors lets you create master/detail relationships where the master table is not a client dataset. With nested details both datasets must be client datasets, but they provide for more options in how to display the data and are better suited for applying edits back to the server.

Making the client dataset a detail of another dataset

A client dataset's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets.

The *MasterSource* property is used to specify a data source from which the client dataset gets data from the master table. This data source can be linked to any type of dataset, it need not be another client dataset. For instance, you can link a client dataset to a unidirectional dataset, so that the client dataset tracks the events occurring in the unidirectional dataset, by specifying the unidirectional dataset's data source component in this property.

The client dataset is linked to the master table based on its current index. Before you specify the fields in the master dataset that are tracked by the (detail) client dataset, first specify the index in the client dataset that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you have specified the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the indexed fields in the (detail) client dataset. To link datasets based on multiple column names, use a semicolon delimited list:

```
ClientDataSet1.MasterFields := 'OrderNo;ItemNo';
ClientDataSet1->MasterFields = "OrderNo;ItemNo";
```

To help create meaningful links between two datasets, you can use the Field Link designer. To use the Field Link designer, double click on the *MasterFields* property in the Object Inspector after you have assigned a *MasterSource* and an index.

The following steps create a simple form in which a user can step through customer records and display all orders for the current customer. The master dataset is called *CustomersTable*, and the detail dataset is *OrdersTable*.

1. Place a *TSQLConnection*, a *TSQLDataSet* component, a *TSQLClientDataSet*, and two *TDataSource* components in a data module.
2. Set the properties of the *TSQLConnection* component as follows:
   - *DriverName*: INTERBASE
   - *Params*:

- *Database*: the full path name for employee.gdb. (This database should be installed when you install interbase.)
- *UserName*: your user name.
- *Password*: your password.

3 Set the properties of the *TSQLDataSet* component as follows:

- *SQLConnection*: *SQLConnection1*
- *CommandType*: *ctTable*
- *CommandText*: CUSTOMER
- *Name*: CustomersTable

4 Set the properties of the first *TDataSource* component as follows:

- *Name*: CustSource
- *DataSet*: CustomersTable

5 Set the properties of the *TSQLClientDataSet* component as follows:

- *DBConnection*: *SQLConnection1*
- *CommandText*: Select * from SALES
- *Name*: OrdersTable
- *IndexFieldNames*: Cust_No
- *MasterSource*: CustSource.

The last two properties (*IndexFieldNames* and *MasterSource*) link the CUSTOMER table (the master table) to the ORDERS table (the detail table) using the CUST_No field on the orders table.

6 Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link Designer to set the following properties:

- Select *Cust_No* in both the Detail Fields and Master Fields field lists.
- Click the Add button to add this join condition. In the Joined Fields list, “CustNo -> CustNo” appears.
- Choose OK to commit your selections and exit the Field Link Designer.

7 Set the properties of the second *TDataSource* component as follows:
   - *Name*: OrdersSource
   - *DataSet*: OrdersTable
8 Place a *TDBText*, a *TDBGrid*, and a *TButton* on a form.
9 In the Delphi IDE, choose File | Use Unit to specify that the form should use the data module. In the C++ IDE, choose File | Include Unit Hdr.
10 Set the *DataSource* property of the DB grid to "OrdersSource".
11 Set the following properties on the DB text control:
   - *DataSource*: CustSource
   - *DataField*: CUSTOMER.
12 Double-click on the button, and in its *OnClick* event handler type:

```
CustomersTable.Next;
CustomersTable->Next();
```

13 Set the *Active* properties of *CustomersTable* and *OrdersTable* to true to display data in the form.
14 Compile and run the application.

If you run the application now, you will see that the datasets are linked together, and that when you press the button to display a new customer name in the DB text control, you see only those records in the ORDERS table that belong to the current customer.

Using nested detail tables

There are two ways to set up master/detail relationships in client datasets using nested tables:

- Obtain records that contain nested details from a provider component. When a provider component represents the master table of a master/detail relationship, it automatically creates a nested dataset field to represent the details for each record. This method only applies to *TClientDataSet*, because you do can't set up a separate provider component to represent a master table when using *TSQLClientDataSet*.
- Define nested details using the Fields Editor. At design time, right click the client dataset and choose Fields Editor. Add a new persistent field to your client dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of your detail table.

Note To use nested detail sets, the *ObjectView* property of the client dataset must be true.

When your client dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. For more information on how this works, see "Displaying dataset fields" on page 23-25.

Alternately, you can display and edit these datasets in data-aware controls by using a separate client dataset for the detail set. At design time, create persistent fields for the fields in your (master) client dataset, including a DataSet field for the nested detail set.

You can now create a client dataset to represent the nested detail set. Set this detail dataset's *DataSetField* property to the persistent dataSet field in the master dataset.

Using nested detail sets is necessary if you want to apply updates from master and detail tables to a database server. In file-based database applications, using nested detail sets lets you save the details with the master records in one file, rather than requiring you load two datasets separately, and then recreate the indexes to re-establish the master/detail relationship.

Constraining data values

Client datasets let you specify limits on the values a user can enter when editing the data. There are two types of constraints you can impose on user edits: field-level constraints and record-level constraints.

Each field component has two properties that you can use to specify constraints:

- The *DefaultExpression* property defines a default value that is assigned to the field if the user does not enter a value. Note that if the database server or source dataset also assigns a default expression for the field, the client dataset's version takes precedence because it is assigned before the update is returned to the provider.
- The *CustomConstraint* property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints are useful for validating data before it is sent to the server. For example, you may want to duplicate constraint conditions that are imposed by the database server. That way, user edits that would violate server constraints are enforced on the client side, and are never passed to the database server where they would be rejected. This means that fewer updates generate error conditions during the updating process. For more information about working with custom constraints on field components, see "Specifying constraints" on page 23-20.

At the record level, you can specify constraints using the client dataset's *Constraints* property. *Constraints* is a collection of *TCheckConstraint* objects, where each object represents a separate condition. Use the *CustomConstraint* property of a *TCheckConstraint* object to add your own constraints that are checked when you post records.

When fetching data from a database server (using *TSQLClientDataSet* or using *TClientDataSet* with provider component), there may be times when you want to turn off enforcement of data constraints, especially when the client dataset does not contain all of the records from the source dataset (or database server). For example, if a server constraint is based on the current maximum value in a field, but the client dataset fetches multiple packets of records, the current maximum value for a field in the client dataset may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client dataset applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call a client dataset's *DisableConstraints* method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset's *EnableConstraints* method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

Tip Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

Making data read-only

TDataSet introduces the *CanModify* property so that applications can determine whether the data in a dataset can be edited. Applications can't change the *CanModify* property, because some *TDataSet* descendants, such as unidirectional datasets, introduce no built-in support for posting edits.

However, because client datasets represent in-memory data, your application can always control whether users can edit that data. (It is possible, however, that the application can't post updates to a database server if the server table is read-only). To prevent users from modifying data, set the *ReadOnly* property of the client dataset to true. Setting *ReadOnly* to true sets the *CanModify* property to false.

You do not need to close a client dataset to change its read-only status. An application can make a client dataset read-only or not on a temporary basis at any time merely by changing the current setting of the *ReadOnly* property.

Editing data

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's *Data* property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the *Delta* property. Using a change log serves two purposes:

- When working with a provider, the change log is required by the mechanism for applying updates to the server.
- In any application, the change log provides sophisticated support for undoing changes.

The *LogChanges* property lets you disable logging temporarily. When *LogChanges* is true, changes are recorded in the log. When *LogChanges* is false, changes are made directly to the *Data* property. You can disable the change log in file-based applications when you do not need the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes
- Saving changes

Note Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

Undoing changes

Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state:

- To remove the last change to a record, call *UndoLastChange*. *UndoLastChange* takes a boolean parameter, *FollowChange*, that indicates whether to reposition the cursor on the restored record (true), or to leave the cursor on the current record (false). If there are several changes to a single record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a boolean value indicating success or failure to remove a change. If the removal occurs, *UndoLastChange* returns false. Use the *ChangeCount* property to determine whether there are any more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.
- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call *RevertRecord*. *RevertRecord* removes any changes to the current record from the change log.
- At any point during edits, you can save the current state of the change log using the *SavePoint* property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling *CancelUpdates*. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

Saving changes

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client dataset stores its data in a file or represents data from a server. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. "Merging changes into data" on page 25-44 describes this process.

You can't use *MergeChangeLog* if you are using the client dataset to represent server data. The information in the change log is required so that the updated records can be resolved with the data stored in the database (or source dataset). Instead, you call *ApplyUpdates*, which sends the changes to the database server (or source dataset) and updates the *Data* property only when the modifications have been successfully posted. See "Applying updates" on page 25-34 for more information about this process.

Sorting and indexing

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They let you apply ranges to limit the available records.
- They let your application to set up relationships between client datasets such as lookup tables or master/detail links.
- They specify the order in which records appear.

If a client dataset uses a provider (including the internal provider used by *TSQLClientDataSet*), it inherits a default index and sort order based on the data it receives from the provider. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called CHANGEINDEX, on the changed records stored in the change log (*Delta* property). CHANGEINDEX orders all records in the client dataset as they would appear if the changes specified in *Delta* were applied. CHANGEINDEX is based on the ordering inherited from DEFAULT_ORDER. As with DEFAULT_ORDER, you cannot change or delete the CHANGEINDEX index.

You can use other existing indexes for a dataset, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets.

Adding a new index

There are three ways to add indexes to a client dataset:

- To create a temporary index at runtime that sorts the records in the client dataset, you can use the *IndexFieldNames* property. Specify field names, separated by semicolons. Ordering of field names in the list determines their order in the index.

  This is the least powerful method of adding indexes. You can't specify a descending or case-insensitive index, and the resulting indexes do not support grouping. These indexes do not persist when you close the dataset, and are not saved when you save the client dataset to a file.
- To create an index at runtime that can be used for grouping, call *AddIndex*. *AddIndex* lets you specify the properties of the index, including
  - The name of the index. This can be used for switching indexes at runtime.

  - The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields.
  - How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can specify options to make the entire index case-insensitive or to sort in descending order. Alternately, you can provide a list of fields to be sorted case-insensitively and a list of fields to be sorted in descending order.
  - The default level of grouping support for the index.

  Indexes created with *AddIndex* do not persist when the client dataset is closed. (That is, they are lost when you reopen the client dataset). You can't call *AddIndex* when the dataset is closed. Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.

- The third way to create an index is at the time the client dataset is created. Before creating the client dataset, specify the desired indexes using the *IndexDefs* property. The indexes are then created along with the underlying dataset when you call *CreateDataSet*. See "Creating a dataset using field and index definitions" on page 25-41 for details.

  As with *AddIndex*, indexes you create with the dataset support grouping, can sort in ascending order on some fields and descending order on others, and can be case insensitive on some fields and case sensitive on others. Indexes created this way always persist and are saved when you save the client dataset to a file.

Tip You can index and sort on internally calculated fields with client datasets.

Deleting and switching indexes

To remove an index you created for a client dataset, call *DeleteIndex* and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes.

To use a different index with a client dataset when more than one index is available, use the *IndexName* property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the Object Inspector.

Obtaining information about indexes

At runtime, your application can ask the client dataset for information about the available indexes.

The *GetIndexNames* method retrieves a list of available indexes for a table. *GetIndexNames* fills a string list with valid index names. For example, the following code fetches the list of indexes available for *ClientDatSet1*:

D **Delphi example**

```
var
  IndexList: TList;
begin
⋮
```

```
IndexList := TStringList.Create;
ClientDataSet1.GetIndexNames(IndexList);
```

C++ example

```
TStringList *IndexList = new TStringList();
ClientDataSet1->GetIndexNames(IndexList);
```

You can also examine a list of fields in the current index. To iterate through all the fields in the current index, use two properties:

- *IndexFieldCount* property, which indicates the number of columns in the index.
- *IndexFields* property, which lists the fields that comprise the index.

IndexFields is an array of *TField* components, one for each field in the index. The following code fragment illustrates how you can use *IndexFieldCount* and *IndexFields* to iterate through the fields in the current index:

Delphi example

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with ClientDataSet1 do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I].FieldName;
  end;
end;
```

C++ example

```
AnsiString ListOfIndexFields[20];
for (int i = 0; i < CustomersTable->IndexFieldCount; i++)
  ListOfIndexFields[i] = CustomersTable->IndexFields[i]->FieldName;
```

Using indexes to group data

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Because of the sort order, adjacent values in the SalesRep column are duplicated. Within the records for SalesRep 1, adjacent values in the Customer column are duplicated. That is, the data is grouped by SalesRep, and within the SalesRep group it is grouped by Customer. Each grouping has an associated level. In this case, the SalesRep group has level 1 (because it is not nested in any other groups) and the Customer group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index.

Client datasets let you determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to only display a field value if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

To determine where the current record falls within any group, use the *GetGroupState* method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index). *GetGroupState* can't provide information about groups beyond that level, even if the index sorts records on additional fields.

Representing calculated values

As with any dataset, you can add calculated fields to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record. For more information about using calculated fields, see "Defining a calculated field" on page 23-7.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields. For more information on internally calculated fields, see "Using internally calculated fields in client datasets" below.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates. For more information on maintained aggregates, see "Using maintained aggregates" on page 25-22.

Using internally calculated fields in client datasets

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an *OnCalcFields* event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnCalcFields* event handler. However, you can optimize your event handler by checking the *State* property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset. If you are creating the client dataset using persistent fields, define fields as internally calculated by selecting InternalCalc in the Fields editor. If you are creating the client dataset using field definitions, set the *InternalCalcField* property of the relevant field definition to true.

Note Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the remote database server.

Using maintained aggregates

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a "maintained aggregate."

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping.

Specifying aggregates

To specify that you want to calculate summaries over the records in a client dataset, use the *Aggregates* property. *Aggregates* is a collection of aggregate specifications (*TAggregate*). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the *Add* method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

Note When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields. For details on creating aggregated persistent fields, see "Defining an aggregate field" on page 23-10.

For each aggregate, the *Expression* property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in Table 25.2

Table 25.2 Summary operators for maintained aggregates

Operator	Use
Sum	Totals the values for a numeric field or expression
Avg	Computes the average value for a numeric or date-time field or expression
Count	Specifies the number of non-blank values for a field or expression
Min	Indicates the minimum value for a string, numeric, or date-time field or expression
Max	Indicates the maximum value for a string, numeric, or date-time field or expression

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can't nest summary operators, however.) You can create expressions by using operators on summarized values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

Sum(Qty * Price)	{**legal** -- summary of an expression on fields}
Max(Field1) - Max(Field2)	{**legal** -- expression on summaries}
Avg(DiscountRate) * 100	{**legal** -- expression of summary and constant}
Min(Sum(Field1))	{**illegal** -- nested summaries}
Count(Field1) - Field2	{**illegal** -- expression of summary and field}

Aggregating over groups of records

By default, maintained aggregates are calculated so that they summarize all the records in a client dataset. However, you can specify that you want to summarize over the records in a group instead. This allows you to provide intermediate summaries such as subtotals for groups of records that share a common field value.

Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping. See "Using indexes to group data" on page 25-20 for information on grouping support.

Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

Delphi example

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

C++ example

```
Agg->Expression = "Sum(Amount)";
Agg->IndexName = "SalesCust";
Agg->GroupingLevel = 1;
Agg->AggregateName = "Total for Rep";
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which aggregates are valid at any time, use the *ActiveAggs* property.

Obtaining aggregate values

To get the value of a maintained aggregate, call the *Value* method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the *GetGroupState* method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's *Aggregates* is automatically updated to include the appropriate aggregate specification. The *AggFields* property contains the new aggregated field component, and the *FindField* method returns it.

Adding application-specific information to the data

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream. It is copied when you copy the data to another dataset. Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the *SetOptionalParam* method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the *GetOptionalParam* method, passing in the name that was used when the information was stored.

Copying data from another dataset

To copy the data from another dataset at design time, right click the dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one from which you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

Assigning data directly

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an OleVariant. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a *TSQLClientDataSet* object or a *TClientDataSet* object that is linked to a provider, data packets are automatically assigned to *Data*. See "Using an SQL client dataset" on page 25-37 for information on using *TSQLClientDataSet*. See "Using a client dataset with a provider" on page 25-27 for information on using *TClientDataSet* with a provider component.

When you are not using the data from the database or provider, you can copy the data from another client dataset as follows:

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

```
ClientDataSet1->Data = ClientDataSet2->Data;
```

Note When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

You can copy data to *TClientDataSet* from a dataset that is not a client dataset by creating a dataset provider component, linking it to the source dataset, and then copying its data:

Delphi example

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

C++ example

```
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = SourceDataSet;
ClientDataSet1->Data = TempProvider->Data;
delete TempProvider;
```

Note When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

With any client dataset, you can use a provider component to merge changes from another client dataset. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

Delphi example

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SQLClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

C++ example

```
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = SQLClientDataSet1;
TempProvider->ApplyUpdates(SourceDataSet->Delta, -1, ErrCount);
delete TempProvider;
```

Note When you merge data this way, it is merged into the destination client dataset's in-memory cache. You must still call that client dataset's *ApplyUpdates* method to then apply those changes back to the database server.

Cloning a client dataset cursor

Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

CloneCursor takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider interface.

When *Reset* and *KeepSettings* are false, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is true, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is false, and no connection component or provider is specified). When *KeepSettings* is true, the destination dataset's properties are not changed.

Using a client dataset with a provider

When using a client dataset to represent data from a database server or another dataset, the client dataset uses a provider component. The provider component passes data from a source dataset to the client dataset. (When using *TSQLClientDataSet*, this source dataset is an internal query component for accessing the data; with *TClientDataSet*, it is a dataset component you add to a form or data module.) After editing the data in memory, the client dataset applies updates, through the provider, back to the source dataset or directly to a remote database server.

Because with *TSQLClientDataSet* components the provider is internal, it can't be accessed directly. However, you can still use the client dataset's properties and methods to affect the communication between the client dataset and its internal provider (and hence between the client dataset and the database server).

With *TClientDataSet*, the provider can reside in the same application as the client dataset, or it can be part of a separate application running on another system.

For more information about provider components, see Chapter 26, "Using provider components."

The following steps describe how to use a client dataset with a provider:

1. If you are using *TClientDataSet*, specify a data provider.
2. Optionally, Get parameters from the source dataset or pass parameters to the source dataset.
3. Depending on the client dataset and provider, you may specify the command to execute on the server.
4. Request data from the source dataset.
5. Update records.
6. Refresh records.

In addition, *TClientDataSet* lets you communicate with the provider using custom events.

Specifying a data provider

Before *TClientDataSet* can receive data from another dataset and apply updates to that dataset or its database server, it must be associated with a dataset provider. The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.

- If the provider is in the same application as the client dataset, you can associate it with a provider by choosing a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This works as long as the provider has the same *Owner* as the client dataset. (The client dataset and the provider have the same *Owner* if they are placed in the same form or data module.) To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset's *SetProvider* method.
- If the provider is on a remote application server, you need to use both the *RemoteServer* and *ProviderName* properties. *RemoteServer* specifies the name of a connection component from which to get a list of providers. The connection component resides in the same data module as the client dataset. It establishes and maintains a connection to an application server, sometimes called a "data broker".

  At design time, after you specify *RemoteServer*, you can select a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This list includes both local providers (in the same form or data module) and remote providers that can be accessed through the connection component.

Note The connection component that connects *TClientDataSet* to a provider on a remote application server must be purchased separately.

At runtime, you can switch among available providers (both local and remote) by setting *ProviderName* in code.

Getting parameters from the source dataset

There are two circumstances when a client dataset needs to obtain parameter values from its source dataset:

- The client needs to know the value of output parameters on a stored procedure.
- The client wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset.

A client dataset stores parameter values in its *Params* property. These values are refreshed with any output parameters whenever the client dataset fetches data from the source dataset. However, there may be times when a *TClientDataSet* component in a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the source dataset by calling the *FetchParams* method. The parameters are returned in a data packet from the provider and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

Note There is never any need to call *FetchParams* when working with *TSQLClientDataSet*, because the *Params* property always reflects the parameters on the internal source dataset. With *TClientDataSet*, The *FetchParams* method (or the Fetch Params command) only works if the client dataset is connected to a provider whose associated dataset can supply parameters. For example, if the source dataset is a *TSQLDataSet* object with a *CommandType* of *ctTable*, there are no parameters to fetch.

If the provider is on a separate system as part of a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a stateless application server, use the *Execute* method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

Passing parameters to the source dataset

Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify

- Parameter values for a query or stored procedure that is run on the application server
- Field values that limit the records sent in data packets

You can specify parameter values that your client dataset sends to the provider at design time or at runtime. At design time, select the client dataset, and then double-click the *Params* property in the Object Inspector. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the Object Inspector to edit the properties of that parameter.

At runtime, use the *CreateParam* method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code sets the value of a parameter named CustNo to 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
    AsInteger := 605;
```

```
TParam *pParam = ClientDataSet1->Params->CreateParam(ftInteger, "CustNo", ptInput);
pParam->AsInteger = 605;
```

If the client dataset is not active, you can send the parameters to the provider and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to true.

Sending query or stored procedure parameters

When a provider represents the results of a query or stored procedure, you can use the *Params* property to specify parameter values. When the client dataset requests data from the source dataset or uses its *Execute* method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset. It then instructs the dataset to execute its query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

Note Parameter names should match the names of the corresponding parameters on the source dataset.

Limiting records with parameters

When a provider represents a *TSQLTable* component, you can use *Params* property to limit the records that are provided to the *Data* property.

Each parameter name must match the name of a field in the *TSQLTable* component associated with the provider. The provider then sends only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider an application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named CustID (or whatever field in the source table is called) whose value identifies the customer whose orders should be displayed. When the client dataset requests data from the source dataset, it passes this parameter value. The provider then sends only the records for the identified customer. This is more efficient than letting the provider send all the orders records to the client application and then filtering the records using the client dataset.

Specifying the command to execute on the server

When using *TSQLClientDataSet*, the application does not have direct access to the source dataset. Instead, the client dataset must specify an SQL statement it executes to produce data packets. It does this using the *CommandText* property. The *CommandType* property indicates whether *CommandText* represents an SQL statement for the server to execute, the name of a table, or the name of a stored procedure.

When using *TClientDataSet*, the source dataset typically determines what data is supplied to clients. This dataset may have a property that specifies an SQL statement to generate the data, or it may represent a specific database table or stored procedure. For example, the *CommandText* property of *TSQLDataSet* specifies an SQL statement, a table name, or a stored procedure name, depending on the value of the *CommandType* property, while the *ProcedureName* property of *TSQLStoredProc* specifies the stored procedure it executes.

If the provider allows, *TClientDataSet* can override the property that indicates what data the dataset represents. This enables the client dataset to specify dynamically what data it wants to see. As with *TSQLClientDataSet*, you can set *CommandText* to an SQL statement that replaces the SQL on the provider's dataset, the name of a table or the name of a stored procedure. Which type of value to use is determined by the type of dataset associated with the provider.

By default, the internal provider used by *TSQLClientDataSet* always allows its client dataset to specify a *CommandText* value, because there is no other way to specify what data to fetch. For external provider components, however, the default is not to allow client datasets to specify *CommandText* in this way. To allow *TClientDataSet* to use its *CommandText* property, you must add *poAllowCommandText* to the *Options* property of the provider. Otherwise, the value of *CommandText* is ignored.

Warning Do not remove *poAllowCommandText* from the *Options* property of *TSQLClientDataSet* before opening the dataset.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the provider, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an *Execute* command to the provider.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the *AppServer* property.

Requesting data from the source dataset

The following table lists the properties and methods of client datasets that determine how data is fetched from a provider:

Table 25.3 Client datasets properties and method for handling data requests

Property or method	Purpose
FetchOnDemand property	Determines whether the client dataset automatically fetches data as needed, or relies on the application to call its *GetNextPacket*, *FetchBlobs*, and *FetchDetails* functions to retrieve additional data.
PacketRecords property	Specifies the type or number of records to include in each data packet.
GetNextPacket method	Fetches the next data packet from the provider.
FetchBlobs method	Fetches any BLOB fields for the current record when the provider does not include BLOB data automatically.
FetchDetails method	Fetches nested detail datasets for the current record when the provider does not include these in data packets automatically.

By default, a client dataset retrieves all records from the source dataset. You can change this using *PacketRecords* and *FetchOnDemand*.

Incremental fetching

PacketRecords specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the client dataset is first opened, or when the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after the client dataset first fetches data, it never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
ClientDataSet1.PacketRecords := 10;
```

```
ClientDataSet1->PacketRecords = 10;
```

This process of fetching records in batches is called "incremental fetching". Client datasets use incremental fetching when *PacketRecords* is greater than zero. By default, the client dataset automatically calls *GetNextPacket* to fetch data as needed. Newly fetched packets are appended to the end of the data already in the client dataset.

GetNextPacket returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

Warning Incremental fetching does not work if you are fetching data from a remote provider on a stateless application server.

You can also use *PacketRecords* to fetch metadata information about the source dataset. To retrieve metadata information, set *PacketRecords* to *0*.

Fetch-on-demand

Automatic fetching of records is controlled by the *FetchOnDemand* property. When *FetchOnDemand* is true (the default), automatic fetching is enabled. To prevent the client dataset from automatically fetching records as needed, set *FetchOnDemand* to false. When *FetchOnDemand* is false, the application must explicitly call *GetNextPacket* to fetch records.

For example, applications that need to represent extremely large read-only datasets can turn off *FetchOnDemand* to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client dataset frees its cache using the *EmptyDataSet* method. This approach, however, does not work well when the client must post updates to the server.

When using *TSQLClientDataSet*, the *Options* property controls whether records in data packets include BLOB data and nested detail datasets. When using *TClientDataSet*, the external provider controls whether this information is included in data packets. If data packets do not include this information, the *FetchOnDemand* property causes the client dataset to automatically fetch BLOB data and detail datasets on an as-needed basis. If *FetchOnDemand* is false and BLOB data and detail datasets are not included in data packets, you must explicitly call the *FetchBlobs* or *FetchDetails* method to retrieve this information.

Updating records

Client datasets work with a local copy of data. The user sees and edits this copy in the client application's data-aware controls. User changes are temporarily stored by the client dataset in an internally maintained change log. The contents of the change log are stored as a data packet in the *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database.

When a client applies updates to the server, the following steps occur:

1. The application calls the *ApplyUpdates* method of a client dataset object. This method passes the contents of the client dataset's *Delta* property to the provider. *Delta* is a data packet that contains a client dataset's updated, inserted, and deleted records.
2. The provider applies the updates to the database (or source dataset), caching any problem records that it can't resolve itself. See "Responding to client update requests" on page 26-8 for details on how the provider applies updates.
3. The provider returns all unresolved records to the client dataset in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
4. The client dataset attempts to reconcile update errors returned in the *Result* data packet on a record-by-record basis.

Applying updates

Changes made to the client dataset's local copy of data are not sent to the database server until the client application calls the *ApplyUpdates* method for the dataset. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called *Delta*) to the provider.

ApplyUpdates takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If *MaxErrors* is *-1*, any number of errors is tolerated, and the change log ends up containing all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

ApplyUpdates returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's *ApplyUpdates* method does the following:

1. It indirectly calls the provider's *ApplyUpdates* method. The provider's *ApplyUpdates* method writes the updates to the database (or source dataset) and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.
2. The client dataset 's *ApplyUpdates* method then attempts to reconcile these problem records by calling the *Reconcile* method. *Reconcile* is an error-handling routine that calls the *OnReconcileError* event handler. You must code the *OnReconcileError* event handler to correct errors. For details about using *OnReconcileError*, see "Reconciling update errors" on page 25-34.
3. Finally, *Reconcile* removes successfully applied changes from the change log and updates *Data* to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

Tip If the provider is on a stateless application server, you may want to communicate with it about persistent state information before or after you apply updates. The client dataset receives a *BeforeApplyUpdates* event before the updates are sent, which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), the client dataset receives an *AfterApplyUpdates* event where you can respond to any persistent state information returned by the application server.

Reconciling update errors

The provider returns error records and error information to the client dataset in a result data packet. If the provider returns an error count greater than zero, then for each record in the result data packet, the client dataset's *OnReconcileError* event occurs.

You should always code the *OnReconcileError* event handler, even if only to discard the records returned by the provider. The *OnReconcileError* event handler is passed four parameters:

- *DataSet:* A client dataset that contains the updated record which couldn't be applied. You can use client dataset methods to obtain information about the problem record and to change the record in order to correct any problems. In particular, you will want to use the *CurValue*, *OldValue*, and *NewValue* properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in an *OnReconcileError* event handler.
- *E:* An *EReconcileError* object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.
- *UpdateKind:* The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).
- *Action:* A **var** (Delphi) or reference (C++) parameter that lets you indicate what action to take when the *OnReconcileError* handler exits. On entry into the handler, *Action* is set to the action taken by the resolution process on the provider. In your event handler, you set this parameter to
  - Skip this record, leaving it in the change log. (raSkip)
  - Stop the entire reconcile operation. (raAbort)
  - Merge the modification that failed into the corresponding record from the server. (raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
  - Replace the current update in the change log with the value of the record in the event handler (which has presumably been corrected). (raCorrect)
  - Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel)
  - Update the current record value to match the record on the server. (raRefresh)

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit, which ships in the object repository directory. To use this dialog, add RecError to your uses clause (Delphi) or include RecError.hpp in your source unit (C++).

D

Delphi example

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

C++ example

```
void __fastcall TForm1::ClientDataSetReconcileError(TCustomClientDataSet *DataSet,
    EReconcileError *E, TUpdateKind UpdateKind, TReconcileAction &Action)
{
  Action = HandleReconcileError(this, DataSet, UpdateKind, E);
}
```

Refreshing records

Client datasets work with an in-memory snapshot of data. If that data comes from a database server, then as time elapses other users may modify it. The data in the client dataset becomes a less and less accurate picture of the underlying data.

Like any other dataset, client datasets have a *Refresh* method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client applications can also update the data while leaving the change log intact. To do this, call the client dataset's *RefreshRecord* method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the provider but leaves any changes in the change log.

Warning It may not always be appropriate to call *RefreshRecord*. If the user's edits conflict with changes to the underlying dataset made by other users, calling *RefreshRecord* will mask this conflict. When the client application applies its updates, no reconcile error will occur and the application can't resolve the conflict.

In order to avoid masking update errors, you may want to check that there are no pending updates for the current record before calling *RefreshRecord*. For example, the following code raises an exception if an attempt is made to refresh a modified record:

```
if ClientDataSet1.UpdateStatus <> usUnModified then
  raise Exception.Create('You must apply updates before refreshing the current record.');
ClientDataSet1.RefreshRecord;
```

```
if (ClientDataSet1->UpdateStatus != usUnModified)
  throw Exception("You must apply updates before refreshing the current record.");
ClientDataSet1->RefreshRecord;
```

Communicating with providers using custom events

Client datasets communicate with a provider component through a special interface called *IAppServer*. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server

TClientDataSet provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives special events that allow it to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

1 The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.
2 The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.
3 The provider goes through its normal process of assembling a data packet (including all the accompanying events).
4 The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client dataset.
5 The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that let you customize the communication between *TClientDataSet* and its provider.

In addition, *TClientDataSet* has a special method, *DataRequest*, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an *OnDataRequest* event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

Using an SQL client dataset

TSQLClientDataSet is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.

TSQLClientDataSet works the same way as a generic client dataset (*TClientDataSet*) that is linked to a unidirectional dataset by a dataset provider. In fact, *TSQLClientDataSet* has its own, internal provider, which it uses to communicate with an internally created unidirectional dataset.

Using an SQL client dataset can simplify the process of two-tiered application development because you don’t need to work with as many components. Although you can’t access the internal provider directly, the SQL client dataset publishes properties that let you configure how it applies updates. Typically, however, the defaults provided by *TSQLClientDataSet* are sufficient.

When to use TSQLClientDataSet

TSQLClientDataSet is intended for use in a simple two-tiered database applications and briefcase model applications. It provides an easy-to-set up component for linking to the database server, fetching data, caching updates, and applying them back to the server. It can be used in most two-tiered applications.

There are times, however, when it is more appropriate to use *TClientDataSet*:

- If you are not using data from a database server (for example, if you are using a dedicated file on disk), then *TClientDataSet* has the advantage of less overhead.
- Only *TClientDataSet* can be used in a multi-tiered database application. Thus, if you are writing a multi-tiered application, or if you intend to scale up to a multi-tiered application eventually, you should use *TClientDataSet* with an external provider and source dataset.
- Because the source dataset is internal to the SQL client dataset component, you can't link two source datasets in a master/detail relationship to obtain nested detail sets. (You can, however, link two SQL client datasets into a master/detail relationship.)
- The SQL client dataset does not surface many of the events that occur on its internal dataset provider. However, in most cases, these events are used in multi-tiered applications, and are not needed for two-tiered applications.

Setting up an SQL client dataset

To use *TSQLClientDataSet*,

1. Place the *TSQLClientDataSet* component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server that contains the data. There are two ways to do this:
   - If you have defined a named connection in the connections file, you can simply specify the name of that connection as the value of the *ConnectionName* property. For details on named connections, see "Naming a connection description" on page 21-3.
   - For greater control over connection properties, transaction support, login support, and the ability to use a single connection for more than one dataset, use a separate *TSQLConnection* component instead. Specify the *TSQLConnection* component as the value of the *DBConnection* property. For details on *TSQLConnection*, see Chapter 21, "Connecting to databases".
3. Indicate what data you want to fetch from the server. There are three ways to do this:
   - Set *CommandType* to *ctQuery* and set *CommandText* to an SQL statement you want to execute on the server. This statement is typically a SELECT statement. Supply the values for any parameters using the *Params* property.

- Set *CommandType* to *ctStoredProc* and set *CommandText* to the name of the stored procedure you want to execute. Supply the values for any input parameters using the *Params* property.
- Set *CommandType* to *ctTable* and set *CommandText* to the name of the database tables whose records you want to use.

4 If the data is to be used with visual data controls, add a data source component to the form or data module, and set its *DataSet* property to the *TSQLClientDataSet* object. The data source component forwards the data in the client dataset's in-memory cache to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.

5 If desired, configure the way the internal provider applies updates. The properties and events that allow this are described in "Configuring the internal provider" below.

6 Activate the dataset by setting the *Active* property to true (or, at runtime, calling the *Open* method).

7 If you executed a stored procedure, use the *Params* property to retrieve any output parameters.

8 When the user has edited the data in the SQL client dataset, you can apply those edits back to the database server by calling the *ApplyUpdates* method. Resolve any update errors in an *OnUpdateError* event handler. For more information on applying updates, see "Updating records" on page 25-33.

Configuring the internal provider

Because *TSQLClientDataSet* does not use an external provider component, you have less control over how it provides data and applies updates. Typically, this is not a problem, because *TSQLClientDataSet* publishes the properties and events of its internal provider that you are likely to need.

Because you specify the SQL statement to execute as the value of the SQL client dataset's *CommandText* property, you can control what data is included in data packets. Field properties can be set using persistent fields on the client dataset.

TSQLClientDataSet has two published properties that influence how the internal provider applies updates. These properties are the same as properties on the provider, and the *TSQLClientDataSet* component simply forwards any values you set on to the internal provider:

- *Options* controls whether nested detail sets and BLOB data are included in data packets or fetched separately, whether specific types of edits (insertions, modifications, or deletions) are disabled, whether a single update can affect multiple server records, and whether the client dataset's records are refreshed when it applies updates. *Options* is identical to the provider's *Options* property. As a result, it allows you to set options that are not relevant or appropriate to *TSQLClientDataSet*. For example, there is no reason to include *poIncFieldProps*, because the internal source dataset does not use persistent field components. Similarly, you do not want to exclude *poAllowCommandText*, which is included by default, because that would make it impossible for the client dataset to specify a query to execute. For information on the provider's *Options* property, see "Setting options that influence the data packets" on page 26-5.

- *UpdateMode* controls what fields are used to locate records in the SQL statements the provider generates for applying updates. *UpdateMode* is identical to the provider's *UpdateMode* property. For information on the provider's *UpdateMode* property, see "Influencing how updates are applied" on page 26-10.

TSQLClientDataSet also publishes several events that correspond to events on the internal provider. Some of the most important of these are

- *OnGetTableName*, which lets you supply the name of the database table to which the dataset should apply updates. This lets the internal provider generate SQL statements for updates when it can't identify the database table from the query specified by *CommandText*. For example, if the query executes a stored procedure or multi-table join that only requires updates to a single table, supplying an *OnGetTableName* event handler allows the internal provider to correctly apply updates.
- *BeforeUpdateRecord*, which occurs for every record in the delta packet. This event lets you make any last-minute changes before the record is inserted, deleted, or modified. It also provides a way for you to execute your own SQL statements to apply the update in cases where the provider can't generate correct SQL (for example, for multi-table joins where multiple tables must be updated.) To execute your own SQL statements, use the *Execute* method of the *TSQLConnection* component.
- *OnUpdateError*, which occurs every time the internal provider can't apply an update to the database server. This event lets you correct update errors during the update process so that they do not count toward the maximum number of errors allowed in the entire update operation.

Using a client dataset with file-based data

Client datasets can function independently of a provider, such as in file-based database applications and "briefcase model" applications. The special files that client datasets use for their data are called MyBase.

In MyBase applications (when it does not use a provider), the client dataset cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables
- Load saved data
- Merge edits into its data
- Save data

Creating a new dataset

There are three ways to define and create client datasets that do not represent server data:

- You can define and create a new client dataset by creating persistent fields for the dataset and then choosing Create Dataset from its context menu.
- You can define and create a new client dataset based on field definitions and index definitions.
- You can copy an existing dataset.

Creating a new dataset using persistent fields

The following steps describe how to create a new client dataset using the Fields Editor:

1. From the Component palette, add a *TClientDataSet* component to your application.
2. Right-click the client dataset and select Fields Editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field definition. Once the field is created, you can alter its properties in the Object Inspector by selecting the field in the Fields editor.

   Continue adding fields in the fields editor until you have described your client dataset.
3. Right-click the client dataset and choose Create DataSet. This creates an empty client dataset from the persistent fields you added in the Fields Editor.
4. Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains a dataset.)
5. In the File Save dialog, choose a file name and save a copy of your client dataset to that file.

Note You can also create the client dataset at runtime using persistent fields that are saved with the client dataset. Simply call the *CreateDataSet* method.

Creating a dataset using field and index definitions

If you want to create persistent indexes for your client dataset as well as fields, you must use field and index definitions. Use the *FieldDefs* property to specify the fields in your dataset and the *IndexDefs* property to specify any indexes. Once the dataset is specified, right-click the client dataset and choose Create DataSet at design time, or call the *CreateDataSet* method at runtime.

When defining the index definitions for your client dataset, two properties of the index definition apply uniquely to client datasets. These are *TIndexDef DescFields* and *TIndexDef CaseInsFields* properties.

DescFields lets you define indexes that sort records in ascending order on some fields and descending order on other fields. Instead of using the *ixDescending* option to sort in descending order on all the fields in the index, list only those fields that should sort in descending order as the value of *DescFields*. For example, when defining an index that sorts on Field1, then Field2, then Field3, setting *DescFields* to

```
Field1;Field3
```

results in an index that sorts Field2 in ascending order and Field1 and Field3 in descending order.

CaseInsFields lets you define indexes that sort records case-sensitively on some fields and case-insensitively on other fields. Instead of using the *isCaseInsensitive* option to sort case-insensitively on all the fields in the index, list only those fields that should sort case-insensitively as the value of *CaseInsFields*. Like *DescFields*, *CaseInsFields* takes a semicolon-delimited list of field names.

You can specify the field and index definitions at design time using the Collection editor. Just choose the appropriate property in the Object Inspector (*FieldDefs* or *IndexDefs*), and double-click to display the Collection editor. Use the Collection editor to add, delete, and rearrange definitions. By selecting definitions in the Collection editor you can edit their properties in the Object Inspector.

You can also specify the field and index definitions in code at runtime. For example, the following code creates and activates a client dataset in the form's *OnCreate* event handler:

Delphi example

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with ClientDataSet1 do
begin
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftInteger;
    Name := 'Field1';
  end;
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftString;
    Size := 10;
    Name := 'Field2';
  end;
  with IndexDefs.AddIndexDef do
  begin
    Fields := 'Field1';
    Name := 'IntIndex';
  end;
  CreateDataSet;
end;
end;
```

C++ example

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  TFieldDef *pDef = ClientDataSet1->FieldDefs->AddFieldDef();
```

```
    pDef->DataType = ftInteger;
    pDef->Name = "Field1";
    pDef = ClientDataSet1->FieldDefs->AddFieldDef();
    pDef->DataType = ftString;
    pDef->Size = 10;
    pDef->Name = "Field2";
    TIndexDef *pIndex = ClientDataSet1->IndexDefs->AddIndexDef();
    pIndex->Fields = "Field1";
    pIndex->Name = "IntIndex";
    ClientDataSet1->CreateDataSet();
  }
```

Creating a dataset based on an existing table

If you are converting a server-based database application into a file-based application, you can copy existing tables and save them to a file from the IDE. The following steps indicate how to copy an existing table:

1 From the dbExpress page of the Component palette, add a *TSQLDataSet* component and a *TSQLConnection* component to your application. Set the *CommandType* and *CommandText* properties of the SQL dataset to identify the existing database table. Set its *SQLConnection* property to the *TSQLConnection* component. Set properties on the *TSQLConnection* component to attach to the database server. Set the SQL dataset's *Active* property to true.

2 From the Component palette, add a *TClientDataSet* component.

3 Right-click the client dataset and select Assign Local Data. In the dialog that appears, choose the SQL dataset component that you added in step 1. Choose OK.

4 Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains data.)

5 In the File Save dialog, choose a file name and save a copy of your database table.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

Loading data from a file or stream

To load data from a file, call a client dataset's *LoadFromFile* method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the *FileName* property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream)* must have previously been saved in a client dataset's data format by this or another client dataset using the *SaveToFile (SaveToStream)* method. For more information about saving data to a file or stream, see "Saving data to a file or stream" on page 25-44.

When you call *LoadFromFile* or *LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property.

Merging changes into data

When you edit the data in a client dataset, the changes you make are recorded in the change log, but the changes do not affect the original version of the data.

To make your changes permanent, call *MergeChangeLog*. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, *Data* contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the *ChangeCount* property to *0*.

Warning Do not call *MergeChangeLog* for client datasets that represent the data from a database server or source dataset. In this case, call *ApplyUpdates* to write changes to the database or source dataset. For more information, see "Applying updates" on page 25-34.

Note It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider. For an example of how to do this, see "Assigning data directly" on page 25-25.

Saving data to a file or stream

If you use a client dataset in a file-based application, then when you edit data and merge it, the changes you make exist only in memory. To make a permanent record of your changes, you must write them to disk. You can save the data to disk using the *SaveToFile* method.

SaveToFile takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

If you always save the data to the same file, you can use the *FileName* property instead. If *FileName* is set, the data is automatically saved to the named file when the client dataset is closed.

You can also save data to a stream, using the *SaveToStream* method. *SaveToStream* takes one parameter, a stream object that receives the data.

Note If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the *LoadFromFile* or *LoadFromStream* method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component.

Note *SaveToFile* does not preserve any indexes you add to the client dataset unless they are created with the dataset using index definitions.

Chapter

26

Using provider components

Provider components (*TDataSetProvider* and *TXMLTransformProvider*) supply the mechanism by which client datasets obtain their data unless they use dedicated files on disk. Providers

- Receive data requests from a client dataset, fetch the requested data from the database server, package the data into a transportable data packet, and return the data to the client dataset. This activity is called "providing."
- Receive updated data from a client dataset, apply updates to the database, source dataset, or source XML document, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is called "resolving."

Most of the work of a provider component happens automatically. You need not write any code on the provider to create data packets from the data in another dataset or XML document or to apply updates to a dataset or database server. However, you may want to use the properties and events of the provider component to control its interaction with clients.

When using *TSQLClientDataSet*, the provider is internal to the client dataset, and the application has no direct access to it. When using *TClientDataSet*, however, the provider is a separate component that you can use to control what information is packaged for clients and for responding to events that occur around the process of providing and resolving. *TSQLClientDataSet* surfaces some of the provider's properties and events as its own properties and events, but for the greatest amount of control, you may want to use *TClientDataSet* with a separate provider component.

When using a separate provider component, it can reside in the same application as the client dataset, or it can reside on an application server as part of a multi-tiered application.

Note The components that connect a client dataset to a provider on a remote application server must be purchased separately.

This chapter describes how to use a provider component to control the interaction with client datasets.

Determining the source of data

When you use a provider component, you must specify the source it uses to get the data it assembles into data packets. You can specify the source as one of the following:

- To provide the data from a dataset, use *TDataSetProvider*.
- To provide the data from an XML document, use *TXMLTransformProvider*.

Using a dataset as the source of the data

If the provider is a dataset provider (*TDataSetProvider*), set the *DataSet* property of the provider to indicate the source dataset. At design time, select from available datasets in the *DataSet* property drop-down list in the Object Inspector.

TDataSetProvider interacts with the source dataset using the *IProviderSupport* interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don’t do anything or that raise exceptions.

The unidirectional dataset classes that ship with the IDE override these methods to implement the *IProviderSupport* interface in a more useful fashion. Client datasets use the default implementation inherited from *TDataSet*, but can still be used as a source dataset as long as the provider’s *ResolveToDataSet* property is true.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to supply data to a provider. If the provider need only provide data packets on a read-only basis (that is, if it does not need to apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

Using an XML document as the source of the data

If the provider is an XML provider, set the *XMLDataFile* property of the provider to indicate the source document.

XML providers must transform the source document into data packets, so in addition to indicating the source document, you must also specify how to transform that document into data packets. This transformation is handled by the provider’s *TransformRead* property. *TransformRead* represents a *TXMLTransform* object. You can set its properties to specify what transformation to use, and use its events to provide your own input to the transformation.

Communicating with the client dataset

All communication between a provider and a client dataset takes place through an *IAppServer* interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you. If the provider is part of a multi-tiered application, this is the interface for the application server (in Delphi, this is the interface for the application server's remote data module).

IAppServer provides the bridge between client dataset and the provider. Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset. However, when necessary, you can make direct calls to the *IAppServer* interface by using the *AppServer* property of the client dataset.

Table 26.1 lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter. In multi-tiered applications, this parameter indicates the provider on the application server with which the client dataset communicates. Most methods also include an OleVariant parameter called *OwnerData* that allows a client application and an application server to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your provider to adjust to application-defined information before and after each call from a client dataset.

Table 26.1 AppServer interface members

IAppServer	TDataSetProvider	TClientDataSet
AS_ApplyUpdates method	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event
AS_DataRequest method	DataRequest method, OnDataRequest event	DataRequest method
AS_Execute method	Execute method, BeforeExecute event, AfterExecute event	Execute method, BeforeExecute event, AfterExecute event
AS_GetParams method	GetParams method, BeforeGetParams event, AfterGetParams event	FetchParams method, BeforeGetParams event, AfterGetParams event
AS_GetProviderNames method	Used to identify all available providers	Used to create a design-time list for ProviderName property
AS_GetRecords method	GetRecords method, BeforeGetRecords event, AfterGetRecords event	GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event
AS_RowRequest method	RowRequest method, BeforeRowRequest event, AfterRowRequest event	FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event

Choosing how to apply updates

TXMLTransformProvider components always apply updates to the associated XML document. When using *TDataSetProvider*, however, you can choose how updates are applied. By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your application does not need to merge updates twice (first to the source dataset, and then to the remote database server).

However, you may not always want to take this approach. For example, you may be using a dataset that does not get its data from an SQL server (for example if you are providing from a *TClientDataSet* component). Alternately, you may be using a custom dataset that can apply updates to an SQL server, but you want to use some of the dataset events.

TDataSetProvider lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the *ResolveToDataSet* property. When this property is true, updates are applied to the dataset. When it is false, the provider generates SQL statements to apply updates and the source dataset forwards them to the database server.

Controlling what information is included in data packets

There are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets
- Setting options that influence the data packets
- Adding custom information to data packets

Note These techniques for controlling the content of data packets are only available for dataset providers. When using *TXMLTransformProvider*, you can only control the content of data packets by controlling the transformation file the provider uses.

Specifying what fields appear in data packets

To control what fields are included in data packets, create persistent fields on the dataset that the provider uses to build the packets. The provider then includes only these fields. Persistent fields whose values are generated dynamically by the source dataset (such as calculated fields) can be included, but appear to client datasets on the receiving end as static read-only fields. For information about persistent fields, see “Persistent field components” on page 23-3.

If the client dataset will be editing the data and applying updates to the database server, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields that are provided only to ensure uniqueness, set the *ProviderFlags* property for those fields to include *pfHidden*.

Note Including enough fields to avoid duplicate records is also a consideration when specifying a query on the source dataset. The query should include enough fields so that records are unique, even if your application does not use all the fields.

Setting options that influence the data packets

The *Options* property of the provider component lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

Table 26.2 Provider options

Value	Meaning
poReadOnly	The client dataset can't apply updates to the provider.
poDisableEdits	Client datasets can't modify existing data values. If the user tries to edit a field, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or delete records.)
poDisableInserts	Client datasets can't insert new records. If the user tries to insert a new record, the client dataset raises an exception. (This does not affect the client dataset's ability to delete records or modify existing data.)
poDisableDeletes	Client datasets can't delete records. If the user tries to delete a record, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or modify records.)
poFetchBlobsOnDemand	BLOB field values are not included in the data packet. Instead, client datasets must request these values on an as-needed basis. If the client dataset's *FetchOnDemand* property is true, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchBlobs* method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider represents the master of a master/detail relationship, nested detail values are not included in the data packet. Instead, client applications request these on an as-needed basis. If the client dataset's *FetchOnDemand* property is true, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchDetails* method to retrieve nested details.
poIncFieldProps	The data packet includes the following field properties (where applicable): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues.*
poCascadeDeletes	When the provider represents the master of a master/detail relationship, the server deletes detail records when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity.

Table 26.2 Provider options (continued)

Value	Meaning
poCascadeUpdates	When the provider represents the master of a master/detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity.
poAllowMultiRecordUpdates	A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, SQL statements on the source dataset, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence.
poNoReset	Client datasets can't specify that the provider should reposition the cursor on the first record before providing data.
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poPropogateChanges	Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset.
poAllowCommandText	The client dataset can override the associated dataset's SQL text or the name of the table or stored procedure it represents.
poRetainServerOrder	The client should not re-sort records to establish a default order.

Adding custom information to data packets

Providers can send application-defined information to the data packets using the *OnGetDataSetProperties* event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets can then retrieve the information using their *GetOptionalParam* method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client may never be aware of the information, but the provider can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using a Variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Multiple attributes can be added by creating a Variant array of Variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number of records in the source dataset. Only information about the time the data was provided is returned when client datasets apply updates:

D

Delphi example

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet;
out Properties: OleVariant);
begin
```

```
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

C++ example

```
void __fastcall TMyDataModule1::Provider1GetDataSetProperties(TObject *Sender, TDataSet
*DataSet, out OleVariant Properties)
{
  int ArrayBounds[2];
  ArrayBounds[0] = 0;
  ArrayBounds[1] = 1;
  Properties = VarArrayCreate(ArrayBounds, 1, varVariant);
  Variant values[3];
  values[0] = Variant("TimeProvided");
  values[1] = Variant(Now());
  values[2] = Variant(true);
  Properties[0] = VarArrayOf(values,2);
  values[0] = Variant("TableSize");
  values[1] = Variant(DataSet->RecordCount);
  values[2] = Variant(false);
  Properties[1] = VarArrayOf(values,2);
}
```

When the client dataset applies updates, the time the original records were provided can be read in the provider's *OnUpdateData* event:

Delphi example

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

C++ example

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  Variant WhenProvided = DataSet->GetOptionalParam("TimeProvided");
  ...
}
```

Responding to client data requests

In most applications, requests for data are handled automatically. A client dataset requests a data packet by calling *GetRecords* (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset or XML document, creating a data packet, and returning the packet to the client dataset.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client dataset. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or, in a multi-tiered application, you might want to encrypt sensitive data before it is sent on to the client.

To edit the data packet before sending it on to the client dataset, write an *OnGetData* event handler. The data packet is provided as a parameter in the form of a client dataset. Using the methods of this client dataset, data can be edited before it is sent to the client.

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client dataset before and after the call to *GetRecords*. This communication takes place using the *BeforeGetRecords* and *AfterGetRecords* event handlers.

Responding to client update requests

A provider applies updates to database records based on a *Delta* data packet received from a client dataset. The client dataset requests updates by calling the *ApplyUpdates* method (indirectly, through the *IAppServer* interface).

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client dataset before and after the call to *ApplyUpdates*. This communication takes place using the *BeforeApplyUpdates* and *AfterApplyUpdates* event handlers.

When a provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet before it is written to the dataset or influence how updates are applied. After the *OnUpdateData* event, the provider writes the changes to the database.

The provider performs the update on a record-by-record basis. Before the provider applies each record, it generates a *BeforeUpdateRecord* event, which you can use to screen updates before they are applied. If an error occurs when updating a record, the provider receives an *OnUpdateError* event where it can resolve the error. Usually errors occur because the change violates a server constraint or the database record was changed by a different application subsequent to its retrieval by the provider, but prior to the client dataset's request to apply updates.

Update errors can be processed by either the provider or the client dataset. When the provider is part of a multi-tiered application, it should handle all update errors that do not require user interaction to resolve. When the provider can't resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it returns to the client dataset for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset based on the update status of records. By filtering the records, your event handler does not need to sort through records it won't be using. To filter the client dataset on the update status of its records, set its *StatusFilter* property.

Note Applications must supply extra support when the updates are directed at a dataset that does not represent a single table. For details on how to do this, see "Applying updates to datasets that do not represent a single table" on page 26-12.

Editing delta packets before updating the database

Before the provider applies updates to the database, it generates an *OnUpdateData* event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the *UpdateStatus* property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in Table 26.3.

Table 26.3 UpdateStatus values

Value	Description
usUnmodified	Record contents have not been changed
usModified	Record contents have been changed
usInserted	Record has been inserted
usDeleted	Record has been deleted

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

D

Delphi example

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
```

C++ example

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  for (DataSet->First();!DataSet->Eof;DataSet->Next())
  {
    if (DataSet->UpdateStatus == usInserted)
    {
      DataSet->Edit();
      DataSet->FieldByName("DateCreated")->AsDateTime = Date();
      DataSet->Post();
    }
  }
}
```

Influencing how updates are applied

The *OnUpdateData* event also gives your provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the *UpdateMode* property of the provider. *UpdateMode* can be assigned any of the following values:

Table 26.4 UpdateMode values

Value	Meaning
upWhereAll	All fields are used to locate fields (the WHERE clause).
upWhereChanged	Only key fields and fields that are changed are used to locate records.
upWhereOnly	Only key fields are used to locate records.

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause, and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the field's

ProviderFlags property. *ProviderFlags* is a set that can include any of the values in Table 26.5

Table 26.5 ProviderFlags values

Value	Description
pfInWhere	The field appears in the WHERE clause of generated INSERT, DELETE, and UPDATE statements when UpdateMode is upWhereAll or upWhereChanged.
pfInUpdate	The field can appear in the UPDATE clause of generated UPDATE statements.
pfInKey	The field is used in the WHERE clause of generated statements when UpdateMode is upWhereKeyOnly.
pfHidden	The field is included in records to ensure uniqueness, but can't be seen or used on the client side.

Thus, the following *OnUpdateData* event handler allows the TITLE field to be updated and uses the EMPNO and DEPT fields to locate the desired record. If an error occurs, and a second attempt is made to locate the record based only on the key, the generated SQL looks for the EMPNO field only:

Delphi example

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
```

C++ example

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  DataSet->FieldByName("EMPNO")->ProviderFlags.Clear();
  DataSet->FieldByName("EMPNO")->ProviderFlags << pfInWHere << pfInKey;
  DataSet->FieldByName("TITLE")->ProviderFlags.Clear();
  DataSet->FieldByName("TITLE")->ProviderFlags << pfInUpdate;
  DataSet->FieldByName("DEPT")->ProviderFlags.Clear();
  DataSet->FieldByName("DEPT")->ProviderFlags << pfInWhere;
}
```

Note You can use the *ProviderFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

Screening individual updates

Immediately before each update is applied, the provider receives a *BeforeUpdateRecord* event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets. For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that an update has been handled already and should not be applied. The provider then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can't be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

Resolving update errors on the provider

When an error condition arises as the provider tries to post a record in the delta packet, an *OnUpdateError* event occurs. If the provider can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client dataset for further reconciliation.

In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the *NewValue*, *OldValue*, and *CurValue* properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

Applying updates to datasets that do not represent a single table

When a provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. Obtaining this table name can be a problem if the provider is applying updates to the data underlying a stored procedure or a multi-table query. There is no easy way to ascertain the name of the table to which updates should be applied.

You can supply the table name programmatically in an *OnGetTableName* event handler. Once an event handler supplies the table name, the provider can generate appropriate SQL statements to apply updates.

Note Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the *BeforeUpdateRecord* event of the provider. Once this event handler has applied an update, you can set its *Applied* parameter to true so that the provider does not generate an error.

Responding to client-generated events

Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the *OnDataRequest* event.

OnDataRequest is not part of the normal functioning of the provider. It is simply a hook to allow client datasets to communicate directly with providers. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, an application calls the *DataRequest* method of the client dataset.

Chapter

27

Using Web Services to create multi-tiered database applications

<~TOPIC<~HEAD
^# FHXR29060
^A CreatingAMultiTieredApplication;CreatingMultiTieredApplications
^X
DesigningDatabaseApplications;UnderstandingDatasets;UsingDataControls;WorkingWithFieldComponents;CreatingAndUsingAClientDataset;ConnectingToDatabases;UsingUniDirectionalDatasets;UsingProviderComponents;CreatingAMultiTieredApplication;UsingXMLInDatabaseApplications
^$ Creating multi-tiered applications: Overview
^+ MULTI:000
^K multi-tiered applications;client applications:multi-tiered;server applications:multi-tiered;applications:client/server;application servers,
^C 2 Creating multi-tiered applications;3
^T DBD_Multitiered
HEAD~>Creating multi-tiered applications

This chapter describes how to create a multi-tiered, client/server database application. A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many <~JMP benefits ~MultiAdvantagesOfTheMultiTieredDatabaseModel JMP~>, such as centralized business logic and thin client applications.

In its simplest form, sometimes called the “three-tiered model,” a multi-tiered application is partitioned into thirds:

- **Client application**: provides a user interface on the user’s machine.
- **Application server**: resides in a central networking location accessible to all clients and provides common data services.

- **Remote database server**: provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a "data broker." There are wizards and components designed to ease the process of writing the application server and its clients. If you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on other platforms.

Support for developing multi-tiered applications is an extension of the way client datasets communicate with a provider component using transportable data packets. See <~JMP Understanding multi-tiered database applications ~MultiUnderstandingProviderBasedMultiTieredApplications JMP~> for an overview of this technology and the architecture of a typical three-tiered application.This chapter focuses on creating a three-tiered database application. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

For information about creating the middle tier, see <~JMP Creating the application server ~MultiCreatingTheApplicationServer JMP~>. For information about creating the client tier, see <~JMP Creating the client application ~MultiCreatingTheClientApplication JMP~>.

TOPIC~>

Advantages of the multi-tiered database model

<~TOPIC<~HEAD
^# MultiAdvantagesOfTheMultiTieredDatabaseModel
^A AdvantagesOfTheMultiTieredDatabaseModel
^X UnderstandingMIDASTechnology;BuildingAMultiTieredApplication;
^$ Advantages of the multi-tiered database model
^+ MULTI:000
^K multi-tiered applications, advantages
^C 3
^T DBD_MULTITIERED
HEAD~>Advantages of the multi-tiered database model

The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the database server.

The advantages of this multi-tiered model include the following:

- **Encapsulation of business logic in a shared middle tier**. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.
- **Thin client applications**. Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier for you to deploy because you don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the database server's client-side software). Thin client applications can be distributed over the Internet for additional flexibility.
- **Distributed data processing**. Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- **Increased opportunity for security**. You can isolate sensitive functionality into tiers that have different access restrictions. This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily.

TOPIC~>

Understanding multi-tiered database applications

<~TOPIC<~HEAD
^# MultiUnderstandingProviderBasedMultiTieredApplications
^A
UnderstandingMIDASTechnology;UnderstandingProviderBasedMultiTieredApplic
ations
^X
AdvantagesOfTheMultiTieredDatabaseModel;BuildingAMultiTieredApplication;
^$ Understanding multi-tiered database applications
^+ MULTI:000
^K multi-tiered applications;providers, multi-tiered applications
^C 3
^T DBD_MULTITIERED
HEAD~>Understanding multi-tiered database applications

The components on the <~JMP WebServices page ~
!Alink(CompPaletteWebServices,1) JMP~> and the<~JMP Data Access page ~
!Alink(DataAccessComponents,1) JMP~> of the component palette support the development of multi-tiered applications. In addition, a SOAP data module is created by a wizard on the Web Services page of the New Items dialog.

Components used in multi-tiered applications are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets. The components needed for a multi-tiered application are described in Table 27.1the following table:

Table 27.1 Components used in multi-tiered applications

Component	Description
SOAP data module	A specialized data module that acts as a SOAP-based Web Service to give client applications access to any providers it contains. Used on the application server.
Provider component	A data broker that provides data by creating data packets and resolves client updates. Used on the application server.
Client dataset component	A specialized dataset that uses midas.so or midaslib.dcu to manage data stored in data packets. The client dataset is used in the client application. It caches updates locally, and applies them in delta packets to the application server.
SOAP connection component	A specialized component that locates the server, generates SOAP-based calls to its *IAppServerSOAP* interface, and uses that to implement an *IAppServer* interface that it makes available to client datasets.

The provider and client dataset components require midas.so or midaslib.dcu, which manages datasets stored as data packets. (Note that, because the provider is used on the application server and the client dataset is used on the client application, if you are using midas.so, you must deploy it on both application server and client application.)

An overview of the architecture into which these components fit is described in "Using a multi-tiered architecture" on page 19-12.<~JMP Using a multi-tiered architecture ~DBDesignUsingAMultiTieredArchitecture JMP~>. For more information on how these components fit together to create a multi-tiered application, see

- <~JMP Overview of a three-tiered application ~MultiOverviewOfAThreeTieredApplication JMP~>
- <~JMP The structure of the client application~MultiTheStructureOfTheClientApplication JMP~>
- <~JMP The structure of the application server ~MultiTheStructureOfTheApplicationServer JMP~>

TOPIC~>

Overview of a three-tiered application

<~TOPIC<~HEAD
^# MultiOverviewOfAThreeTieredApplication
^A
AStructuralOverviewOfAMultiTieredApplication;OverviewOfAMIDASBasedMultiTieredApplication;OverviewOfAThreeTieredApplication
^X
TheStructureOfTheClientApplication;TheStructureOfTheApplicationServer;UsingAMultiTieredArchitecture

^$ Overview of a three-tiered application
^+ MULTI:000
^K multi-tiered applications, overview
^T DBD_MULTITIERED
^C 3
^T DBD_MULTITIERED
HEAD~> Overview of a three-tiered application

The following numbered steps illustrate a normal sequence of events for a provider-based three-tiered application:

1. A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServerSOAP* interface from the application server.
2. The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand).
3. The application server retrieves the data (first establishing a database connection, if necessary), packages it for the client, and returns a data packet to the client. Additional information, (for example, field display characteristics) can be included in the metadata of the data packet. This process of packaging data into data packets is called "providing."
4. The client decodes the data packet and displays the data to the user.
5. As the user interacts with the client application, the data is updated (records are added, deleted, or modified). These modifications are stored in a change log by the client.
6. Eventually the client applies its updates to the application server, usually in response to a user action. To apply updates, the client packages its change log and sends it as a data packet to the server.
7. The application server decodes the package and posts updates (in the context of a transaction if appropriate). If a record can't be posted (for example, because another application changed the record after the client requested it and before the client applied its updates), the application server either attempts to reconcile the client's changes with the current data, or saves the records that could not be posted. This process of posting records and caching problem records is called "resolving."
8. When the application server finishes the resolving process, it returns any unposted records to the client for further resolution.
9. The client reconciles unresolved records. There are many ways a client can reconcile unresolved records. Typically the client attempts to correct the situation that prevented records from being posted or discards the changes. If the error situation can be rectified, the client applies updates again.
10. The client refreshes its data from the server.

TOPIC~>

The structure of the client application

<~TOPIC<~HEAD
^# MultiTheStructureOfTheClientApplication
^A
TheStructureOfTheClientApplication;TheStructureOfTheMIDASClientApplication
^X
OverviewOfAThreeTieredApplication;TheStructureOfTheApplicationServer;CreatingTheClientApplication
^$ The structure of the client application
^+ MULTI:000
^K client applications, multi-tiered;connection components
^C 3
^T DBD_MULTITIERED
HEAD~> The structure of the client application

To the end user, the client application of a multi-tiered application looks and behaves no differently than a traditional two-tiered application that uses cached updates. User interaction takes place through standard data-aware controls that display data from a *TClientDataSet* component. For detailed information about using the properties, events, and methods of client datasets, see <~JMP Using client datasets~!Alink(CreatingAndUsingAClientDataset,1) JMP~>.Chapter 25, “Using client datasets.”

TClientDataSet fetches data from and applies updates to a provider component, just as in two-tiered applications that use a client dataset with an external provider. For details about providers, see Chapter 26, “Using provider components”<~JMP Using provider components ~ProviderUsingProviderComponents JMP~>. For details about client dataset features that facilitate its communication with a provider, see “Using a client dataset with a provider” on page 25-27<~JMP Using a client dataset with a provider ~5Ds3UsingAClientDataSetWithADataProvider JMP~>

The client dataset communicates with the provider through the *IAppServer* interface. It gets this interface from a *TSoapConnection* component, which establishes the connection to the application server. *TSoapConnection* uses SOAP as the protocol for communicating with the application server. SOAP marshals method calls using an XML encoding. SOAP connections use HTTP as a transport protocol.

SOAP connections work in cross-platform applications because they are supported on both Windows and Linux. HTTP provides a lowest common denominator that you know is available on all clients, and clients can communicate with an application server that is protected by a “firewall”. For more information about using SOAP to distribute applications, see Chapter 33, “Using Web Services.”<~JMP Using Web Services.~!Alink(UsingWebServices,1) JMP~>.

Note The Data Access page of the component palette includes a similar connection component that does not connect to an application server at all. This component, *TLocalConnection*, supplies an *IAppServer* interface for client datasets to use when communicating with providers in the same application. It is not required, but makes it easier to later scale up to a multi-tiered application.

For more information about using *TSoapConnection*, see <~JMP Connecting to the application server ~FHXR24933 JMP~>."Connecting to the application server" on page 27-16.

TOPIC~>

The structure of the application server

<~TOPIC<~HEAD
^# MultiTheStructureOfTheApplicationServer
^A TheStructureOfTheApplicationServer
^X
OverviewOfAMIDASBasedMultiTieredApplication;TheStructureOfTheClientApplication;CreatingTheApplicationServer;oocCodeGeneratedByWizards
^$ The structure of the application server
^+ MULTI:000
^K application servers,;server applications, application servers;remote data modules
^C 3
^T DBD_MULTITIERED
HEAD~> The structure of the application server

When you set up and run an application server, it does not establish any connection with client applications. Instead, connection is initiated and maintained by client applications. The client application uses its *TSoapConnection* component to establish a connection to the application server, which it uses to communicate with its selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

The basis of an application server is a remote data module, *TSoapDataModule*, which is a specialized data module that supports the *IAppServer*SOAP interface as an invokable interface. The *TSoapConnection* component uses this *IAppServerSOAP* interface to <~JMP communicate with providers~ProviderCommunicatingWithTheClientDataset JMP~> on the application server. When *TSoapDataModule* is included in a Web Service application, it is automatically registered to respond to incoming requests on the *IAppServerSOAP* interface.

Note SOAP connections limit you to a single remote data module in the application server. This is because the Web Service application can only associated a single object as the implementer for incoming *IAppServerSOAP* calls.

The contents of the remote data module

As with any data module, you can include any nonvisual component in the SOAP data module. There are certain components, however, that you must include:

- If the SOAP data module is exposing information from a database server, it must include a dataset component to represent the records from that database server. Other components, such as an <~JMP database connection component ~uniconnecConnectingToDatabases JMP~>, may be required to allow the dataset to interact with a database server. For information about datasets, see Chapter 22, "Understanding datasets." For information about database connection components, see Chapter 21, "Connecting to databases."

For every dataset that the SOAP data module exposes to clients, it must include a <~JMP dataset provider~ProviderUsingProviderComponents JMP~>. A dataset provider packages data into data packets that are sent to client datasets and applies updates received from client datasets back to a source dataset or a database server. For more information about dataset providers, see Chapter 26, "Using provider components."

- For every XML document that the remote data module exposes to clients, it must include an<~JMP XML provider~XMLDataUsingAnXMLDocumentWithAProvider JMP~>. An XML provider acts like a dataset provider, except that it fetches data from and applies updates to an XML document rather than a database server.

Note Do not confuse database connection components, which connect datasets to a database server, with the connection components used by client applications in a multi-tiered application. The *TSoapConnection* component used in multi-tiered applications can be found on the <~JMP WebServices page ~ !Alink(CompPaletteWebServices,1) JMP~> of the Component palette.

TOPIC~>

Creating the application server

<~TOPIC<~HEAD
^# MultiCreatingTheApplicationServer
^A CreatingTheApplicationServer
^X
CreatingTheClientApplication;RegisteringTheApplicationServer;TheStructureOfTheApplicationServer
^$ Creating the application server
^+ MULTI:000
^K application servers:creating;remote data modules, creating
^C 3
^T DBD_MULTITIERED
HEAD~>Creating the application server

You create an application server much as you create most Web Service applications. The major difference is that the application server uses a SOAP data module.

To create an application server, follow these steps:

1 Start a new project that is a Web Service application. Choose File | New | Other, and on the WebServices page of the new items dialog, choose SOAP Server application. Specify the type of Web Server you want to use. When prompted if you want to define a new interface for the SOAP module, say no. Save the new project.

2 Add a new SOAP data module to the project. From the main menu, choose File | New | Other, and on the WebServices page of the new items dialog, select SOAP

Data Module. For more detailed information about setting up a remote data module, see <~JMP Setting up the remote data module ~MultiSettingUpTheRemoteDataModule JMP~>"Setting up the remote data module" on page 27-10.

3 Place the appropriate dataset components on the data module and set them up to access the database server.

4 Place a *TDataSetProvider* component on the data module for each dataset you want to expose to clients. This provider is required for brokering client requests and packaging data. Set the *DataSet* property for each provider component to the name of the dataset to access. You can set additional properties for the provider. See <~JMP Using provider components ~ProviderUsingProviderComponents JMP~>Chapter 26, "Using provider components" for more detailed information about setting up a provider.

   If you are working with data from XML documents, you can use a *TXMLTransformProvider* component instead of a dataset and *TDataSetProvider* component. When using *TXMLTransformProvider*, set the *XMLDataFile* property to specify the XML document from which data is provided and to which updates are applied.

5 Write application server code to implement events, shared business rules, shared data validation, and shared security. When writing this code, you may want to

   - <~JMP Extend the application server's interface ~MultiExtendingTheInterfaceOfTheApplicationServer JMP~> to provide additional ways for the client application to call the server. See "Extending the application server's interface" on page 27-11.
   - <~JMP Create master/detail relationships ~MultiSupportingMasterDetailRelationships JMP~> between the datasets in your application server. Master/detail relationships are described in "Supporting master/detail relationships" on page 27-12.
   - <~JMP Ensure your application server is stateless~MultiSupportingStateInformationInRemoteDataModules JMP~>. Handling state information is described in "Supporting state information in remote data modules" on page 27-13.

6 Compile and save your application server. Because the application server uses SOAP, it must be a Web Service application. As such, it must be registered with your Web Server, so that it receives incoming HTTP messages. In addition, to allow clients to access any of the interfaces in your application, you will want to publish a WSDL document that describes the invokable interfaces in your application. For information about exporting a WSDL document for a Web Service application, see "Generating WSDL documents for a Web Service application" on page 33-18<~JMP Generating WSDL documents for a Web Service application~!Alink(GeneratingWSDLDocumentsForAWebServiceApplication, 1) JMP~>.

TOPIC~>

Setting up the remote data module

<~TOPIC<~HEAD
^# MultiSettingUpTheRemoteDataModule
^A
SettingUpTheRemoteDataModule;EstablishingAConnectionWithAClientApplicatio
n
^X
ManagingTransactionsInMultiTieredApplications;ExtendingTheApplicationServersI
nterface;MultiMasterDetail;SupportingStateInformationInRemotedataModules;
^$ Setting up the remote data module
^+ MULTI:000
^K remote data modules, setting up; remote data modules, creating;remote data
modules, SOAP
^C 3
^T DBD_MULTITIERED
HEAD~>Setting up the remote data module

SOAP data modules are more than simple data modules. They implement and register an invokable interface (*IAppServerSOAP*) and are designed to work in a Web Service application.

To create the SOAP data module, use the SOAP Server data module wizard. You can invoke this wizard by choosing File | New | Other and selecting SOAP Server Data Module from the WebServices page of the New items dialog. The SOAP data module wizard appears.

You must supply a class name for your SOAP data module. This is the base name of a *TSoapDataModule* descendant that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TSoapDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note The new *TSoapDataModule* object should be added to a Web Service application. The *IAppServerSOAP* interface is an invokable interface, which is registered in the initialization section of the new unit (Delphi) or in the generated *RegTypes* method of the new unit (C++). This allows the invoker component in the main Web module to forward all incoming calls to your data module.

{bmc IC_C16.bmp} If you are using C++, and you want your application server to respond to clients that were written using Kylix 2 or Delphi 6 (prior to update patch 2), you must add code to register the *IAppServer* interface. Locate the generated *RegTypes* method where it registers your *IAppServerSOAP* descendant. Immediately after that registration call, add a call to the global <~JMP *RegDefIAppServerInvClass* function ~!Alink(RegDefIAppServerInvClass_function,1) JMP ~>, which registers the data module as the implementation for *IAppServer*. (This step is not necessary if you are writing your application server in Delphi.)

You may want to edit the definitions of the generated interface and *TSoapDataModule* descendant, adding your own properties and methods. These properties and methods are not called automatically, but client applications that request your new interface by name can use any of the properties and methods that you add. For information about adding properties and methods to the generated interface, see "Extending the application server's interface" on page 27-11<~~JMP Extending the application server's interface ~MultiExtendingTheInterfaceOfTheApplicationServer JMP~>.

TOPIC~>

Extending the application server's interface

<~TOPIC<~~HEAD
^# MultiExtendingTheInterfaceOfTheApplicationServer
^A
ExtendingTheInterfaceOfTheApplicationServer;ExtendingTheApplicationServersInterface
^X
ManagingTransactionsInMultiTieredApplications;ExtendingTheApplicationServersInterface;MultiMasterDetail;SupportingStateInformationInRemotedataModules;SettingUpTheRemoteDataModule;
^$ Extending the interface of the application server
^+ MULTI:000
^K interfaces, remote data modules;remote data modules, interfaces
^C 3
^T DBD_MULTITIERED
HEAD~>Extending the interface of the application server

Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server.

You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServerSOAP* and is created for you automatically by the wizard when you <~~JMP create the remote data module ~MultiSettingUpTheRemoteDataModule JMP~>.

The *IAppServerSOAP* descendant that the wizard creates introduces no new properties or methods. To extend this interface, edit the declaration of the interface that appears in the interface section of the unit (Delphi) or the header file for the unit (C++).

Once you have added to your remote data module's interface, you must make the same changes to the new *TSoapDataModule* descendant. Change the declaration in the interface section of the unit (Delphi) or its header file (C++). Then, in the implementation section (Delphi) or the .cpp file (C++), write the implementations for any methods that you added.

Note You can't use callbacks in your interface extensions.

Client applications call your interface extensions using a remote interfaced object. For information on how to do this, see "Calling invokable interfaces" on page 33-20<~~JMP Calling invokable interfaces ~!Alink(CallingInvokableInterfaces,1) JMP~>.

TOPIC~>

Supporting master/detail relationships

<~TOPIC<~HEAD
^# MultiSupportingMasterDetailRelationships
^A SupportingMasterDetailRelationships;MultiMasterDetail
^X
ManagingTransactionsInMultiTieredApplications;ExtendingTheApplicationServersInterface;MultiMasterDetail;SupportingStateInformationInRemotedataModules;SettingUpTheRemoteDataModule;
^$ Supporting master/detail relationships
^+ MULTI:000
^K master/detail relationships, multi-tiered applications;nested tables: multi-tiered applications;multi-tiered applications:master/detail relationships
^C 3
^T DBD_MULTITIERED
HEAD~>Supporting master/detail relationships

You can create master/detail relationships between client datasets in your client application in the same way you set them up using any table-type dataset. For more information about setting up master/detail relationships in this way, see "Representing master/detail relationships" on page 25-11<~JMP Representing master/detail relationships ~5Ds3SupportingMasterDetailRelationships JMP~>.

However, this approach has two major drawbacks:

- The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. (This problem can be mitigated by using parameters. For more information, see "Limiting records with parameters" on page 25-30<~JMP Limiting records with parameters ~5Ds3LimitingRecordsWithParameters JMP~>.)
- It is very difficult to apply updates, because client datasets apply updates at the dataset level and master/detail updates span multiple datasets. Even in a two-tiered environment, where you can use the database connection component to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this when providing from datasets, set up a master/detail relationship between the datasets on the application server. Then set the *DataSet* property of your provider component to the master table. To use nested tables to represent master/detail relationships when providing from XML documents, use a transformation file that defines the nested detail sets.

When clients call the *GetRecords* method of the provider, it automatically includes the detail dataset as a DataSet field in the records of the data packet. When clients call the *ApplyUpdates* method of the provider, it automatically handles applying updates in the proper order.

TOPIC~>

Supporting state information in remote data modules

<~TOPIC<~HEAD
^# MultiSupportingStateInformationInRemotedataModules
^A
SupportingMTSJustInTimeActivation;SupportingStatelessMTSDataModules;SupportingStateInformationInRemotedataModules
^X
ManagingTransactionsInMultiTieredApplications;ExtendingTheApplicationServersInterface;MultiMasterDetail;SupportingStateInformationInRemotedataModules;SettingUpTheRemoteDataModule;UsingtheIAppServerInterface;
^$ Supporting state information in remote data modules
^+ MULTI:000
^K multi-tiered applications, state information;data modules, persistent states;incremental fetching, stateless data modules
^C 3
^T DBD_MULTITIERED
HEAD~>Supporting state information in remote data modules

SOAP data modules do not provide any mechanism for preserving state information between client calls. If your application server has multiple clients, you must manage the code you write to take into account that the application server does not "remember" anything that happened in previous calls by the client. The *IAppServerSOAP* interface is designed to work in such a stateless environment.

However, there are times when you want to maintain state information between calls to the application server. For example, when requesting data using incremental fetching, the provider on the application server must "remember" information from previous calls (the current record).

Before and after any calls to the *IAppServer* interface that the client dataset makes (AS_*ApplyUpdates*, AS_*Execute*, AS_*GetParams*, AS_*GetRecords*, or AS_*RowRequest*), it receives an event where it can send or retrieve custom state information. Similarly, before and after providers respond to these client-generated calls, they receive events where they can retrieve or send custom state information. Using this mechanism, you can communicate persistent state information between client applications and the application server, even if the application server is stateless.

For example, consider a dataset that represents the following parameterized query:

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

To enable incremental fetching in a stateless application server, you can do the following:

- When the provider packages a set of records in a data packet, it notes the value of CUST_NO on the last record in the packet:

D {bmc IC_D16.bmp}

Delphi example

```
TMyAppServer.DataSetProvider1GetData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
  DataSet.Last; { move to the last record }
  with Sender as TDataSetProvider do
```

```
    Tag := DataSet.FieldValues['CUST_NO']; {save the value of CUST_NO }
end;
```

{bmc IC_C16.bmp}

C++ example

```
TMyAppServer::DataSetProvider1GetData(TObject *Sender, TCustomClientDataSet *DataSet)
{
  DataSet->Last(); // move to the last record
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  pProvider->Tag = DataSet->FieldValues["CUST_NO"];
}
```

- The provider sends this last CUST_NO value to the client after sending the data packet:

{bmc IC_D16.bmp}

Delphi example

```
TMyAppServer.DataSetProvider1AfterGetRecords(Sender: TObject;
                   var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
    OwnerData := Tag; {send the last value of CUST_NO }
end;
```

{bmc IC_C16.bmp}

C++ example

```
TMyAppServer::DataSetProvider1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  OwnerData = pProvider->Tag;
}
```

- On the client, the client dataset saves this last value of CUST_NO:

{bmc IC_D16.bmp}

Delphi example

```
TDataModule1.ClientDataSet1AfterGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
    Tag := OwnerData; {save the last value of CUST_NO }
end;
```

{bmc IC_C16.bmp}

C++ example

```
TDataModule1::ClientDataSet1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pDS = dynamic_cast<TComponent *>(Sender);
  pDS->Tag = OwnerData;
}
```

- Before fetching a data packet, the client sends the last value of CUST_NO it received:

{bmc IC_D16.bmp}

Delphi example

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
  begin
    if not Active then Exit;
    OwnerData := Tag; { Send last value of CUST_NO to application server }
  end;
end;
```

{bmc IC_C16.bmp}

C++ example

```
TDataModule1::ClientDataSet1BeforeGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TClientDataSet *pDS = dynamic_cast<TClientDataSet *>(Sender);
  if (!pDS->Active)
    return;
  OwnerData = pDS->Tag;
}
```

- Finally, on the server, the provider uses the last CUST_NO sent as a minimum value in the query:

{bmc IC_D16.bmp}

Delphi example

```
TMyAppServer.DataSetProvider1BeforeGetRecords(Sender: TObject;
                    var OwnerData: OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
      begin
        Params.ParamValues['MinVal'] := OwnerData;
        Refresh; { force the query to reexecute }
    end;
end;
```

{bmc IC_C16.bmp}

C++ example

```
TMyAppServer::DataSetProvider1BeforeGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  if (!VarIsEmpty(OwnerData))
  {
    TDataSetProvider *pProv = dynamic_cast<TDataSetProvider *>(Sender);
    TSQLDataSet *pDS = (dynamic_cast<TSQLDataSet *>(pProv->DataSet);
    pDS->Params->ParamValues["MinVal"] = OwnerData;
    pDS->Refresh(); // force the query to reexecute
  }
}
```

TOPIC~>

Creating the client application

<~TOPIC<~HEAD
^# MultiCreatingTheClientApplication
^A CreatingTheClientApplication
^X CreatingTheApplicationServer;RegisteringTheApplicationServer;
^$ Creating the client application
^+ MULTI:000
^K clients:multi-tiered applications;multi-tiered applications:creating;client

applications:creating
^C 3
^T DBD_MULTITIERED
HEAD~>Creating the client application

In most regards, creating a multi-tiered client application is similar to creating a two-tiered client that uses a client dataset to cache updates. The major difference is that a multi-tiered client uses a connection component to establish a conduit to the application server.

To create a multi-tiered client application, start a new project and follow these steps:

1. Add a new data module object to the project.
2. Place a *TSoapConnection* component on the data module and set properties to indicate how to connect to the application server. See <~JMP Connecting to the application server ~FHXR24933 JMP~>"Connecting to the application server" on page 27-16 for details.
3. Write any code to control when to establish or break the connection. See <~JMP Managing server connections ~MultiManagingServerConnections JMP~>"Managing server connections" on page 27-18 for details.
4. Place as many *TClientDataSet* components as needed on the data module, and set the *RemoteServer* property for each component to the name of the connection component you placed in Step 2. For a full introduction to client datasets, see <~JMP Using client datasets ~FHXR30623B JMP~>.Chapter 25, "Using client datasets."
5. Set the *ProviderName* property for each *TClientDataSet* component. If your connection component is connected to the application server at design time, you can choose available application server providers from the *ProviderName* property's drop-down list.
6. Continue in the same way you would create any other database application. There are a few additional features available to clients of multi-tiered applications:
   - Your application may want to make direct calls to the application server's interface. "Calling server interfaces" on page 27-19<~JMP Calling server interfaces ~MultiCallingServerInterfaces JMP~> describes how to do this.
   - You may want to use the special features of client datasets that support their interaction with the provider components. These are described in "Using a client dataset with a provider" on page 25-27<~JMP Using a client dataset with a provider ~5Ds3UsingAClientDataSetWithADataProvider JMP~>.

TOPIC~>

Connecting to the application server

<~TOPIC<~HEAD
^# FHXR24933
^A
EstablishingAndMaintainingAConnectionToAnApplicationServer;ConnectingToThe

ApplicationServer
^X
TheStructureOfTheClientApplication;ManagingServerConnections;UsingAClientDataSetWithADataProvider;CallingServerInterfaces
^$ Connecting to the application server
^+ MULTI:000
^K connection components;client applications, connecting to servers
^C 3
^T DBD_MULTITIERED
HEAD~>Connecting to the application server

To establish and maintain a connection to an application server, a client application uses one or more <~JMP *TSoapConnection* ~!Alink(TSoapConnection_object,1) JMP~> components. You can find this component on the WebServices page of the Component palette. *TSoapConnection* uses HTTP as a transport protocol. Thus, you can use *TSoapConnection* from any machine that has a TCP/IP address, and it can take advantage of SSL security to communicate with a server that is protected by a firewall.

The SOAP connection component establishes a connection to a Web server application that implements the *IAppServerSOAP* or *IAppServer* interface as a Web Service. If possible, you want to use *IAppServerSOAP*, which works better as a Web Service. However, if the server was built using Kylix 2 or Delphi 6 (prior to update patch 2), *IAppServerSOAP* is not available. In that case, you must tell the connection component to look for *IAppServer* instead by setting the <~JMP *UseSoapAdapter* ~!Alink(TSoapConnection_UseSOAPAdapter,1) JMP~> property to false. Regardless of which interface it uses to communicate with the application server, *TSoapConnection* makes an *IAppServer* interface available to the client dataset that uses it.

TSoapConnection locates this Web Server application using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server, the name of the Web Service application, and a path that matches the path name of the *THTTPSoapDispatcher* on the application server. Specify this value using the <~JMP *URL* ~!Alink(TSoapConnection_URL,1) JMP~> property.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the <~JMP *UserName* ~!Alink(TSoapConnection_UserName,1) JMP~> and <~JMP *Password* ~!Alink(TSoapConnection_Password,1) JMP~> properties so that the connection component can log on.

Usually the application server is on a different machine from the client application, but even if the server resides on the same machine as the client application (for example, during the building and testing of the entire multi-tier application), you can still use the SOAP connection component to identify the application server's URL and use its interface.

TOPIC~>

Managing server connections

<~TOPIC<~HEAD
^# MultiManagingServerConnections
^A ManagingServerConnections
^X
ConnectingToTheApplicationServer;TheStructureOfTheClientApplication;UsingAClientDataSetWithADataProvider
^$ Managing server connections
^+ MULTI:000
^K remote connections:managing;connections:client applications;client applications, managing connections;connection components, managing connections
^C 3
^T DBD_MULTITIERED
HEAD~>Managing server connections
The main purpose of the SOAP connection component is to locate and connect to the application server.

The following topics describe how to use a *TSoapConnection* component for

- <~JMP Connecting to the server ~FHXR34341 JMP~>.
- <~JMP Dropping or changing a server connection ~MultiDroppingOrChangingAServerConnection JMP~>.

TOPIC~>

Connecting to the server

<~TOPIC<~HEAD
^# FHXR34341
^A EstablishingAConnectionToTheServer;ConnectingToTheServer
^X DroppingOrChangingAServerConnection;CallingServerInterfaces
^$ Connecting to the server
^+ MULTI:000
^K client applications, connecting to servers;application servers, opening connections;connection components, opening connections
^C 3
^T DBD_MULTITIERED
HEAD~>Connecting to the server
To locate and connect to the application server, you must first set the properties of the connection component to identify the application server and supply any connection information. This process is described in "Connecting to the application server" on page 27-16<~JMP Connecting to the application server ~FHXR24933 JMP~>. Before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component.

The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to true opens the connection, as long as the *RemoteServer* property has been set.

If you do not link any client datasets to the *TSoapConnection* component, you can open the connection by setting its *Connected* property to true.

Before *TSoapConnection* establishes a connection to an application server, it generates a *BeforeConnect* event. You can perform any special actions prior to connecting in a *BeforeConnect* handler that you code. After establishing a connection, the connection component generates an *AfterConnect* event for any special actions.

TOPIC~>

Dropping or changing a server connection

<~TOPIC<~HEAD
^# MultiDroppingOrChangingAServerConnection
^A DroppingAServerConnection;DroppingOrChangingAServerConnection
^X ConnectingToTheServer;CallingServerInterfaces;
^$ Dropping or changing a server connection
^+ MULTI:000
^K client applications, dropping connections;client applications, changing connections;connection components, dropping connections;connection components, changing connections
^C 3
^T DBD_MULTITIERED
HEAD~>Dropping or changing a server connection
TSoapConnection drops a connection to the application server when you

- set the *Connected* property to false.
- free the *TSoapConnection* component. A connection object is automatically freed when a user closes the client application.
- change the *URL* property of the *TSoapConnection* component. Changing this property allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one.

Note Instead of using a single connection component to switch among available application servers, a client application can instead have more than one connection component, each of which is connected to a different application server.

Before a connection component drops a connection, it automatically calls its *BeforeDisconnect* event handler, if one is provided. To perform any special actions prior to disconnecting, write a *BeforeDisconnect* handler. Similarly, after dropping the connection, the *AfterDisconnect* event handler is called. If you want to perform any special actions after disconnecting, write an *AfterDisconnect* handler.

TOPIC~>

Calling server interfaces

<~TOPIC<~HEAD
^# MultiCallingServerInterfaces
^A CallingServerInterfaces

^X ConnectingToTheServer;DroppingOrChangingAServerConnection;
^$ Calling server interfaces
^+ MULTI:000
^K client applications, calling server interfaces;remote data modules, interfaces;AppServer, using to call interfaces;interfaces, remote data modules
^C 3
^T DBD_MULTITIERED
HEAD~>Calling server interfaces
Applications do not need to call the *IAppServerSOAP* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServerSOAP* interface, you may have added your own extensions to the remote data module's interface. When you <~JMP extend the application server's interface ~MultiExtendingTheInterfaceOfTheApplicationServer JMP~>, you need a way to call those extensions. For information about extending the application server's interface, see "Extending the application server's interface" on page 27-11.

To call your extensions, you must use a remote interfaced object (<~JMP *THTTPRio*~!Alink(THTTPRio_object,1) JMP~>) which makes early-bound calls. As with all early-bound calls, the client application must know the application server's interface declaration at compile time. Add this to your client application referencing a WSDL document that describes the interface. For information on importing a WSDL document that describes the server interface, see "Importing WSDL documents" on page 33-19<~JMP Importing WSDL documents ~!Alink(ImportingWSDLDocuments,1) JMP~>.

Note The unit that declares the server interface must also register it with the invocation registry. For details on invokable interfaces and how to register them, see "Understanding invokable interfaces" on page 33-2<~JMP Understanding invokable interfaces ~!Alink(UnderstandingInvokableInterfaces,1) JMP~>.

Once you have imported a WSDL document to generate a unit that declares and registers the application server's interface, create an instance of *THTTPRio* for that interface:

D **{bmc IC_D16.bmp}**

```
X := THTTPRio.Create(nil); { in Delphi }
```

C++ **{bmc IC_C16.bmp}**

```
THTTPRio *X = new THTTPRio(NULL); // in C++
```

Next, assign the URL that your connection component uses to the remote interfaced object, appending the name of the interface you want to call:

D **{bmc IC_D16.bmp}**

```
X.URL := SoapConnection1.URL + 'IMyInterface'; { in Delphi }
```

C++ **{bmc IC_C16.bmp}**

```
X->URL = SoapConnection1->URL + "IMyInterface"; // in C++
```

{bmc IC_D16.bmp} In Delphi, you can then use the **as** operator to cast the instance of *THTTPRio* to the application server's interface:

```
InterfaceVariable := X as IMyAppServer;
InterfaceVariable.SpecialMethod(x,y);
```

{bmc IC_C16.bmp} In C++, use the *QueryInterface* method to obtain that interface:

```
_di_IMyAppServer InterfaceVariable;
if (X->QueryInterface(InterfaceVariable) == S_OK)
{
  InterfaceVariable->SpecialMethod(x,y);
}
```

TOPIC~>

Chapter

28

Using XML in database applications

In addition to the support for connecting to database servers, the CLX database architecture lets you work with XML documents as if they were database servers. XML (Extensible Markup Language) is a markup language for describing structured data. XML documents provide a standard, transportable format for data that is used in Web applications, business-to-business communication, and so on. For information on support for working directly with XML documents, see Chapter 32, "Working with XML documents."

Support for working with XML documents in database applications is based on a set of components that can convert data packets (the *Data* property of a client dataset) into XML documents and convert XML documents into data packets. To use these components, you must first define the transformation between the XML document and the data packet. Once you have defined the transformation, you can use special components to

- convert XML documents into data packets.
- provide data from and resolve updates to an XML document.
- use an XML document as the client of a provider.

Defining transformations

Before you can convert between data packets and XML documents, you must define the relationship between the metadata in a data packet and the nodes of the corresponding XML document. A description of this relationship is stored in a special XML document called a transformation.

Each transformation file contains two things: the mapping between the nodes in an XML schema and the fields in a data packet, and a skeletal XML document that represents the structure for the results of the transformation. A transformation is a one-way mapping: from an XML schema or document to a data packet or from the metadata in a data packet to an XML schema. Often, you create transformation files in pairs: one that maps from XML to data packet, and one that maps from data packet to XML.

In order to create the transformation files for a mapping, use the XMLMapper utility that ships in the bin directory.

Mapping between XML nodes and data packet fields

XML provides a text-based way to store or describe structured data. Datasets provide another way to store and describe structured data. To convert an XML document into a dataset, therefore, you must identify the correspondences between the nodes in an XML document and the fields in a dataset.

Consider, for example, an XML document that represents a set of email messages. It might look like the following (containing a single message):

```
<?xml version="1.0" standalone="yes" ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
   <body>
      <subject>XML components</subject>
      <content>
        Joe,
        Attached is the specification for the XML component support in CLX.
        This looks like a good solution to our buisness-to-buisness application!
        Also attached, please find the project schedule. Do you think its reasonable?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
   </body>
</email>
```

One natural mapping between this document and a dataset would map each e-mail message to a single record. The record would have fields for the sender's name and address. Because an e-mail message can have multiple recipients, the recipient (<to>

would map to a nested dataset. Similarly, the cc list maps to a nested dataset. The subject line would map to a string field while the message itself (<content>) would probably be a memo field. The names of attachment files would map to a nested dataset because one message can have several attachments. Thus, the e-mail above would map to a dataset something like the following:

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(DataSet)	(DataSet)	XML components	(MEMO)	(DataSet)

where the nested dataset in the “To” field is

Name	Address
Joe Engineer	jengineer@MyCo.Com

the nested dataset in the “CC” field is

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

and the nested dataset in the “Attach” field is

Attachfile
XMLSpec.txt
Schedule.txt

Defining such a mapping involves identifying those nodes of the XML document that can be repeated and mapping them to nested datasets. Tagged elements that have values and appear only once (such as <content>...</content>) map to fields whose datatype reflects the type of data that can appear as the value. Attributes of a tag (such as the AttachFile attribute of the attachment tag) also map to fields.

Note that not all tags in the XML document appear in the corresponding dataset. For example, the <head>...<head/> element has no corresponding element in the resulting dataset. Typically, only elements that have values, elements that can be repeated, or the attributes of a tag map to the fields (including nested dataset fields) of a dataset. The exception to this rule is when a parent node in the XML document maps to a field whose value is built up from the values of the child nodes. For example, an XML document might contain a set of tags such as

```
<FullName>
   <Title> Mr. </Title>
   <FirstName> John </FirstName>
   <LastName> Smith </LastName>
</FullName>
```

which could map to a single dataset field with the value

```
Mr. John Smith
```

Using XMLMapper

The XML mapper utility, xmlmapper, lets you define mappings in three ways:

- From an existing XML schema (or document) to a client dataset that you define. This is useful when you want to create a database application to work with data for which you already have an XML schema.
- From an existing data packet to a new XML schema you define. This is useful when you want to expose existing database information in XML, for example to create a new business-to-business communication system.
- Between an existing XML schema and an existing data packet. This is useful when you have an XML schema and a database that both describe the same information and you want to make them work together.

Once you define the mapping, you can generate the transformation files that are used to convert XML documents to data packets and to convert data packets to XML documents. Note that only the transformation file is directional: a single mapping can be used to generate both the transformation from XML to data packet and from data packet to XML.

Note XML mapper relies on the midas.so shared library to work correctly. Be sure that you have this library installed before you try to use xmlmapper.

Loading an XML schema or data packet

Before you can define a mapping and generate a transformation file, you must first load descriptions of the XML document and the data packet between which you are mapping.

You can load an XML document or schema by choosing File|Open and selecting the document or schema in the resulting dialog.

You can load a data packet by choosing File|Open and selecting a data packet file in the resulting dialog. (The data packet is simply the file generated when you call a client dataset's *SaveToFile* method.)

You can load only an XML document or schema, only a data packet, or you can load both. If you load only one side of the mapping, XML mapper can generate a natural mapping for the other side.

Defining mappings

The mapping between an XML document and a data packet need not include all of the fields in the data packet or all of the tagged elements in the XML document. Therefore, you must first specify those elements that are mapped. To specify these elements, first select the Mapping page in the central pane of the dialog.

To specify the elements of an XML document or schema that are mapped to fields in a data packet, select the Sample or Structure tab of the XML document pane and double-click on the nodes for elements that map to data packet fields.

To specify the fields of the data packet that are mapped to tagged elements or attributes in the XML document, double-click on the nodes for those fields in the Datapacket pane.

If you have only loaded one side of the mapping (the XML document or the data packet), you can generate the other side after you have selected the nodes that are mapped.

- If you are generating a data packet from an XML document, you first define attributes for the selected nodes that determine the types of fields to which they correspond in the data packet. In the center pane, select the Node Repository page. Select each node that participates in the mapping and indicate the attributes of the corresponding field. If the mapping is not straightforward (for example, a node with subnodes that corresponds to a field whose value is built from those subnodes), check the User Defined Translation check box. You will need to write an event handler later to perform the transformation on user defined nodes.

  Once you have specified the way nodes are to be mapped, choose Create | Datapacket from XML. The corresponding data packet is automatically generated and displayed in the Datapacket pane.

- If you are generating an XML document from a data packet, choose Create | XML from Datapacket. A dialog appears where you can specify the names of the tags and attributes in the XML document that correspond to fields, records, and datasets in the data packet. For field values, the way you name them indicates whether they map to a tagged element with a value or to an attribute. Names that begin with an @ symbol map to attributes of the tag that corresponds to the record, while names that do not begin with an @ symbol map to tagged elements that have values and that are nested within the element for the record.
- If you have loaded both an XML document and a data packet (client dataset file), be sure you select corresponding nodes in the same order. The corresponding nodes should appear next to each other in the table at the top of the Mapping page.

Once you have loaded or generated both the XML document and the data packet and selected the nodes that appear in the mapping, the table at the top of the Mapping page should reflect the mapping you have defined.

Generating transformation files

To generate a transformation file, use the following steps:

1. First select the radio button that indicates what the transformation creates:
   - Choose the Datapacket to XML button if the mapping goes from data packet to XML document.
   - Choose the XML to Datapacket button if the mapping goes from XML document to data packet.
2. If you are generating a data packet, you will also want to use the radio buttons in the Create Datapacket As section. These buttons let you specify how the data packet will be used: as a dataset, as a delta packet for applying updates, or as the parameters to supply to a provider before fetching data.

3 Click Create and Test Transformation to generate an in-memory version of the transformation. XML mapper displays the XML document that would be generated for the data packet in the Datapacket pane or the data packet that would be generated for the XML document in the XML Document pane.

4 Finally, choose File | Save | Transformation to save the transformation file. The transformation file is a special XML file (with the .xtr extension) that describes the transformation you have defined.

Converting XML documents into data packets

Once you have created a transformation file that indicates how to transform an XML document into a data packet, you can create data packets for any XML document that conforms to the schema used in the transformation. These data packets can then be assigned to a client dataset and saved to a file so that they form the basis of a file-based database application.

The *TXMLTransform* component transforms an XML document into a data packet according to the mapping in a transformation file.

Note You can also use *TXMLTransform* to convert a data packet that appears in XML format into an arbitrary XML document.

Specifying the source XML document

There are three ways to specify the source XML document:

- If the source document is an .xml file on disk, you can use the *SourceXmlFile* property.
- If the source document is an in-memory string of XML, you can use the *SourceXml* property.
- If you have an *IDOMDocument* interface for the source document, you can use the *SourceXmlDocument* property.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *SourceXmlFile* property. Only if *SourceXmlFile* is an empty string does it check the *SourceXml* property. Only if *SourceXml* is an empty string does it then check the *SourceXmlDocument* property.

Specifying the transformation

There are two ways to specify the transformation that converts the XML document into a data packet:

- Set the *TransformationFile* property to indicate a transformation file that was created using xmlmapper.
- Set the *TransformationDocument* property if you have an *IDOMDocument* interface for the transformation.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *TransformationFile* property. Only if *TransformationFile* is an empty string does it check the *TransformationDocument* property.

Obtaining the resulting data packet

To cause *TXMLTransform* to perform its transformation and generate a data packet, you need only read the *Data* property. For example, the following code uses an XML document and transformation file to generate a data packet, which is then assigned to a client dataset:

Delphi example

```
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
```

C++ example

```
XMLTransform1->SourceXMLFile = "CustomerDocument.xml";
XMLTransform1->TransformationFile = "CustXMLToCustTable.xtr";
ClientDataSet1->XMLData = XMLTransform1->Data;
```

Converting user-defined nodes

When you define a transformation using xmlmapper, you can specify that some of the nodes in the XML document are "user-defined". User-defined nodes are nodes for which you want to provide the transformation in code rather than relying on a straightforward node-value-to-field-value translation.

You can provide the code to translate user-defined nodes using the *OnTranslate* event. An *OnTranslate* event handler is called every time the *TXMLTransform* component encounters a user-defined node in the XML document. In the *OnTranslate* event handler, you can read the source document and specify the resulting value for the field in the data packet.

For example, the following *OnTranslate* event handler converts a node in the XML document with the following form

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

into a single field value:

Delphi example

```
procedure TForm1.XMLTransform1Translate(Sender: TObject; Id: String; SrcNode: IDOMNode;
  var Value: String; DestNode: IDOMNode);
var
```

```
  CurNode: IDOMNode;
begin
  if Id = 'FullName' then
  begin
    Value = '';
    if SrcNode.hasChildNodes then
    begin
      CurNode := SrcNode.firstChild;
      Value := Value + CurNode.nodeValue;
      while CurNode <> SrcNode.lastChild do
      begin
        CurNode := CurNode.nextSibling;
        Value := Value + ' ';
        Value := Value + CurNode.nodeValue;
      end;
    end;
  end;
end;
```

C++ example

```
void __fastcall TForm1::XMLTransform1Translate(TObject *Sender, AnsiString Id,
  _di_IDOMNode SrcNode, AnsiString &Value, _di_IDOMNode DestNode)
{
  if (Id == "FullName")
  {
    Value = "";
    if (SrcNode.hasChildNodes)
    {
      _di_IXMLDOMNode CurNode = SrcNode.firstChild;
      Value = SrcValue + AnsiString(CurNode.nodeValue);
      while (CurNode != SrcNode.lastChild)
      {
        CurNode = CurNode.nextSibling;
        Value = Value + AnsiString(" ");
        Value = Value + AnsiString(CurNode.nodeValue);
      }
    }
  }
}
```

Using an XML document as the source for a provider

The *TXMLTransformProvider* component lets you use an XML document as if it were a database table. *TXMLTransformProvider* packages the data from an XML document and applies updates from clients back to that XML document. It appears to clients such as client datasets or XML brokers like any other provider component. For information on provider components, see Chapter 26, “Using provider components” For information on using provider components with client datasets, see “Using a client dataset with a provider” on page 25-27.

You can specify the XML document from which the XML provider provides data and to which it applies updates using the *XMLDataFile* property.

TXMLTransformProvider components use internal *TXMLTransform* components to translate between data packets and the source XML document: one to translate the XML document into data packets, and one to translate data packets back into the XML format of the source document after applying updates. These two *TXMLTransform* components can be accessed using the *TransformRead* and *TransformWrite* properties, respectively.

When using *TXMLTransformProvider*, you must specify the transformations that these two *TXMLTransform* components use to translate between data packets and the source XML document. You do this by setting the *TXMLTransform* component's *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component.

In addition, if the transformation includes any user-defined nodes, you must supply an *OnTranslate* event handler to the internal *TXMLTransform* components.

You do not need to specify the source document on the *TXMLTransform* components that are the values of *TransformRead* and *TransformWrite*. For *TransformRead*, the source is the file specified by the provider's *XMLDataFile* property (although, if you set *XMLDataFile* to an empty string, you can supply the source document using the *XmlSource* or *XmlSourceDocument* property of the *TXMLTransform* that is the value of *TransformRead*). For *TransformWrite*, the source is generated internally by the provider when it applies updates.

Using an XML document as the client of a provider

The *TXMLTransformClient* component acts as an adapter to let you use an XML document (or set of documents) as the client for an application server (or simply as the client of a dataset to which it connects via a *TDataSetProvider* component). That is, *TXMLTransformClient* lets you publish database data as an XML document and to make use of update requests (insertions or deletions) from an external application that supplies them in the form of XML documents.

To specify the provider from which the *TXMLTransformClient* object fetches data and to which it applies updates, set the *ProviderName* property. As with the *ProviderName* property of a client dataset, *ProviderName* can be the name of a provider on a remote application server or it can be a local provider in the same form or data module as the *TXMLTransformClient* object. For information about providers, see Chapter 26, "Using provider components"

If the provider is on a remote application server, you must use a *TSoapConnection* component to connect to that application server. Specify the connection component using the *RemoteServer* property. For information on *TSoapConnection* components, see "Connecting to the application server" on page 27-16.

Fetching an XML document from a provider

TXMLTransformClient uses an internal *TXMLTransform* component to translate data packets from the provider into an XML document. You can access this *TXMLTransform* component as the value of the *TransformGetData* property.

Before you can create an XML document that represents the data from a provider, you must specify the transformation file that *TransformGetData* uses to translate the data packet into the appropriate XML format. You do this by setting the *TXMLTransform* component's *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component. If that transformation includes any user-defined nodes, you will want to supply *TransformGetData* with an *OnTranslate* event handler as well.

There is no need to specify the source document for *TransformGetData*, *TXMLTransformClient* fetches that from the provider. However, if the provider expects any input parameters, you may want to set them before fetching the data. Use the *SetParams* method to supply these input parameters before you fetch data from the provider. *SetParams* takes two arguments: a string of XML from which to extract parameter values, and the name of a transformation file to translate that XML into a data packet. *SetParams* uses the transformation file to convert the string of XML into a data packet, and then extracts the parameter values from that data packet.

Note You can override either of these arguments if you want to specify the parameter document or transformation in another way. Simply set one of the properties on *TransformSetParams* property to indicate the document that contains the parameters or the transformation to use when converting them, and then set the argument you want to override to an empty string when you call *SetParams*. For details on the properties you can use, see "Converting XML documents into data packets" on page 28-6.

Once you have configured *TransformGetData* and supplied any input parameters, you can call the *GetDataAsXml* method to fetch the XML. *GetDataAsXml* sends the current parameter values to the provider, fetches a data packet, converts it into an XML document, and returns that document as a string. You can save this string to a file:

Delphi example

```
var
  XMLDoc: TFileStream;
  XML: string;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
  XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
  XMLTransformClient1.TransFormSetParams.SourceXmlFile := 'InputParams.xml';
  XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
  XML := XMLTransformClient1.GetDataAsXml;
  XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
  try
    XMLDoc.Write(XML, Length(XML));
  finally
    XMLDoc.Free;
  end;
end;
```

C++ example

```
XMLTransformClient1->ProviderName = "Provider1";
XMLTransformClient1->TransformGetData->TransformationFile = "CustTableToCustXML.xtr";
XMLTransformClient1->TransFormSetParams->SourceXmlFile = "InputParams.xml";
XMLTransformClient1->SetParams("", "InputParamsToDP.xtr");
AnsiString XML = XMLTransformClient1->GetDataAsXml();
TFileStream pXMLDoc = new TFileStream("Customers.xml", fmCreate || fmOpenWrite);
__try
{
  pXMLDoc->Write(XML.c_str(), XML.Length());
}
__finally
{
  delete pXMLDoc;
}
```

Applying updates from an XML document to a provider

TXMLTransformClient also lets you insert all of the data from an XML document into the provider's dataset or to delete all of the records in an XML document from the provider's dataset. To perform these updates, call the *ApplyUpdates* method, passing in

- A string whose value is the contents of the XML document with the data to insert or delete.
- The name of a transformation file that can convert that XML data into an insert or delete delta packet. (When you define the transformation file using the XML mapper utility, you specify whether the transformation is for an insert or delete delta packet.)
- The number of update errors that can be tolerated before the update operation is aborted. If fewer than the specified number of records can't be inserted or deleted, *ApplyUpdates* returns the number of actual failures. If more than the specified number of records can't be inserted or deleted, the entire update operation is rolled back, and no update is performed.

The following call transforms the XML document Customers.xml into a delta packet and applies all updates regardless of the number of errors:

```
StringList1.LoadFromFile('Customers.xml');
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
```

```
StringList1->LoadFromFile("Customers.xml");
nErrors = ApplyUpdates(StringList1->Text, "CustXMLToInsert.xtr", -1);
```

Part III

Writing Internet applications

The chapters in “Writing Internet applications” present concepts and skills necessary for building applications that are distributed over the Internet.

Note The components described in this section are not available in all editions.

Chapter 29

Creating Internet server applications

Web server applications extend the functionality and capability of existing Web servers. A Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Many operations that you can perform with an ordinary application can be incorporated into a Web server application.

This product provides two different architectures for developing Web server applications: Web Broker and WebSnap. Although these two architectures are different, WebSnap and Web Broker have many common elements. The WebSnap architecture acts as a superset of Web Broker. It provides additional components and new features like the Preview tab, which allows the content of a page to be displayed without the developer having to run the application. Applications developed with WebSnap can include Web Broker components, whereas applications developed with Web Broker cannot include WebSnap components.

This chapter describes the features of the Web Broker and WebSnap technologies and provides general information on Internet-based client/server applications.

About Web Broker and WebSnap

Part of the function of any application is to make data accessible to the user. In a standard application you accomplish this by creating traditional front end elements, like dialogs and scrolling windows. Developers can specify the exact layout of these objects using familiar form design tools. Web server applications must be designed differently, however. All information passed to users must be in the form of HTML pages which are transferred through HTTP. Pages are generally interpreted on the client machine by a Web browser application, which displays the pages in a form appropriate for the user's particular system in its present state.

The first step in building a Web server application is choosing which architecture you want to use, Web Broker or WebSnap. Both approaches provide many of the same features, including

- Support for CGI, Web App Debugger and Apache Web server application types. These are described in "Types of Web server applications" on page 29-6.
- Multithreading support so that incoming client requests are handled on separate threads.
- Caching of Web modules for quicker responses.
- Cross-platform development. You can easily port your Web server application between the Windows and Linux operating systems. Your source code will compile on either platform.

Both the Web Broker and WebSnap components handle all of the mechanics of page transfer. WebSnap uses Web Broker as its foundation, so it incorporates all of the functionality of Web Broker's architecture. WebSnap offers a much more powerful set of tools for generating pages, however. Also, WebSnap applications allow you to use server-side scripting to help generate pages at runtime. Web Broker does not have this scripting capability. The tools offered in Web Broker are not nearly as complete as those in WebSnap, and are much less intuitive. If you are developing a new Web server application, WebSnap is probably a better choice of architecture than Web Broker.

The major differences between these two approaches are outlined in the following table:

Table 29.1 Web Broker versus WebSnap

Web Broker	WebSnap
Backward compatible	Although WebSnap applications can use any Web Broker components that produce content, the Web modules and dispatcher that contain them are new.
Only one Web module allowed in an application.	Multiple Web modules can partition the application into units, allowing multiple developers to work on the same project with fewer conflicts.
Only one Web dispatcher allowed in the application.	Multiple, special-purpose dispatchers handle different types of requests.
Specialized components for creating content include page producers and Web Services components.	Supports all the content producers that can appear in Web Broker applications, plus many others designed to let you quickly build complex data-driven Web pages.
No scripting support.	Support for server-side scripting allows HTML generation logic to be separated from the business logic.
No built-in support for named pages.	Named pages can be automatically retrieved by a page dispatcher and addressed from server-side scripts.
No session support.	Sessions store information about an end user that is needed for a short period of time. This can be used for such tasks as login/logout support.

Table 29.1 Web Broker versus WebSnap (continued)

Web Broker	WebSnap
Every request must be explicitly handled, using either an action item or an auto-dispatching component.	Dispatch components automatically respond to a variety of requests.
Only a few specialized components provide previews of the content they produce. Most development is not visual.	WebSnaplets you build Web pages more visually and view the results at design time. Previews are available for all components.

For more information on Web Broker, see Chapter 30, "Using Web Broker." For more information on WebSnap, see Chapter 31, "Creating Web server applications using WebSnap."

Terminology and standards

Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development arm of the Internet. There are several important RFCs that you will find useful when writing Internet applications:

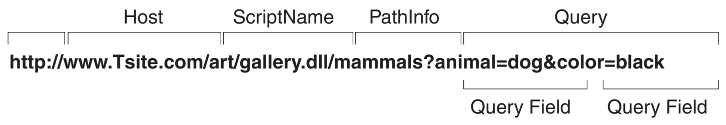
- RFC822, "Standard for the format of ARPA Internet text messages," describes the structure and content of message headers.
- RFC1521, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," describes the method used to encapsulate and transport multipart and multiformat messages.
- RFC1945, "Hypertext Transfer Protocol — HTTP/1.0," describes a transfer mechanism used to distribute collaborative hypermedia documents.

The IETF maintains a library of the RFCs on their Web site, `www.ietf.cnri.reston.va.us`.

Parts of a Uniform Resource Locator

The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the Internet. It is composed of several parts that may be accessed by an application. These parts are illustrated in Figure 29.1:

Figure 29.1 Parts of a Uniform Resource Locator


Host
ScriptName
PathInfo
Query
http://www.Tsite.com/art/gallery.dll/mammals?animal=dog&color=black
Query Field
Query Field

The first portion (not technically part of the URL) identifies the protocol (http). This portion can specify other protocols such as https (secure http), ftp, and so on.

The Host portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port that receives messages. Usually, there is no need to specify a port, because the port number is implied by the protocol.

The ScriptName portion specifies the name of the Web server application. This is the application to which the Web server passes messages.

Following the script name is the pathinfo. This identifies the destination of the message within the Web server application. Path info values may refer to directories on the host machine, the names of components that respond to specific messages, or any other mechanism the Web server application uses to divide the processing of incoming messages.

The Query portion contains a set a named values. These values and their names are defined by the Web server application.

URI vs. URL

The URL is a subset of the Uniform Resource Identifier (URI) defined in the HTTP standard, RFC1945. Web server applications frequently produce content from many sources where the final result does not reside in a particular location, but is created as necessary. URIs can describe resources that are not location-specific.

HTTP request header information

HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as “Host” followed by a string value. For example, consider the following HTTP request:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line identifies the request as a GET. A GET request message asks the Web server application to return the content associated with the URI that follows the word GET (in this case /art/gallery.dll/animals?animal=doc&color=black). The last part of the first line indicates that the client is using the HTTP 1.0 standard.

The second line is the Connection header, and indicates that the connection should not be closed once the request is serviced. The third line is the User-Agent header, and provides information about the program generating the request. The next line is the Host header, and provides the Host name and port on the server that is contacted to form the connection. The final line is the Accept header, which lists the media types the client can accept as valid responses.

HTTP server activity

The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML.

Before building a Web server application, it is useful to understand how the client issues a request and how the server responds to client requests.

Composing client requests

When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request.

For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:

```
http://www.TSite.com/art/gallery/animals?animal=dog&color=black
```

which specifies an HTTP server in the www.TSite.com domain. The client contacts www.TSite.com, connects to the HTTP server, and passes it a request. The request might look something like this:

```
GET /art/gallery/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Serving client requests

The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.dll portion of the request as a program, it passes information about the request to that program. The way information about the request is passed to the program depends on the type of Web server application.

For example, if the program is a Common Gateway Interface (CGI) program, the server passes the information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes the content directly back to the server.

In all cases, the program acts on the request of and performs actions specified by the programmer: accessing databases, doing simple table lookups or calculations, constructing or selecting HTML documents, and so on.

Responding to client requests

When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. The way the response is sent also differs based on the type of program.

When a dynamic shared object (DSO) finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client.

Types of Web server applications

Whether you use Web Broker or WebSnap, you can create several standard types of Web server applications. In addition, you can create a Web App Debugger executable, which integrates the Web server into your application so that you can debug your application logic. The Web App Debugger executable is intended only for debugging. When you deploy your application, you should migrate to one of the other types.

CGI stand-alone

A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by the CGI application, which creates appropriate request and response objects. Each request message is handled by a separate instance of the application.

Apache

An Apache DSO module is a library application. The Apache program loads a shared object in memory, and then reserves a memory block which is used to pass data between the Apache program and module.

Note In Delphi, you can choose whether or not to build your Apache application with runtime packages. In C++, however, you must not build with runtime packages. This is the default setting in the C++ IDE. To ensure that the IDE is configured correctly, choose Project | Options | Packages and uncheck the appropriate box (if necessary).

installing Apache

Before you can deploy an Apache DSO application, either in a production or testing environment, you must first make sure that your Apache server is compiled for DSO support. Unfortunately, many off-the-shelf Linux Apache packages are not configured to support DSOs. Most users will need to remove their current Apache installation and replace it with one custom-built from Apache 1.3 source code. Fortunately, the build process is well-automated.

Note Apache installations generally belong to the root user of a Linux system. Since Web server DSOs are strongly tied to the Apache server program, only the root user can install and debug them. If you want to test and debug an application without superuser privileges, install a second version of Apache in your home directory and configure it to use an unused port number higher than 1024. See Apache's documentation for more information.

To compile Apache for DSO support:

1 Download Apache source code from httpd.apache.org. For example, you can use:

```
http://httpd.apache.org/dist/httpd/apache_1.3.20.tar.gz
```

2 Extract the downloaded source tar. For example, if you downloaded the source code file to the current directory, then you can use the following command from a command prompt (with $ as the root user's prompt in the bash shell) to place the Apache source code in the subdirectory apache_1.3.20:

```
$ gunzip < apache_1.3.20.tar.gz | tar xvf -
```

3 Change your directory to the root of the extracted directory (such as apache_1.3.20).

4 If the config.status file does not exist, create it. Perhaps the best way is to do a standard Apache installation according to the instructions in the README.configure file. The resulting installation will be replaced by an installation that supports DSOs.

5 Make sure that all of the lines shown below are included in the config.status file:

```
LIBS="/usr/lib/libpthread.so" \
./configure \
"--with-layout=Apache" \
"--enable-module=so" \
"--enable-rule=SHARED_CORE" \
"$@"
```

Add any missing lines as needed. Leave other lines in place, but modify them if necessary.

6 At the command prompt ($), type the following lines:

```
$ ./config.status
$ make
$ make install
```

This configures Apache for DSO and debugging support, installing it into the target directory. You can now start the Apache server using the command `apachectl start` from the bin directory of your Apache installation.

For more information on Apache DSO support, see the following document on the Apache Web site:

```
http://httpd.apache.org/docs/dso.html
```

Deploying Applications to Apache

When you deploy your Apache Web server application, you will need to specify some application-specific information in the Apache configuration files. The default module name is the project name with `_module` appended to the end. For example, a project named Project1 would have `Project1_module` as its module name. Similarly, the default content type is the project name with `-content` appended, and the default handler type is the project name with`-handler` appended.

These definitions can be changed in the project (.dpr or .bpr) file when necessary. For example, when you create your project a default module name is stored in the project file. Here are some common examples:

```
exports
  apache_module name 'Project1_module';
```

```
extern "C"
{
  Httpd::module __declspec(dllexport) Project1_module;
}
```

Note When you rename the project during the save process, that name isn't changed automatically. Whenever you rename your project, you must change the module name in your project file to match your project name. The content and handler definitions should change automatically once you change the module name.

For more information on using module, content, and handler definitions in your Apache configuration files, see the documentation on the Apache Web site httpd.apache.org.

Web App Debugger

The server types mentioned above have their advantages and disadvantages for production environments, but none of them is well-suited for debugging. Deploying your application and configuring the debugger can make Web server application debugging far more tedious than debugging other application types.

Fortunately, the IDE includes a Web App Debugger which makes debugging simple. The Web App Debugger is a small, easily configured Web server for your development machine. If you build your Web server application as a Web App Debugger executable, deployment happens automatically during the build process. To debug your application, start it using Run | Run. Next, select Tools | Web App Debugger, click the default URL and select your application in the Web browser which appears. Your application will launch in the browser window, and you can use the IDE to set breakpoints and obtain debugging information.

When your application is ready to be tested or deployed in a production environment, you can convert your Web App Debugger project to one of the other target types using the steps given below.

Note When you create a Web App Debugger project, you will need to provide a Class Name for your project. This is simply a name used by the Web App Debugger to refer to your application. You can safely use the application's name as the Class Name.

Converting Web server application target types

One powerful feature of Web Broker and WebSnap is that they offer several different target server types. You can easily convert from one target type to another. For example, you can convert a Web App Debugger application to another target type (such as CGI) prior to final deployment.

Because Web Broker and WebSnap have slightly different design philosophies, you must use a different conversion method for each architecture. To convert your Web Broker application target type, use the following steps:

1. Right-click the Web module and choose Add To Repository.
2. In the Add To Repository dialog box, give your Web module a title, text description, Repository page (probably Data Modules), author name, and icon.
3. Choose OK to save your Web module as a template.
4. From the main menu, choose File | New | Other and select Web Server Application. In the New Web Server Application dialog box, choose the appropriate target type.
5. Delete the automatically generated Web module.
6. From the main menu, choose File | New | Other and select the template you saved in step 3. This will be on the page you specified in step 2.

To convert a WebSnap application's target type:

1. Open your project in the IDE.
2. Display the Project Manager using View | Project Manager. Expand your project so all of its units are visible.
3. In the Project Manager, click the New button to create a new Web server application project. Double-click the WebSnap Application item in the WebSnap tab. Select the appropriate options for your project, including the server type you want to use, then click OK.
4. Expand the new project in the Project Manager. Select any files appearing there and delete them.
5. One at a time, select each file in your project (except for the form file in a Web App Debugger project) and drag it to the new project. When a dialog appears asking if you want to add that file to your new project, click Yes.

Debugging server applications

Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting.

The following topics describe techniques you can use to debug Web server applications.

- <~JMP Debugging Apache DSO applications~WServerDebuggingApacheDSOApplications JMP~>

Using the Web App Debugger

The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server.

To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application.

For information on how to write this Web server application using Web Broker, see Chapter 30, "Using Web Broker." For more information on using WebSnap, see Chapter 31, "Creating Web server applications using WebSnap."

Launching your application with the Web App Debugger

Once you have developed your Web server application, you can run and debug it as follows:

1. With your project loaded in the IDE, set any breakpoints so that you can debug your application just like any other executable.
2. Choose Run | Run. This displays a window which represents your Web server application. The first time you run your application, it registers itself so that the Web App debugger can access it.
3. Select Tools | Web App Debugger.
4. Click the Start button. This displays the ServerInfo page in your default Browser.
5. The ServerInfo page provides a drop-down list of all registered Web Application Debugger executables. Select your application from the drop-down list. If you do not find your application in this drop-down list, try running your application as an executable. Your application must be run once so that it can register itself. If you still do not find your application in the drop-down list, try refreshing the Web page. (Sometimes the Web browser caches this page, preventing you from seeing the most recent changes.)
6. Once you have selected your application in the drop-down list, press the Go button. This launches your application in the Web Application Debugger, which provides you with details on request and response messages that pass between your application and the Web Application Debugger.

Note You may need to change the Web App Debugger port numbers using File | Options if they conflict with those of other services on the host system.

Converting your application to another type of Web server application

When you have finished debugging your Web server application with the Web App Debugger, you will need to convert it to another type that can be installed on a commercial Web server. To learn more about converting your application, see "Converting Web server application target types" on page 29-8.

Debugging CGI applications

A CGI Web server application is basically a console application. It is launched by the Web server in response to a client request. A CGI Web server application can be deployed for use by placing it in a directory accessible to the Web server application. For example, Apache Web server installation directories usually include a directory named cgi-bin for this purpose. (Other directories can be used as well, depending on how Apache is configured.)

Normal console applications can be launched using the Run command on the Run menu, so debugging is automatically supported. Web server applications are launched by the Web server in response to an HTTP request. Debugging a CGI is more complicated than debugging a console application because request information must somehow be passed to the application. There are several ways this can be accomplished.

The first debugging method involves simulating the deployment environment closely. A browser is used to send request information; the Web server launches the CGI application and passes it the necessary request information. The Kylix IDE can then be used for debugging by attaching it to the running Web server application using the Run | Attach to process command.

This method is tricky because the CGI application terminates after each request. If the source code is not modified, the application developer has too little time to attach to the running Web server application before it terminates. This problem can be solved by including an infinite loop in the Web server application's source code. By including an infinite loop, you give yourself plenty of time to attach to the application's process after it has been spawned. Once the process is attached, execution pauses within the debugger, and variables can be modified to move execution past the loop.

For simplicity, place the infinite loop just prior to the breakpoints you are inserting into your application. Here is an example of how the loop can be written:

Delphi example

```
var
   waitForDebugger: Boolean;
...
// Debugging loop example.
waitForDebugger := True;
while (waitForDebugger = True) do
begin
   // Does nothing but wait.
   // Loop can be broken by debugger only.
end;
```

C++ example

```
bool waitForDebugger = true;
// Debugging loop example.
while (waitForDebugger)
```

```
{
   // Does nothing but wait.
   // Loop can be broken by debugger only.
}
```

To exit the loop, select Run | Inspect and specify waitForDebugger. A Debug Inspector window appears, which lets you change waitForDebugger from true to false. Select Run | Run, and execution will move out of the loop. Debugging now proceeds as it would for any other application.

There is another option available for debugging CGI applications which allows you to remove the browser and server from the process. When a CGI server application executes, it reads its request information from environment variables set by the Web server. The environment variables can also be set manually, through the Tools | Environment Options | Environment Variables tab or some other method. By setting environment variables to contain request information, you can mimic the presence of a user making a request through a browser. Debugging becomes simple: just use Run | Run.

For more information on CGI environment variables, see the CGI specification, which can be found through the following Web site:

```
http://www.w3.org/CGI
```

Debugging Apache DSO applications

Debugging Apache DSO applications can be much simpler than debugging CGI applications. If you set the run parameters appropriately, the debugging process becomes almost identical to that of any other Kylix application. You can use Run | Run to launch Apache, then use a browser to start execution in your Web server application. You can then pause execution, inspect variables, and perform other debugging tasks as you would with other types of Kylix applications.

Deploying a DSO application

The first step in debugging is building the Web server application and making it accessible to Apache. The project can be built using the Project | Build command, which creates the project's executable and other necessary objects.

To incorporate your Web server application into your Apache server, you need to make two modifications to the Apache configuration file httpd.conf. (If you have installed Apache using the directions in this chapter, that file is in the conf directory of your Apache installation.) First, you need to add a LoadModule line to the file. This line directs Apache to load your Web server application when it starts. It also specifies the location of the application's shared object file. To add the LoadModule directive, open the file with a text editor. A number of LoadModule directives should already exist in the file. After the last directive, add a line like this:

```
LoadModule Project1_module PROJECTPATH/Project1.so
```

In the example above, *PROJECTPATH* is a path to the directory where your compiled executable (Project1.so) resides. You can modify the example to fit the specifics of your own application.

The second modification you need to make is to add a location specification, which indicates what URL should launch your application. The location specification can be added to httpd.conf anywhere after the LoadModule directive. Here is an example:

```
# Sample location specification for a project named Project1.
<Location /Project1>
   SetHandler project1-handler
</Location>
```

The SetHandler directive specifies the Web server application which handles the request. The argument of the SetHandler directive should be set to the value of the *ContentType* global variable.

D In Delphi, *ContentType* can be defined in your project's .dpr (Delphi), just after the begin statement. For example, the following line of code appears in Project1.dpr:

```
ContentType := 'project1-handler';
```

In this example, a browser could launch the executable by appending "/Project1" to the address of the server.

In C++, the procedures for changing *ContentType* are described in your own project's source code as comments. For more information, refer to your own source code.

Note WebSnap applications which use JavaScript server-side scripting must have the script engine shared object (libjs.so) accessible to them. This object is included in the bin directory of the Kylix installation, and there are many ways to make it accessible. For example, you could specify Kylix's bin directory in the application's LD_LIBRARY_PATH environment variable.

Setting up for DSO debugging

To debug Apache DSO applications, configure the IDE to launch Apache (the host application) by setting the run parameters as shown below. Be sure to replace the path /your/apache with the path to the Apache installation you intend to use.

To debug an Apache DSO application:

1 Set up the run parameters to launch the host application.
   - Choose Run | Parameters.
   - Set the host application to /your/apache/libexec/libhttpd.so.
   - Set parameters to -X -f /your/apache/conf/httpd.conf.
   - Click OK.

2 Set the output and search directories.
   - Choose Project | Options | Directories
   - Set the output and search directories to the directory containing your Apache DSO and related files, such as /your/apache/libexec.

- Click OK.

3 Build the project by selecting the Build command from the Project menu.

4 Deploy the Web server application as shown above.

5 In the Kylix IDE, select Run | Run to start Apache.

6 Open a Web browser and request a URL that will invoke your Web server application.

You can now debug your Web server application.

Chapter

30

Using Web Broker

Web Broker components (located on the Internet tab of the component palette) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle numerous simultaneous, thread-safe database connections.

The following sections in this chapter explain how you use the Web Broker components to create a Web server application.

Creating Web server applications with Web Broker

To create a new Web server application using the Web Broker architecture:

1 Select File | New | Other.

2 In the New Items dialog box, select the New tab and choose Web Server Application.

3 A dialog box appears, where you can select one of the Web server application types:

- CGI stand-alone: Selecting this type of application sets up your project as a console application.
- Apache Shared Module (DSO): Selecting this type of application sets up your project as a dynamic shared object, or DSO, with the exported methods expected by the Apache Web server.

- Web Application Debugger stand-alone executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Choose the type of Web Server Application that communicates with the type of Web Server your application will use. This creates a new project configured to use Internet components and containing an empty Web Module.

The Web module

The Web module (*TWebModule*) is a descendant of *TDataModule* and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application.

Add any content producers that your application uses to generate response messages. These can be the built-in content producers, such as *TPageProducer*, or descendants of *TCustomContentProducer* that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components or special components for writing a Web server that acts as a client in a multi-tiered database application.

In addition to storing non-visual components and business rules, the Web module also acts as a Web dispatcher, matching incoming HTTP request messages to action items that generate the responses to those requests.

You may have a data module already that is set up with many of the non-visual components and business rules that you want to use in your Web application. You can replace the Web module with your pre-existing data module. Simply delete the automatically generated Web module and replace it with your data module. Then, add a *TWebDispatcher* component to your data module, so that it can dispatch request messages to action items, the way a Web module can. If you want to change the way action items are chosen to respond to incoming HTTP request messages, derive a new dispatcher component from *TCustomWebDispatcher*, and add that to the data module instead.

Your project can contain only one dispatcher. This can either be the Web module that is automatically generated when you create the project, or the *TWebDispatcher* component that you add to a data module that replaces the Web module. If a second data module containing a dispatcher is created during execution, the Web server application generates a runtime error.

Note The Web module that you set up at design time is actually a template.

The Web Application object

The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* (such as *TApacheApplication* or *TCGIApplication*) that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

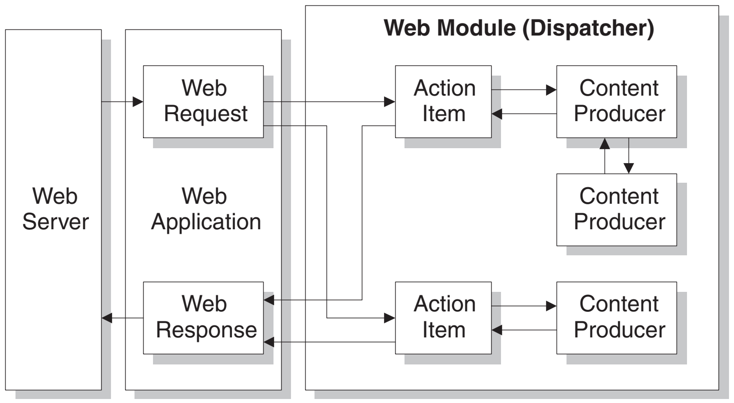
Warning Do not include the Forms or QForms units after a Web Broker application unit (CGIApp, ApacheApp, or ISAPIApp) in the project file. For example, Forms and QForms should not be part of the uses clause (Delphi) or include statements (C++) of your application if any of the Web Broker application units are listed. Like the Web Broker application units, Forms and QForms declare a global variable named *Application*, and if either of these units is included after a Web Broker application unit, *Application* will be initialized to an object of the wrong type.

The structure of a Web Broker application

When the Web application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component).

The Web dispatcher controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and passes the request and response objects to the identified handler so that it can perform any requested actions or formulate a response message. It is described more fully in the section "The Web dispatcher" on page 30-4.

Figure 30.1 Structure of a Server Application



The action items are responsible for reading the request and assembling a response message. Specialized content producer components aid the action items in dynamically generating the content of response messages, which can include custom HTML code or other MIME content. The content producers can make use of other content producers or descendants of *THTMLTagAttributes*, to help them create the content of the response message. For more information on content producers, see "Generating the content of response messages" on page 30-12.

If you are creating the Web Client in a multi-tiered database application, your Web server application may include additional, auto-dispatching components that represent database information encoded in XML and database manipulation classes encoded in JavaScript. If you are creating a server that implements a Web Service, your Web server application may include an auto-dispatching component that passes SOAP-based messages on to an invoker that interprets and executes them. The dispatcher calls on these auto-dispatching components to handle the request message after it has tried all of its action items.

When all action items (or auto-dispatching components) have finished creating the response by filling out the *TWebResponse* object, the dispatcher passes the result back to the Web application. The application sends the response on to the client via the Web server.

The Web dispatcher

If you are using a Web module, it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (*TWebDispatcher*) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it chooses one or more action items to respond to the request.

Adding actions to the dispatcher

Open the action editor from the Object Inspector by clicking the ellipsis on the *Actions* property of the dispatcher. Action items can be added to the dispatcher by clicking the Add button in the action editor.

Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate action item for every type of request, where each action item completely handles the request.

Action items are discussed in further detail in “Action items” on page 30-5.

Dispatching request messages

When the dispatcher receives the client request, it generates a *BeforeDispatch* event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items.

Next, the dispatcher iterates over its list of action items, looking for an entry that matches the PathInfo portion of the request message’s target URL and that also provides the service specified as the method of the request message. It does this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds an appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response or signals that the request is completely handled.
- Adds to the response and then allows other action items to complete the job.
- Defers the request to other action items.

If, after checking all the action items and any specially registered auto-dispatching components, the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

If the dispatcher reaches the end of the action list (including the default action, if any) and no actions have been triggered, nothing is passed back to the server. The server simply drops the connection to the client.

If the request is handled by the action items, the dispatcher generates an *AfterDispatch* event. This provides a final opportunity for your application to check the response that was generated, and make any last minute changes.

Action items

Each action item (*TWebActionItem*) performs a specific task in response to a given type of request message.

Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message.

Determining when action items fire

Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the Object Inspector and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the Object Inspector.

The target URL

The dispatcher compares the *PathInfo* property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

```
http://www.TSite.com/art/gallery/mammals?animal=dog&color=black
```

and assuming that the `/gallery` part indicates the Web server application, the path information portion is

```
/mammals
```

Use path information to indicate where your Web application should look for information when servicing requests, or to divide you Web application into logical subservices.

The request method type

The *MethodType* property of an action item indicates what type of request messages it can process. The dispatcher compares the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values:

Table 30.1 MethodType values

Value	Meaning
mtGet	The request is asking for the information associated with the target URI to be returned in a response message.
mtHead	The request is asking for the header properties of a response, as if servicing an *mtGet* request, but omitting the content of the response.
mtPost	The request is providing information to be posted to the Web application.
mtPut	The request asks that the resource associated with the target URI be replaced by the content of the request message.
mtAny	Matches any request method type, including *mtGet*, *mtHead*, *mtPut*, and *mtPost*.

Enabling and disabling action items

Each action item has an *Enabled* property that can be used to enable or disable that action item. By setting *Enabled* to false, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request.

A *BeforeDispatch* event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message.

Caution Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. Use the *BeforeDispatch* event to ensure that all action items are correctly initialized to their appropriate starting states.

Choosing a default action item

Only one of the action items can be the default action item. The default action item is selected by setting its *Default* property to true. When the *Default* property of an action item is set to true, the *Default* property for the previous default action item (if any) is set to false.

When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items, it executes the default action item.

The dispatcher does not check the *PathInfo* or *MethodType* of the default action item. The dispatcher does not even check the *Enabled* property of the default action item. Thus, you can make sure the default action item is only called at the very end by setting its *Enabled* property to false.

The default action item should be prepared to handle any request that is encountered, even if it is only to return an error code indicating an invalid URI or *MethodType*. If the default action item does not handle the request, no response is sent to the Web client.

Caution Changing the *Default* property of an action during execution may cause unexpected results for the current request. If the *Default* property of an action that has already been triggered is set to true, that action will not be reevaluated and the dispatcher will not trigger that action when it reaches the end of the action list.

Responding to request messages with action items

The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways:

- If the action item has an associated producer component as the value of its *Producer* property, that producer automatically assigns the *Content* of the response message using its *Content* method. The Internet page of the component palette includes a number of content producer components that can help construct an HTML page for the content of the response message.
- After the producer has assigned any response content (if there is an associated producer), the action item receives an *OnAction* event. The *OnAction* event handler is passed the *TWebRequest* object that represents the HTTP request message and a *TWebResponse* object to fill with any response information.

If the action item's content can be generated by a single content producer, it is simplest to assign the content producer as the action item's *Producer* property. However, you can always access any content producer from the *OnAction* event handler as well. The *OnAction* event handler allows more flexibility, so that you can use multiple content producers, assign response message properties, and so on.

Both the content-producer component and the *OnAction* event handler can use any objects or runtime library methods to respond to request messages. They can access databases, perform calculations, construct or select HTML documents, and so on. For more information about generating response content using content-producer components, see "Generating the content of response messages" on page 30-12.

Sending the response

An *OnAction* event handler can send the response back to the Web client by using the methods of the *TWebResponse* object. However, if no action item sends the response to the client, it will still get sent by the Web server application as long as the last action item to look at the request indicates that the request was handled.

Using multiple action items

You can respond to a request from a single action item, or divide the work up among several action items. If the action item does not completely finish setting up the response message, it must signal this state in the *OnAction* event handler by setting the *Handled* parameter to false.

If many action items divide up the work of responding to request messages, each setting *Handled* to false so that others can continue, make sure the default action item leaves the *Handled* parameter set to true. Otherwise, no response will be sent to the Web client.

When dividing the work among several action items, either the *OnAction* event handler of the default action item or the *AfterDispatch* event handler of the dispatcher should check whether all the work was done and set an appropriate error code if it is not.

Accessing client request information

When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of a *TWebRequest* object. *TISAPIRequest* In CGI applications, the request message is encapsulated by a *TCGIRequest* object. *TWinCGIRequest*

The properties of the request object are read-only. You can use them to gather all of the information available in the client request.

Properties that contain request header information

Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the *GetFieldByName* method.

Properties that identify the target

The full target of the request message is given by the *URL* property. Usually, this is a URL that can be broken down into the protocol (HTTP), *Host* (server system), *ScriptName* (server application), *PathInfo* (location on the host), and *Query*.

Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you as the *QueryFields* property.

Properties that describe the Web client

The request also includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the *From* property), to the URI where the message originated (the *Referer* or *RemoteHost* property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the *DerivedFrom* property. You can also determine the IP address of the client (the

RemoteAddr property), and the name and version of the application that sent the request (the *UserAgent* property).

Properties that identify the purpose of the request

The *Method* property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods:

Value	What the message requests
OPTIONS	Information about available communication options.
GET	Information identified by the *URL* property.
HEAD	Header information from an equivalent GET message, without the content of the response.
POST	The server application to post the data included in the *Content* property, as appropriate.
PUT	The server application to replace the resource indicated by the *URL* property with the data included in the *Content* property.
DELETE	The server application to delete or hide the resource identified by the *URL* property.
TRACE	The server application to send a loop-back to confirm receipt of the request.

The *Method* property may indicate any other method that the Web client requests of the server.

The Web server application does not need to provide a response for every possible value of *Method*. The HTTP standard does require that it service both GET and HEAD requests, however.

The *MethodType* property indicates whether the value of *Method* is GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) or some other string (mtAny). The dispatcher matches the value of the *MethodType* property with the *MethodType* of each action item.

Properties that describe the expected response

The *Accept* property indicates the media types the Web client will accept as the content of the response message. The *IfModifiedSince* property specifies whether the client only wants information that has changed recently. The *Cookie* property includes state information (usually added previously by your application) that can modify the response.

Properties that describe the content

Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the *ContentType* property, and its length in the *ContentLength* property. If the content of the message was encoded (for example, for data compression), this information is in the *ContentEncoding* property. The name and version number of the application that produced the content is specified by the *ContentVersion* property. The *Title* property may also provide information about the content.

The content of HTTP request messages

In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database accessed by the Web server application.

The unprocessed value of the content is given by the *Content* property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the *ContentFields* property.

Creating HTTP response messages

When the Web server application creates a *TWebRequest* object for an incoming HTTP request message, it also creates a corresponding *TWebResponse* object to represent the response message that will be sent in return. In Apache applications, the response message is encapsulated by a *TApacheResponse* object. Console CGI applications use *TCGIResponse* objects.

The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In other cases, this may involve complicated calculations that require the action item to fetch information from other sources and assemble it into a finished form. Most request messages require some response, even if it is only the acknowledgment that a requested action was carried out.

Filling in the response header

Most of the properties of the *TWebResponse* object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler.

Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response.

Indicating the response status

Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the *StatusCode* property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values.

Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows:

- 1xx: Informational (The request was received but has not been fully processed).
- 2xx: Success (The request was received, understood, and accepted).
- 3xx: Redirection (Further action by the client is needed to complete the request).
- 4xx: Client Error (The request cannot be understood or cannot be serviced).
- 5xx: Server Error (The request was valid but the server could not handle it).

Associated with each status code is a string that explains the meaning of the status code. This is given by the *ReasonString* property. For predefined status codes, you do not need to set the *ReasonString* property. If you define your own status codes, you should also set the *ReasonString* property.

Indicating the need for client action

When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the request, set the *Location* property. If the client must provide a password before you can proceed, set the *WWWAuthenticate* property.

Describing the server application

Some of the response header properties describe the capabilities of the Web server application. The *Allow* property indicates the methods to which the application can respond. The *Server* property gives the name and version number of the application used to generate the response. The *Cookies* property can hold state information about the client's use of the server application which is included in subsequent request messages.

Describing the content

Several properties describe the content of the response. *ContentType* gives the media type of the response, and *ContentVersion* is the version number for that media type. *ContentLength* gives the length of the response. If the content is encoded (such as for data compression), indicate this with the *ContentEncoding* property. If the content came from another URI, this should be indicated in the *DerivedFrom* property. If the value of the content is time-sensitive, the *LastModified* property and the *Expires* property indicate whether the value is still valid. The *Title* property can provide descriptive information about the content.

Setting the response content

For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer.

You can set the content of the response message by using either the *Content* property or the *ContentStream* property.

The *Content* property is a string. Strings are not limited to text values, so the value of the *Content* property can be a string of HTML commands, graphics content such as a bit-stream, or any other MIME content type.

Use the *ContentStream* property if the content for the response message can be read from a stream. For example, if the response message should send the contents of a file, use a *TFileStream* object for the *ContentStream* property. As with the *Content* property, *ContentStream* can provide a string of HTML commands or other MIME content type. If you use the *ContentStream* property, do not free the stream yourself. The Web response object automatically frees it for you.

Note If the value of the *ContentStream* property is not nil (Delphi) or NULL (C++), the *Content* property is ignored.

Sending the response

If you are sure there is no more work to be done in response to a request message, you can send a response directly from an *OnAction* event handler. The response object provides two methods for sending a response: *SendResponse* and *SendRedirect*. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient.

If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if none of the action items indicate that they have handled the response, the application will close the connection to the Web client without sending any response.

Generating the content of response messages

Web Broker provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from *TCustomContentProducer* or one of its descendants.

TCustomContentProducer provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers:

- Page producers scan HTML documents for special tags that they replace with customized HTML code. They are described in the following section.
- Table producers create HTML commands based on the information in a dataset. They are described in “Using database information in responses” on page 30-17.

Using page producer components

Page producers (*TPageProducer* and its descendants) take an HTML template and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.

HTML templates

An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

The angle brackets (< and >) define the entire scope of the tag. A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The tag name can be any valid identifier and identifies the type of conversion the tag represents.

Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form *ParamName*=*Value*, where there is no space between the parameter name, the equals symbol (=) and the value. The parameters are separated by whitespace.

The angle brackets (< and >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

Using predefined HTML-transparent tag names

While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the *TTag* data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

Tag Name	TTag value	What the tag should be converted to
Link	*tgLink*	A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an </A> tag.
Image	*tgImage*	A graphic image. The result is an HTML <IMG> tag.
Table	*tgTable*	An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag.
ImageMap	*tgImageMap*	A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag.

Any other tag name is associated with *tgCustom*. The page producer supplies no built-in processing of the predefined tag names. They are simply provided to help applications organize the conversion process into many of the more common tasks.

Note The predefined tag names are case insensitive.

Specifying the HTML template

Page producers provide you with many choices in how to specify the HTML template. You can set the *HTMLFile* property to the name of a file that contains the HTML template. You can set the *HTMLDoc* property to a *TStrings* object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the *Content* method.

In addition, you can call the *ContentFromString* method to directly convert an HTML template that is a single string which is passed in as a parameter. You can also call the *ContentFromStream* method to read the HTML template from a stream. Thus, for example, you could store all your HTML templates in a memo field in a database, and use the *ContentFromStream* method to obtain the converted HTML commands.

Converting HTML-transparent tags

The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers the *OnHTMLTag* event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content.

If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.

Using page producers from an action item

A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:

Delphi example

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  PageProducer1.HTMLFile := 'Greeting.html';
  Response.Content := PageProducer1.Content;
end;
```

C++ example

```
void __fastcall WebModule1::MyActionEventHandler(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  PageProducer1->HTMLFile = "Greeting.html";
  Response->Content = PageProducer1->Content();
}
```

Greeting.html is a file that contains this HTML template:

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello <#UserName>!  Welcome to our Web site.
</BODY>
</HTML>
```

The *OnHTMLTag* event handler replaces the custom tag (`<#UserName>`) in the HTML during execution:

Delphi example

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
  if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

C++ example

```
void __fastcall WebModule1::HTMLTagHandler(TObject *Sender, TTag Tag,
   const AnsiString TagString, TStrings *TagParams, AnsiString &ReplaceText)
{
  if (CompareText(TagString,"UserName") == 0)
    ReplaceText = ((TPageProducer *)Sender)->Dispatcher->Request->Content;
}
```

If the content of the request message was the string *Mr. Ed,* the value of *Response*'s *Content* property would be

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!  Welcome to our Web site.
</BODY>
</HTML>
```

Note This example uses an *OnAction* event handler to call the content producer and assign the content of the response message. You do not need to write an *OnAction* event handler if you assign the page producer's *HTMLFile* property at design time. In that case, you can simply assign *PageProducer1* as the value of the action item's *Producer* property to accomplish the same effect as the *OnAction* event handler above.

Chaining page producers together

The replacement text from an *OnHTMLTag* event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next.

The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure the message is not considered handled. The next action item uses the *ContentFromString* method of its associated producer to manipulate the content of the Web response message, and so on. Action items after the first one use an *OnAction* event handler such as the following:

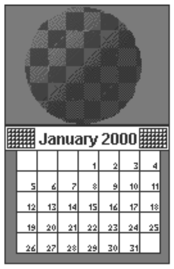
Delphi example

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

C++ example

```
void __fastcall WebModule1::Action2Action(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  Response->Content = PageProducer2->ContentFromString(Response->Content);
}
```

For example, consider an application that returns calendar pages in response to request messages that specify the month and year of the desired page. Each calendar page contains a picture, followed by the name and year of the month between small images of the previous month and next months, followed by the actual calendar. The resulting image looks something like this:



The general form of the calendar is stored in a template file. It looks like this:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

The *OnHTMLTag* event handler of the first page producer looks up the month and year from the request message. Using that information and the template file, it does the following:

- Replaces <#MonthlyImage> with <#Image Month=January Year=2000>.

- Replaces <#TitleLine> with <#Calendar Month=December Year=1999 Size=Small> January 2000 <#Calendar Month=February Year=2000 Size=Small>.
- Replaces <#MainBody> with <#Calendar Month=January Year=2000 Size=Large>.

The *OnHTMLTag* event handler of the next page producer uses the content produced by the first page producer, and replaces the <#Image Month=January Year=2000> tag with the appropriate HTML <IMG> tag. Yet another page producer resolves the #Calendar tags with appropriate HTML tables.

Using database information in responses

The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table.

Adding a session to the Web module

Both console CGI applications and Apache DSO applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application.

Representing database information in HTML

Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers:

- The dataset page producer, which formats the fields of a dataset into the text of an HTML document.
- Table producers, which format the records of a dataset as an HTML table.

Using dataset page producers

Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tag name which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see "Using page producer components" on page 30-13.

To use a dataset page producer, add a *TDataSetPageProducer* component to your Web module and set its *DataSet* property to the dataset whose field values should be displayed in the HTML content. Create an HTML template that describes the output of your dataset page producer. For every field value you want to display, include a tag of the form

```
<#FieldName>
```

in the HTML template, where *FieldName* specifies the name of the field in the dataset whose value should be displayed.

When your application calls the *Content*, *ContentFromString*, or *ContentFromStream* method, the dataset page producer substitutes the current field values for the tags that represent fields.

Using table producers

The Internet palette page includes two components that create an HTML table to represent the records of a dataset:

- Dataset table producers, which format the fields of a dataset into the text of an HTML document.
- Query table producers, which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table.

Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table's color, border, separator thickness, and so on. To set the properties of a table producer at design time, double-click the table producer component to display the Response Editor dialog.

Specifying the table attributes

Table producers use the *THTMLTableAttributes* object to describe the visual appearance of the HTML table that displays the records from the dataset. The *THTMLTableAttributes* object includes properties for the table's width and spacing within the HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer.

At design time, specify these properties using the Object Inspector. Select the table producer object in the Object Inspector and expand the *TableAttributes* property to access the display properties of the *THTMLTableAttributes* object.

You can also specify these properties programmatically at runtime.

Specifying the row attributes

Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The *RowAttributes* property is a *THTMLTableRowAttributes* object.

At design time, specify these properties using the Object Inspector by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime.

You can also adjust the number of rows shown in the HTML table by setting the *MaxRows* property.

Specifying the columns

If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the Object Inspector after selecting them in the Columns editor.

If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at design time. However, you can still customize the columns programmatically at runtime, by setting up the appropriate *THTMLTableColumn* objects and using the methods of the *Columns* property to add them to the table. If you do not set up the *Columns* property, the table producer creates a default set of columns that match the fields of the dataset and specify no special display characteristics.

Embedding tables in HTML documents

You can embed the HTML table that represents your dataset in a larger document by using the *Header* and *Footer* properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table.

You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties.

If you embed your table in a larger document, you may want to add a caption to the table. Use the *Caption* and *CaptionAlignment* properties to give your table a caption.

Setting up a dataset table producer

TDataSetTableProducer is a table producer that creates an HTML table for a dataset. Set the *DataSet* property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally.

You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the HTTP request message.

Setting up a query table producer

You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the *TQuery* object that uses those parameters as the *Query* property of a *TSQLQueryTableProducer*SQL component.

If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message.

When you call the *Content* method of *TSQLQueryTableProducer*, it runs the query, using the parameters it finds in the request object. It then formats an HTML table to display the records in the resulting dataset.

As with any table producer, you can customize the display properties or column bindings of the HTML table, or embed the table in a larger HTML document.

Chapter

31

Creating Web server applications using WebSnap

WebSnap augments Web Broker with additional components, wizards, and views—making it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of application developers and Web designers.

WebSnap allows HTML design experts on your team to make a more effective contribution to Web server development and maintenance. The final product of the WebSnap development process includes a series of scriptable HTML page templates. These pages can be changed using HTML editors that support embedded script tags, like Microsoft FrontPage, or even a simple text editor. Changes can be made to the templates as needed, even after the application is deployed. There is no need to modify the project source code at all, which saves valuable development time. Also, WebSnap’s multiple module support can be used to partition your application into smaller pieces during the coding phases of your project. Application developers can work more independently.

The dispatcher components automatically handle requests for page content, HTML form submissions, and requests for dynamic images. WebSnap components called adapters provide a means to define a scriptable interface to the business rules of your application. For example, the *TDataSetAdapter* object is used to make dataset components scriptable. You can use WebSnap producer components to quickly build complex, data-driven forms and tables, or to use XSL to generate a page. You can use the session component to keep track of end users. You can use the user list component to provide access to user names, passwords, and access rights.

The Web application wizard allows you to quickly build an application that is customized with the components that you will need. The Web page module wizard allows you to create a module that defines a new page in your application. Or use the Web data module wizard to create a container for components that are shared across your Web application.

The page module views show the result of server-side scripting without running the application. You can view the generated HTML in an embedded browser using the Preview tab, or in text form using the HTML Result tab. The HTML Script tab shows the page with server-side scripting, which is used to generate HTML for the page.

The following sections of this chapter explain how you use the WebSnap components to create a Web server application.

Fundamental WebSnap components

Before you can build Web server applications using WebSnap, you must first understand the fundamental components used in WebSnap development. They fall into three categories:

- Web modules, which contain the components that make up the application and define pages
- Adapters, which provide an interface between HTML pages and the Web server application itself
- Page producers, which contain the routines that create the HTML pages to be served to the end user

The following sections examine each type of component in more detail.

Web modules

Web modules are the basic building block of WebSnap applications. Every WebSnap server application must have at least one Web module. More can be added as needed. There are four Web module types:

- Web application page modules (*TWebAppPageModule* objects)
- Web application data modules (*TWebAppDataModule* objects)
- Web page modules (*TWebPageModule* objects)
- Web data modules (*TWebDataModule* objects)

Web page modules and Web application page modules provide content for Web pages. Web data modules and Web application data modules act as containers for components shared across your application; they serve the same purpose in WebSnap applications that ordinary data modules serve in other types of applications. You can include any number of Web page or data modules in your server application.

You may be wondering how many Web modules your application needs. Every WebSnap application needs one (and only one) Web application module of some type. Beyond that, you can add as many Web page or data modules as you need.

For Web page modules, a good rule of thumb is one per page style. If you intend to implement a page that can use the format of an existing page, you may not need a new Web page module. Modifications to an existing page module may suffice. If the page is very different from your existing modules, you will probably want to create a new module. For example, let’s say you are trying to build a server to handle online catalog sales. Pages which describe available products might all share the same Web page module, since the pages can all contain the same basic information types using the same layout. An order form, however, would probably require a different Web page module, since the format and function of an order form is different from that of an item description page.

The rules are different for Web data modules. Components that can be shared by many different Web modules should be placed in a Web data module to simplify shared access. You will also want to place components that can be used by many different Web applications in their own Web data module. That way you can easily share those items among applications. Of course, if neither of these circumstances applies you might choose not to use Web data modules at all. Use them the same way you would use regular data modules, and let your own judgment and experience be your guide.

Web application module types

Web application modules provide centralized control for business rules and non-visual components in the Web application. The two types of Web application modules are tabulated below.

Table 31.1 Web application module types

Web application module type	Description
Page	Creates a content page. The page module contains a page producer which is responsible for generating the content of a page. The page producer displays its associated page when the HTTP request pathinfo matches the page name. The page can act as the default page when the pathinfo is blank.
Data	Used as a container for components shared by other modules, such as database components used by multiple Web page modules.

Web application modules act as containers for components that perform functions for the application as a whole—such as dispatching requests, managing sessions, and maintaining user lists. If you are already familiar with the Web Broker architecture, you can think of Web application modules as being similar to *TWebApplication* objects. Web application modules also contain the functionality of a regular Web module, either page or data, depending on the Web application module type. Your project can contain only one Web application module. You will never need more than one anyway; you can add regular Web modules to your server to provide whatever extra features you want.

Use the Web application module to contain the most basic features of your server application. If your server will maintain a home page of some sort, you may want to make your Web application module a *TWebAppPageModule* instead of a *TWebAppDataModule*, so you don’t have to create an extra Web page module for that page.

Web page modules

Each Web page module has a page producer associated with it. When a request is received, the page dispatcher analyzes the request and calls the appropriate page module to process the request and return the content of the page.

Like Web data modules, Web page modules are containers for components. A Web page module is more than a mere container, however. A Web page module is used specifically to produce a Web page.

All web page modules have an editor view, called Preview, that allows you to preview the page as you are building it. You can take full advantage of the visual application development environment offered by the IDE.

Page producer component

Web page modules have a property that identifies the page producer component responsible for generating content for the page. (To learn more about page producers, see “Page producers” on page 31-6.) The WebSnap page module wizard automatically adds a producer when creating a Web page module. You can change the page producer component later by dropping in a different producer from the WebSnap palette. However, if the page module has a template file, be sure that the content of this file is compatible with the replacement producer component.

Page name

Web page modules have a page name that can be used to reference the page in an HTTP request or within the application's logic. A factory in the Web page module’s unit specifies the page name for the Web page module.

Producer template

Most page producers use a template. HTML templates typically contain some static HTML mixed in with transparent tags or server-side scripting. When page producers create their content, they replace the transparent tags with appropriate values and execute the server-side script to produce the HTML that is displayed by a client browser. (The XSLPageProducer is an exception to this. It uses XSL templates, which contain XSL rather than HTML. The XSL templates do not support transparent tags or server-side script.)

Web page modules may have an associated template file that is managed as part of the unit. A managed template file appears in the Project Manager and has the same base file name and location as the unit service file. If the Web page module does not have an associated template file, the properties of the page producer component specify the template.

Web data modules

Like standard data modules, Web data modules are a container for components from the palette. Data modules provide a design surface for adding, removing, and selecting components. The Web data module differs from a standard data module in the structure of the unit and the interfaces that the Web data module implements.

Use the Web data module as a container for components that are shared across your application. For example, you can put a dataset component in a data module and access the dataset from both:

- a page module that displays a grid, and
- a page module that displays an input form.

You can also use Web data modules to contain sets of components that can be used by several different Web server applications.

Structure of a Web data module unit

Standard data modules have a variable called a form variable, which is used to access the data module object. Web data modules replace the variable with a function, which is defined in a Web data module's unit and has the same name as the Web data module. The function's purpose is the same as that of the variable it replaces. WebSnap applications may be multi-threaded and may have multiple instances of a particular module to service multiple requests concurrently. Therefore, the function is used to return the correct instance.

The Web data module unit also registers a factory to specify how the module should be managed by the WebSnap application. For example, flags indicate whether to cache the module and reuse it for other requests or to destroy the module after a request has been serviced.

Adapters

Adapters define a script interface to your server application. They allow you to insert scripting languages into a page and retrieve information by making calls from your script code to the adapters. For example, you can use an adapter to define data fields to be displayed on an HTML page. A scripted HTML page can then contain HTML content and script statements that retrieve the values of those data fields. This is similar to the transparent tags used in Web Broker applications. Adapters also support actions that execute commands. For example, clicking on a hyperlink or submitting an HTML form can initiate adapter actions.

Adapters simplify the task of creating HTML pages dynamically. By using adapters in your application, you can include object-oriented script that supports conditional logic and looping. Without adapters and server-side scripting, you must write more of your HTML generation logic in Delphi or C++ event handlers. Using adapters can significantly reduce development time.

See "Server-side scripting in WebSnap"on page 31-31 and "Dispatching requests and responses" on page 31-34 for more details about scripting.

Four types of adapter components can be used to create page content: fields, actions, errors and records.

Fields

Fields are components that the page producer uses to retrieve data from your application and to display the content on a Web page. Fields can also be used to retrieve an image. In this case, the field returns the address of the image written to the Web page. When a page displays its content, a request is sent to the Web server application, which invokes the adapter dispatcher to retrieve the actual image from the field component.

Actions

Actions are components that execute commands on behalf of the adapter. When a page producer generates its page, the scripting language calls adapter action components to return the name of the action along with any parameters necessary to execute the command. For example, consider clicking a button on an HTML form to delete a row from a table. This returns, in the HTTP request, the action name associated with the button and a parameter indicating the row number. The adapter dispatcher locates the named action component and passes the row number as a parameter to the action.

Errors

Adapters keep a list of errors that occur while executing an action. Page producers can access this list of errors and display them in the Web page that the application returns to the end user.

Records

Some adapter components, such as *TDataSetAdapter*, represent multiple records. The adapter provides a scripting interface which allows iteration through the records. Some adapters support paging and iterate only through the records on the current page.

Page producers

Page producers to generate content on behalf of a Web page module. Page producers provide the following functionality:

- They generate HTML content.
- They can reference an external file using the HTMLFile property, or the internal string using the HTMLDoc property.
- When the producers are used with a Web page module, the template can be a file associated with a unit.
- Producers dynamically generate HTML that can be inserted into the template using transparent tags or scripting. Transparent tags can be used in the same way as WebBroker applications. To learn more about using transparent tags, see “Converting HTML-transparent tags” on page 30-14. Scripting support allows you to embed commands in a scripting language (such as JavaScript) inside the HTML page.

The standard WebSnap method for using page producers is as follows. When you create a Web page module, you must choose a page producer type in the Web Page Module wizard. You have many choices, but most WebSnap developers prototype their pages by using an adapter page producer, *TAdapterPageProducer*. The adapter page producer lets you build a prototype Web page using a process analogous to the standard component model. You add a type of form, an adapter form, to the adapter page producer. As you need them, you can add adapter components (such as adapter grids) to the adapter form. Using adapter page producers, you can create Web pages in a way that is similar to the technique for building user interfaces in other types of applications.

There are some circumstances where switching from an adapter page producer to a regular page producer is more appropriate. For example, part of the function of an adapter page producer is to dynamically generate script in a page template at runtime. You may decide that static script would help optimize your server. Also, users who are experienced with script may want to make changes to the script directly. In this case, a regular page producer must be used to avoid conflicts between dynamic and static script. To learn how to change to a regular page producer, see "Advanced HTML design" on page 31-22

You can also use page producers the same way you would use them in Web Broker applications, by associating the producer with a Web dispatcher action item. The advantages of using the Web page module are

- the ability to preview the page's layout without running the application, and
- the ability to associate a page name with the module, so that the page dispatcher can call the page producer automatically.

Creating Web server applications with WebSnap

If you look at the source code for WebSnap, you will discover that WebSnap comprises hundreds of objects. In fact, WebSnap is so rich in objects and features that you could spend a long time studying its architecture in detail before understanding it completely. Fortunately, you really don't need to understand the whole WebSnap system before you start developing your server application.

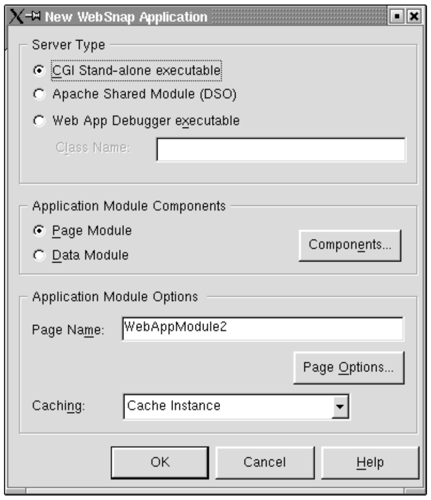
Here you will learn more about how WebSnap works by creating a new Web server application.

To create a new Web server application using the WebSnap architecture:

1. Choose File | New | Other.
2. In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.

   A dialog box appears (as shown in Figure 31.1).
3. Specify the correct server type.
4. Use the components button to specify application module components.
5. Use the Page Options button to select application module options.

Figure 31.1 New WebSnap application dialog box


New WebSnap Application

Server Type
- [✓ CGI Stand-alone executable]
- Apache Shared Module (DSO)
- Web App Debugger executable

Class Name:

Application Module Components
- [✓ Page Module]
- Data Module

Application Module Options

Page Name: WebAppModule2

Caching: Cache Instance

Selecting a server type

Select one of the following types of Web server application, depending on your application's type of Web server.

Table 31.2 Web server application types

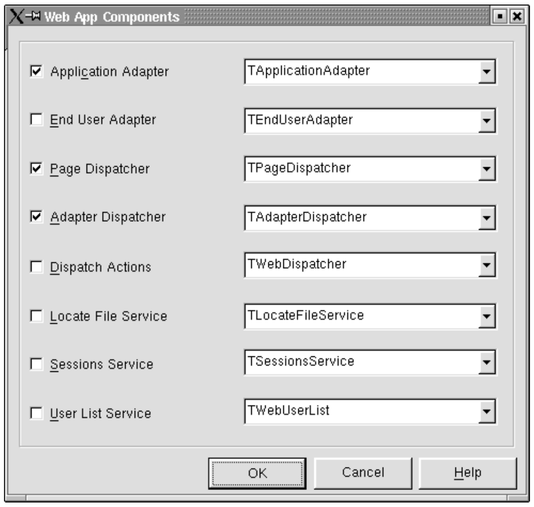
Server type	Description
CGI stand-alone	Sets up your project as a console application which conforms to the Common Gateway Interface (CGI) standard.
Apache	Sets up your project as a DSO with the exported methods expected by the Apache Web server.
Web App Debugger executable	Creates an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Specifying application module components

Application components provide the Web application's functionality. For example, including an adapter dispatcher component automatically handles HTML form submissions and the return of dynamically generated images. Including a page dispatcher automatically displays the content of a page when the HTTP request pathinfo contains the name of the page.

Selecting the Components button on the new WebSnap application dialog (see Figure 31.1) displays another dialog that allows you to select one or more of the Web application module components. The dialog, which is called the Web App Components dialog, is shown in Figure 31.2.

Figure 31.2 Web App Components dialog



☑ Application Adapter	TApplicationAdapter
☐ End User Adapter	TEndUserAdapter
☑ Page Dispatcher	TPageDispatcher
☑ Adapter Dispatcher	TAdapterDispatcher
☐ Dispatch Actions	TWebDispatcher
☐ Locate File Service	TLocateFileService
☐ Sessions Service	TSessionsService
☐ User List Service	TWebUserList

The following table contains a brief explanation of the available components:

Table 31.3 Web application components

Component type	Description
Application Adapter	Contains information about the application, such as the title. The default type is *TApplicationAdapter*.
End User Adapter	Contains information about the user, such as their name, access rights, and whether they are logged in. The default type is *TEndUserAdapter*. *TEndUserSessionAdapter* may also be selected.
Page Dispatcher	Examines the HTTP request's pathinfo and calls the appropriate page module to return the content of a page. The default type is *TPageDispatcher*.
Adapter Dispatcher	Automatically handles HTML form submissions and requests for dynamic images by calling adapter action and field components. The default type is *TAdapterDispatcher*.
Dispatch Actions	Allows you to define a collection of action items to handle requests based on pathinfo and method type. Action items call user-defined events or request the content of page-producer components. The default type is *TWebDispatcher*.
Locate File Service	Provides control over the loading of template files, and script include files, when the Web application is running. The default type is *TLocateFileService*.
Sessions Service	Stores information about end users that is needed for a short period of time. For example, you can use sessions to keep track of logged-in users and to automatically log a user out after a period of inactivity.
User List Service	Keeps track of authorized users, their passwords, and their access rights. The default type is *TWebUserList*.

For each of the above components, the component types listed are the default types. Users can create their own component types or use third-party component types.

Selecting Web application module options

If the selected application module type is a page module, you can associate a name with the page by entering a name in the Page Name field in the New WebSnap Application dialog box. At runtime, the instance of this module can be either kept in cache or removed from memory when the request has been serviced. Select either of the options from the Caching field. You can select more page module options by choosing the Page Options button. The Application Module Page Options dialog is displayed and provides the following categories:

- Producer: The producer type for the page can be set to one of *AdapterPageProducer*, *DataSetPageProducer*, *PageProducer*, or *XSLPageProducer*. If the selected page producer supports scripting, then use the Script Engine drop-down list to select the language used to script the page.

Note The *AdapterPageProducer* supports only JavaScript (which is labeled as JScript in the Script Engine drop-down).

- HTML: When the selected producer uses an HTML template this group will be visible.
- XSL: When the selected producer uses an XSL template, such as *TXSLPageProducer*, this group will be visible.
- New File: Check New File if you want a template file to be created and managed as part of the unit. A managed template file appears in the Project Manager and has the same file name and location as the unit source file. Uncheck New File if you want to use the properties of the producer component (typically the *HTMLDoc* or *HTMLFile* property).
- Template: When New File is checked, choose the default content for the template file from the Template drop-down. The standard template displays the title of the application, the title of the page, and hyperlinks to published pages. The blank template creates a blank page.
- Page: Enter a page name and title for the page module. The page name is used to reference the page in an HTTP request or within the application's logic, whereas the title is the name that the end user will see when the page is displayed in a browser.
- Published: Check Published to allow the page to automatically respond to HTTP requests where the page name matches the pathinfo in the request message.
- Login Required: Check Login Required to require the user to log on before the page can be accessed.

You have now learned how to begin creating a WebSnap server application. The WebSnap tutorial in the next section describes how to develop a more complete application.

WebSnap tutorial

This tutorial provides step-by-step instructions on building a WebSnap application. The WebSnap application demonstrates how to use WebSnap HTML components to build an application that edits the content of a table. Follow the instructions from beginning to end. If you need a break, you can use File | Save All at any time to save your project.

Create a new application

This topic describes how to create a new WebSnap application called Country Tutorial. It displays a table of information about various countries to users on the Web. Users can add and delete countries and edit information about existing countries.

Step 1. Start the WebSnap application wizard

1. Run the IDE and choose File | New | Other.
2. In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.
3. In the New WebSnap Application dialog box:
   - Select Web App Debugger executable.
   - In the Class Name edit control, type CountryTutorial.
   - Select Page Module as the component type.
   - In the Page Name field, type Home.
4. Click OK.

Step 2. Save the generated files and project

To save the Pascal unit file and project:

1. Select File | Save All.
2. In the Save dialog, change to an appropriate directory where you can save all of the files in the Country Tutorial project.
3. In the File name field enter HomeU.pas (Delphi) or HomeU.cpp (C++) and click Save.
4. In the File name field enter CountryU.pas (Delphi) or CountryU.cpp (C++) and click Save.
5. In the File name field enter CountryTutorial.dpr (Delphi) or CountryTutorial.bpr (C++) and click Save.

Note Save the units and the project in the same directory because the application looks for the HomeU.html file in the same directory as the executable.

Step 3. Specify the application title

The application title is the name displayed to the end user. To specify the application title:

1 Choose View | Project Manager.

2 In the Project Manager expand CountryTutorial and double-click the HomeU entry.

3 In the top line of the Object Inspector window, select ApplicationAdapter from the pull-down list.

4 In the Properties tab, enter Country Tutorial for the ApplicationTitle property.

5 Click the Preview tab in the editor window. The application title is displayed at the top of the page with the page name, Home.

This page is very basic. You can improve it by editing the HomeU.html file, either by using the HomeU.html editor tab or by using an external editor. For more information on how to edit the page template, see “Advanced HTML design” on page 31-22.

Create a CountryTable page

A Web page module defines a published page, and it also acts as a container for data components. Whenever a Web page needs to be returned to the end user, the Web page module extracts the necessary information from the data components it contains and uses that information to help create a page.

Here you will add a new page module to the WebSnap application. The page module will add a second viewable page to the Country Tutorial application. The first page, Home, was defined when you created the application. The second page, called CountryTable, shows the table of country information.

Step 1. Add a new Web page module

To add a new page module:

1 Choose File | New | Other.

2 In the New Items dialog box, select the WebSnap tab, choose WebSnap Page Module, and click OK.

3 In the dialog box, set the Producer Type to AdapterPageProducer.

4 In the Page Name field enter CountryTable. Notice that the Title also changes as you type.

5 Leave the rest of the fields with their default values.

The dialog should appear as shown in Figure 31.3.

6 Click OK.

Figure 31.3 New WebSnap Page Module dialog box for the CountryTable page

The CountryTable module should now appear in the IDE. After saving the module, you will add new components to the CountryTable module.

Step 2. Save the new Web page module

Save the unit to the directory of the project file. When the application runs, it searches for the CountryTableU.html file in the same directory as the executable.

1. Choose File | Save.
2. In the File name field, enter CountryTableU.pas (Delphi) or CountryTableU.cpp (C++) and click Save.

Add data components to the CountryTable module

A *TClientDataSet* component provides the data for the HTML table. The *TDataSetAdapter* component allows server side script to access the *TClientDataSet* component. Here we will add these data-aware components to our application.

Steps 1 and 2 below assume some basic familiarity with database programming, but you don't need it in order to complete this tutorial. WebSnap acts only as an interface (through adapter components) to database components. To learn more about database programming, you can refer to Part II of this manual.

Step 1. Add data-aware components

1. Use a tool outside of the IDE (such as a command line or a file manager of your choice), place a copy of country.xml in the directory where your project files are stored. The file country.xml is stored in the demos/delphi/db/data directory of your product installation.
2. Change the permissions on country.xml so that all users have write permission. For example, you can use the following command in a shell window:

   ```
   chmod a+w country.xml
   ```
3. Return to the IDE and choose View | Project Manager.
4. In the Project Manager window, expand CountryTutorial and double-click the CountryTableU entry.
5. Select the Data Access tab in the Component palette.
6. Select a ClientDataSet component and add it to the CountryTable Web module.
7. Select the ClientDataSet component in the Web page module window. This displays the ClientDataSet component values in the Object Inspector.
8. In the Object Inspector, change the following properties:
   - In the *Name* property, type Country.
   - Set the *FileName* property to country.xml.

- Set the *Active* property to true.

Step 2. Specify a key field

The key field is used to identify records within a client data set. This becomes important when you add an edit page to the application. To specify a key field:

1. Double-click the Country component in the CountryTable module to display the Fields Editor.
2. Right-click the editor window and select the Add All Fields command.
3. Select the Name field from the list of added fields.
4. In the Object Inspector, expand the *ProviderFlags* property.
5. Set the *pfInKey* property value to true.

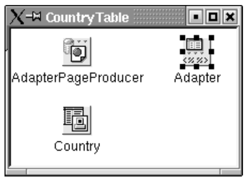
Step 3. Add an adapter component

You are finished adding database components. Now, to expose the data in the *TClientDataSet* through server-side scripting, you must include a dataset adapter (*TDataSetAdapter*) component. To add a dataset adapter:

1. Choose the DataSetAdapter component from the WebSnap tab of the component palette. Add it to the CountryTable Web module.
2. Change the following properties in the Object Inspector:
   - Set the *DataSet* property to Country.
   - Set the *Name* property to Adapter.

   When you are finished, the CountryTable Web page module should look similar to what is shown in Figure 31.4.

Figure 31.4 CountryTable Web page module



Since the elements in the module aren't visual, it doesn't matter where they appear in the module. What matters is that your module contains the same components as shown in the figure.

Create a grid to display the data

Step 1. Add a grid

Now, add a grid to display the data from the country table database:

1. Choose View | Project Manager.

2 In the Project Manager, expand CountryTutorial and double-click the CountryTableU entry.

3 Double-click the AdapterPageProducer component in the CountryTable module. This component generates server-side script to quickly build an HTML table. After you double-click it, a Web page editor will appear.

4 Right-click the AdapterPageProducer entry and choose New Component.

5 In the Add Web Component dialog box, select AdapterForm, then click OK. An AdapterForm1 component appears in the Web page editor.

6 Right-click AdapterForm1 and select New Component.

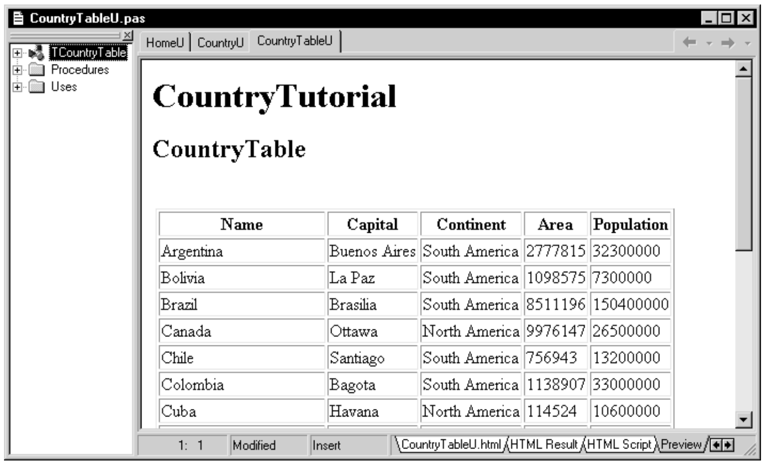
7 In the Add Web Component window, select AdapterGrid then click OK. An AdapterGrid1 component appears in the Web page editor.

8 Right-click the AdapterGrid1 component and choose Add All Columns. Five columns are added to the adapter group.

9 In the Object Inspector window, set the Adapter property to Adapter.

To preview the page, select the CountryTableU tab at the top of the code editor, then select the Preview tab at the bottom. If the Preview tab is not shown, use the right arrow at the bottom to scroll through the tabs. The preview should appear similar to Figure 31.5.

Figure 31.5 CountryTable Preview tab



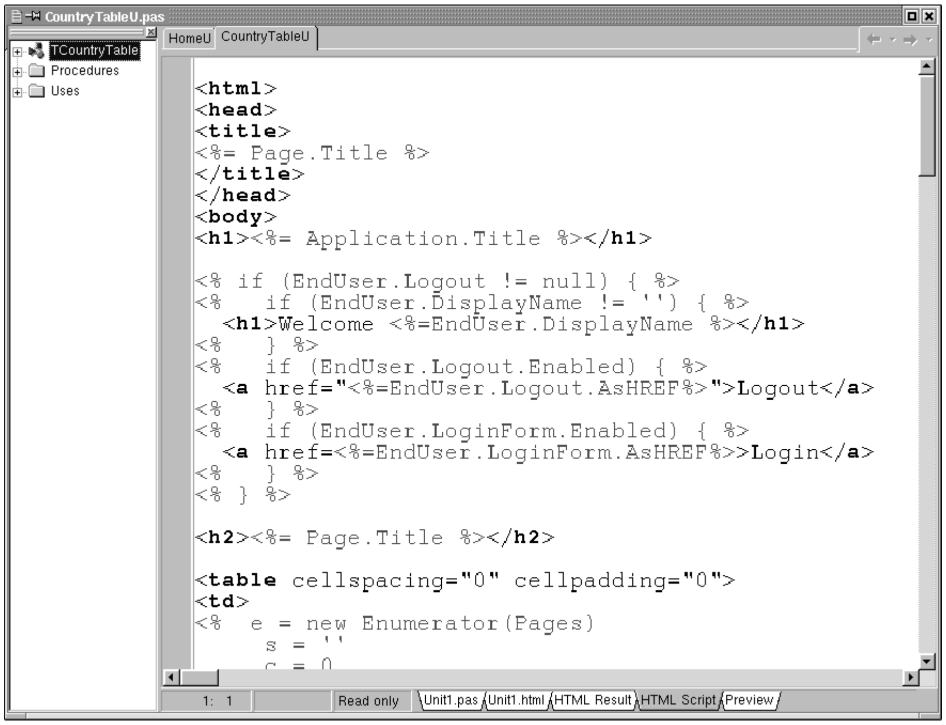
CountryTutorial

CountryTable

Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32300000
Bolivia	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000

The Preview tab shows you what the final, static HTML page looks like in a Web browser. That page is generated from a dynamic HTML page that includes script. To see a text representation showing the script commands, select the HTML Script tab at the bottom of the editor window (see Figure 31.6).

Figure 31.6 CountryTable HTML Script tab



```
<html>
<head>
<title>
<%= Page.Title %>
</title>
</head>
<body>
<h1><%= Application.Title %></h1>

<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
<%   if (EndUser.Logout.Enabled) { %>
  <a href="<%=EndUser.Logout.AsHREF%>">Logout</a>
<%   } %>
<%   if (EndUser.LoginForm.Enabled) { %>
  <a href=<%=EndUser.LoginForm.AsHREF%>>Login</a>
<%   } %>
<% } %>

<h2><%= Page.Title %></h2>

<table cellspacing="0" cellpadding="0">
<td>
<%  e = new Enumerator(Pages)
      s = ''
      c = 0
```

The HTML Script tab shows a mixture of HTML and script. HTML and script are differentiated in the editor by font color and attributes. By default, HTML tags appear in boldfaced black text, while HTML attribute names appear in black and HTML attribute values appear in blue. Script, which appears between the script brackets `<% %>`, is colored green. You can change the default font colors and attributes for these items in the Color tab of the Editor Properties dialog, which can be displayed by right-clicking on the editor and selecting Properties.

There are two other HTML-related editor tabs. The HTML Result tab shows the raw HTML contents of the preview. Note that the HTML Result, HTML Script, and Preview views are all read-only. You can use the last HTML-related editor tab, CountryTable.html, for editing.

If you want to improve the look of this page, you can add HTML using either the CountryTable.html tab or an external editor at any time. For more information on how to edit the page template, see “Advanced HTML design” on page 31-22.

Step 2. Add editing commands to the grid

Users may need to update the content of the table by deleting, inserting or editing a row. To allow users to make such updates, add command components.

To add command components:

1 In the Web page editor for the AdapterPageProducer component, expand the AdapterPageProducer component and all its branches.

2 Right-click the AdapterGrid1 component and choose New Component.

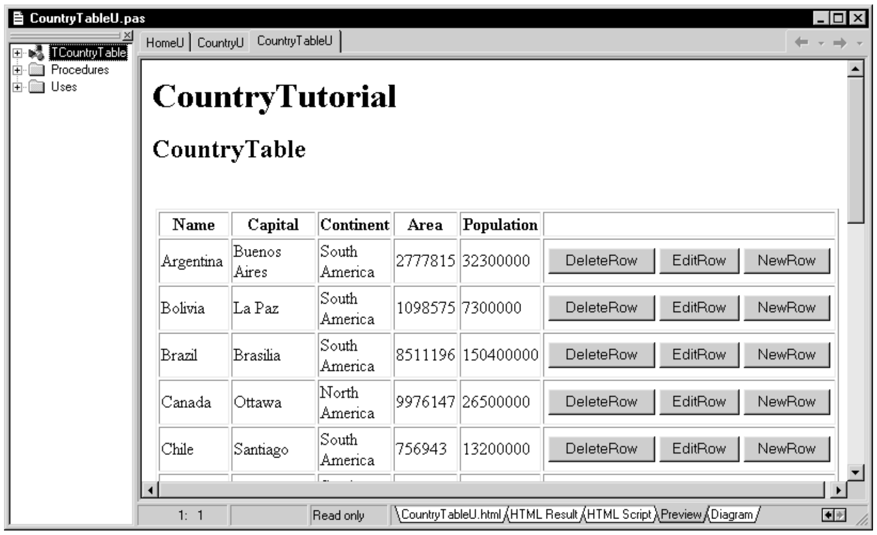
3 Select AdapterCommandColumn and then click OK. An AdapterCommandColumn1 entry is added to the AdapterGrid1 component.

4 Right-click AdapterCommandColumn1 and choose Add Commands.

5 Multi-select the DeleteRow, EditRow, and NewRow commands, then click OK.

6 To preview the page, click the Preview tab at the bottom of the code editor. There are now three new buttons (DeleteRow, EditRow, and NewRow) at the end of each row in the table, as shown in Figure 31.7. When the application is running, pressing one of these buttons performs the associated action.

Figure 31.7 CountryTable Preview after editing commands have been added



CountryTutorial

CountryTable

Name	Capital	Continent	Area	Population			
Argentina	Buenos Aires	South America	2777815	32300000	DeleteRow	EditRow	NewRow
Bolivia	La Paz	South America	1098575	7300000	DeleteRow	EditRow	NewRow
Brazil	Brasilia	South America	8511196	150400000	DeleteRow	EditRow	NewRow
Canada	Ottawa	North America	9976147	26500000	DeleteRow	EditRow	NewRow
Chile	Santiago	South America	756943	13200000	DeleteRow	EditRow	NewRow

Click the Save All button to save the application before continuing.

Add an edit form

You can now create a Web page module to handle the Edit form for the country table. Users will be able to change data in the CountryTutorial application through the Edit form. Specifically, when the user presses the EditRow or NewRow buttons, an Edit form appears. When the user is finished with the Edit form, the modified information automatically appears in the table.

Step 1. Add a new Web page module

To add a new Web page module:

1 Choose File | New | Other.

2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Page Module.

3 In the dialog box, set the Producer Type to AdapterPageProducer.

4 Set the Page Name field to CountryForm.

5 Uncheck the Published box, so this page does not appear in a list of available pages on this site. The Edit form is accessed through the Edit button, and its contents depend on which row of the country table is to be modified.

6 Leave the rest of the fields and selections set to their default values. Click OK.

Step 2. Save the new module

Save the module in the same directory as the project file. When the application runs, it looks for the CountryFormU.html file in the same directory as the executable.

1 Choose File | Save.

2 In the File name field enter CountryFormU.pas (Delphi) or CountryFormU.cpp (C++) and click OK.

Step 3. Make CountryTableU accessible to the new module

Add CountryTableU unit to the uses clause (Delphi) or include directives (C++) to allow the module access to the Adapter component.

1 Choose File | Use Unit (Delphi) or File | Include Unit Hdr (C++).

2 Choose CountryTableU from the list then click OK.

3 Choose File | Save.

Step 4. Add input fields

Add components to the AdapterPageProducer component to generate data entry fields in the HTML form.

To add input fields:

1 Choose View | Project Manager.

2 In the Project Manager window, expand CountryTutorial and double-click the CountryFormU entry.

3 In the CountryForm module, double-click the AdapterPageProducer component to expose its Web page editor.

4 In the Web page editor, right-click AdapterPageProducer and select New Component.

5 Select AdapterForm, then click OK. An AdapterForm1 entry appears in the Web page editor.

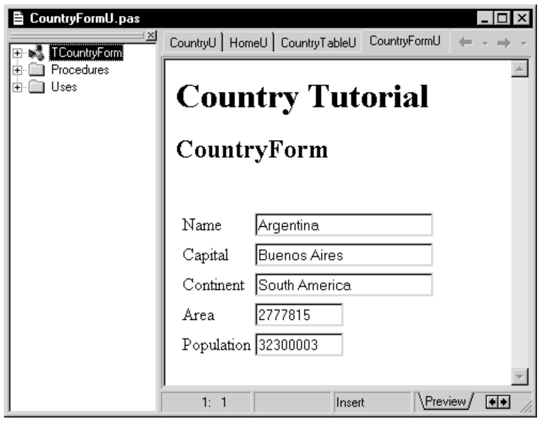
6 Right-click AdapterForm1 and select New Component.

7 Select AdapterFieldGroup then click OK. An AdapterFieldGroup1 entry appears in the Web page editor.

8 In the Object Inspector window, set the *Adapter* property to CountryTable.Adapter (Delphi) or CountryTable->Adapter (C++). Set the *AdapterMode* property to Edit.

9 To preview the Page, click the Preview tab at the bottom of the code editor. Your preview should resemble the one shown in Figure 31.8.

Figure 31.8 Previewing the CountryForm



Country Tutorial

CountryForm

Name	Argentina
Capital	Buenos Aires
Continent	South America
Area	2777815
Population	32300003

Step 5. Add buttons

Add components to the AdapterPageProducer component to generate the submit buttons in the HTML form. To add components:

1 Right-click AdapterForm1 in the Web page editor and choose New Component.

2 Select AdapterCommandGroup then click OK. An AdapterCommandGroup1 entry appears in the Object TreeView.

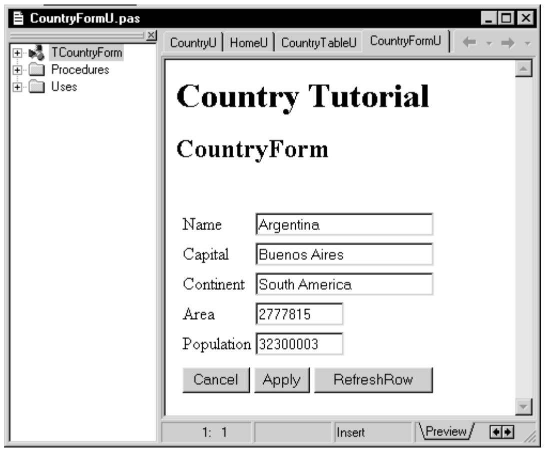
3 In the Object Inspector, set the *DisplayComponent* property to AdapterFieldGroup1.

4 Right-click AdapterCommandGroup1 entry and choose Add Commands.

5 Multi-select the Cancel, Apply, and Refresh Row commands, then click OK.

6 To preview the page, click the Preview tab at the bottom of the code editor window. If the preview does not show the country form, click the Code tab and then click the Preview tab again. Your preview should resemble the one shown in Figure 31.9.

Figure 31.9 CountryForm with submit buttons



Argentina
Buenos Aires
South America
2777815
32300003

Step 6. Link form actions to the grid page

When the user clicks a button, an adapter action is executed which in turn carries out the described action. To specify which page to display after an adapter action is executed:

1. Click on the AdapterCommandGroup1 to show the CmdCancel, CmdApply, and CmdRefreshRow entries in the top right pane of the Web page editor window.
2. Select CmdCancel. In the Object Inspector window, type CountryTable in the *PageName* property.
3. Select CmdApply. In the Object Inspector window, type CountryTable in the *PageName* property.

Step 7. Link grid actions to the form page

An adapter action is executed by pushing a button in the grid. To specify which page is displayed in response to the adapter action:

1. Choose View | Project Manager.
2. In the Project Manager, expand CountryTutorial and double-click the CountryTableU entry.
3. In the CountryTable module, double-click the *AdapterPageProducer* component to show its Web page editor. Click the AdapterCommandColumn1 item to show the CmdNewRow, CmdEditRow, and CmdDeleteRow entries in the top right pane of the Web page editor.
4. Select CmdNewRow. In the Object Inspector, type CountryForm in the *PageName* property.
5. Select CmdEditRow. In the Object Inspector, type CountryForm in the *PageName* property.

To verify that the application is working and that all buttons perform some action, run the application. To learn how to run the application, see "Run the completed application" on page 31-21. When you run the application, you are running a server. To check that the application is working, you must view it in a Web browser. You can do this by launching it from the Web App Debugger.

Note There will be no indication of database errors, such as an invalid type. For example, try adding a new country with an invalid value (for example, 'abc') in the Area field.

Run the completed application

To run the completed application:

1 Choose Run | Run. A form is displayed.

2 Choose Tools | Web App Debugger.

3 Click the default URL link to display the ServerInfo page. The ServerInfo page displays the names of all registered Web App Debugger executables.

4 Choose CountryTutorial in the drop-down list and click on the Go button.

Your browser now shows the Country Tutorial application. Click on the CountryTable link to see the CountryTable page.

Add error reporting

Right now, your application won't show any errors to the user. For example, if someone types letters into the Area field of a country record, that user will receive no notification that an error occurred. You will now add an *AdapterErrorList* component to display errors that occur while executing adapter actions which edit the country table.

Step 1. Add error support to the grid

To add error support to the grid:

1 In the CountryTable module, double-click the AdapterPageProducer component to show its Web page editor.

2 Right-click AdapterForm1 and choose New Component.

3 Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Web page editor.

4 Move AdapterErrorList1 above AdapterGrid1 (either by dragging it or by using the up arrow button in the Web page editor toolbar).

5 In the Object Inspector, set the *Adapter* property to Adapter.

Step 2. Add error support to the form

To add error support to the form:

1 In the CountryForm module, double-click the AdapterPageProducer component to show its Web page editor.
2 Right-click AdapterForm1 and choose New Component.
3 Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Web page editor.
4 Move AdapterErrorList1 above AdapterFieldGroup1 (either by dragging it or by using the up arrow in the Web page editor toolbar).
5 In the Object Inspector, set the *Adapter* property to CountryTable.Adapter (Delphi) or CountryTable->Adapter (C++).

Step 3. Test the error-reporting mechanism

To observe the error-reporting mechanism, you must first make a small change to the IDE. Select Tools | Debbugger Options. In the Language Exceptions tab, make sure the Stop on Delphi Exceptions checkbox is unchecked, which will allow the application to proceed when exceptions are detected. Now, to test for grid errors:

1 Run the application and browse to the CountryTable page using the Web App Debugger. For information on how to do this, see "Run the completed application" on page 31-21.
2 Open another browser window and browse to the CountryTable page.
3 Click the DeleteRow button on the first row in the grid.
4 Without refreshing the second browser, click the DeleteRow button on the first row in the grid.

An error message, "Row not found in Country," is displayed above the grid.

To test for form errors:

1 Run the application, and browse to the CountryTable page using the Web App Debugger.
2 Click on the EditRow Button. The CountryForm page is displayed.
3 Change the area field to 'abc', and click the Apply Button.

An error message ("Invalid value for field 'Area'") will be displayed above the first field.

You have now completed the WebSnap tutorial. You might want to recheck the Stop on Delphi Exceptions checkbox before continuing. Also, save the application by choosing File | Save All so your completed application is available for future reference.

Advanced HTML design

Using adapters and adapter page producers, WebSnap makes it easy to create scripted HTML pages in your Web server application. You can create a Web front end for your application data using WebSnap tools that may suit all of your needs.

One powerful feature of WebSnap, however, is the ability to incorporate Web design expertise from other sources into your application. This section discusses some strategies for expanding the Web server design and maintenance process to include other tools and non-programmer team members.

The end products of WebSnap development are your server application and HTML templates for the pages that the server produces. The templates include a mixture of scripting and HTML. Once they have been generated initially, they can be edited at any time using any HTML tool you like. (It would be best to use a tool that supports embedded script tags, like Microsoft FrontPage, to ensure that the editor doesn't accidentally damage the script.) The ability to edit template pages outside of the IDE can be used many ways.

For example, application developers can edit the HTML templates at design time using any external editor they prefer. This allows them to use advanced formatting and scripting features that may be present in an external HTML editor but not in the IDE. To enable an external HTML editor from the IDE, use the following steps:

1 From the main menu, select Tools | Environment Options. In the Environment Options dialog, click on the Internet tab.
2 In the Internet File Types box, select HTML and click the Edit button to display the Edit Type dialog box.
3 In the Edit Action box, select an action associated with your HTML editor. For example, to select the default HTML editor on your system, choose Edit from the drop-down list. Click OK twice to close the Edit Type and Environment Options dialog boxes.

To edit an HTML template, open the unit which contains that template. In the Edit window, right-click and select html Editor from the context menu. The HTML editor displays the template for editing in a separate window. The editor runs independent of the IDE; save the template and close the editor when you're finished.

After the product has been deployed, you may wish to change the look of the final HTML pages. Perhaps your software development team is not even responsible for the final page layout. That duty may belong to a dedicated Web page designer in your organization, for example. Your page designers may not have any experience with application development. Fortunately, they don't have to. They can edit the page templates at any point in the product development and maintenance cycle, without ever changing the source code. Thus, WebSnap HTML templates can make server development and maintenance more efficient.

Manipulating server-side script in HTML files

HTML in page templates can be modified at any time in the development cycle. Server-side scripting can be a different matter, however. It is always possible to manipulate the server-side script in the templates outside of the IDE, but it is not recommended for pages generated by an adapter page producer. The adapter page producer is different from ordinary page producers in that it can change the server-side scripting in the page templates at runtime. It can be difficult to predict how your script will act if other script is added dynamically. If you want to manipulate script directly, make sure that your Web page module contains a page producer instead of an adapter page producer.

If you have a Web page module that uses an adapter page producer, you can convert it to use a regular page producer instead by using the following steps:

1 In the module you want to convert (let's call it ModuleName), copy all of the information from the HTML Script tab to the ModuleName.html tab, replacing all of the information that it contained previously.
2 Drop a page producer (located on the Internet tab of the component palette) onto your Web page module.
3 Set the page producer's *ScriptEngine* property to match that of the adapter page producer it replaces.
4 Change the page producer in the Web page module from the adapter page producer to the new page producer. Click on the Preview tab to verify that the page contents are unchanged.
5 The adapter page producer has now been bypassed. You may now delete it from the Web page module.

Login support

Many Web server applications require login support. For example, a server application may require a user to login before granting access to some parts of a Web site. Pages may have a different appearance for different users; logins may be necessary to enable the Web server to send the right pages. Also, because servers have physical limitations on memory and processor cycles, server applications sometimes need the ability to limit the number of users at any given time.

With WebSnap, incorporating login support into your Web server application is fairly simple and straightforward. In this section, you will learn how you can add login support, either by designing it in from the beginning of your development process or by retrofitting it onto an existing application.

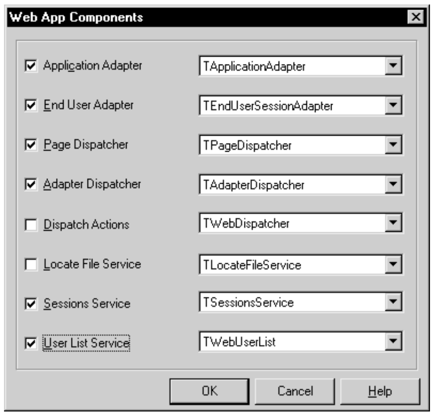
Adding login support

In order to implement login support, you need to make sure your Web application module has the following components:

- A user list service (an object of type *TWebUserList*), which contains the usernames, passwords and permissions for server users
- A sessions service (*TSessionsService* or *TCookieSessionsService*), which stores information about users currently logged in to the server
- An end user adapter (*TEndUserSessionAdapter*) which handles actions associated with logging in

When you first create your Web server application, you can add these components using the New WebSnap Application dialog box. Click the Components button on that dialog to display the New Web App Components dialog box. Check the End User Adapter, Sessions Service and Web User List boxes. Select *TEndUserSessionAdapter* on the drop down menu next to the End User Adapter box to select the end user adapter type. (The default choice, *TEndUserAdapter*, is not appropriate for login support because it cannot track the current user.) When you're finished, your dialog should look similar to the one shown below. Click OK twice to dismiss the dialog boxes. Your Web application module now has the necessary components for login support.

Figure 31.10 Web App Components dialog with options for login support selected



☑ Application Adapter	TApplicationAdapter
☑ End User Adapter	TEndUserSessionAdapter
☑ Page Dispatcher	TPageDispatcher
☑ Adapter Dispatcher	TAdapterDispatcher
☐ Dispatch Actions	TWebDispatcher
☐ Locate File Service	TLocateFileService
☑ Sessions Service	TSessionsService
☑ User List Service	TWebUserList

If you are adding login support to an existing Web application module, you can drop these components directly into your module from the WebSnap tab of the component palette. The Web application module will configure itself automatically.

The sessions service and the end user adapter may not require your attention during your design phase, but the Web user list probably will. You can add default users and set their read/modify permissions through the WebUserList component editor. Double-click on the component to display an editor which lets you set usernames, passwords and access rights. For more information on how to set up access rights, see "User access rights" on page 31-29.

Using the sessions service

The sessions service keeps track of the users who are logged into your Web server application. The sessions service is responsible for assigning a different session for each user and for associating name/value pairs (such as a username) with a user. There are two object types which can function as a sessions service: *TSessionsService* and *TCookieSessionsService*. They differ in how they store session information.

TSessionsService

Information contained in a *TSessionsService* is stored in the application's memory. The Web server application must keep running between requests for the *TSessionsService* to work. Also, if the Web server application is divided up into several processes, all of the different processes must share one memory space for session information.

Some server application types, such as CGI, terminate between requests. Others, such as Apache DSOs in Linux, don't allow forked processes to share memory easily. Unfortunately, these are the only deployable target types available on Linux systems at the time of this writing. Clearly, Linux users need to use another sessions service component type.

Note On Windows, if your Web App Debugger application uses a *TSessionsService*, you must have the application running in the background before it receives a page request. Otherwise it will terminate after each page request, and you will never be able to navigate past the login page. (This restriction doesn't apply on Linux.)

TCookieSessionsService

The Linux component *TCookieSessionsService* obtains its session information from cookies stored in the user's Web browser. These cookies are managed by the browser itself, not by the Web server application.

Cookies are a common, well-documented method of session management. They may have performance and security implications for your Web server application project. An in-depth discussion of cookies is beyond the scope of this work.

TCookieSessionsService is not preinstalled on the component palette. You can find it in the Delphi language WebSnap demos directory. Install it using the Component | Install Component. The unit file name is WebCookieSess.pas, and the package file name is WebCookieSessPackage.dpk. Click OK. A package window appears: click its Install button. After installation, *TCookieSessionsService* will appear on the WebSnap component palette.

To use *TCookieSessionsService* instead of *TSessionsService*, first create your application using a *TSessionsService*. Drop a *TCookieSessionsService* on the Web application module and delete the *TSessionsService* component. Finally, set the *Domain* property of the *TCookieSessionsService* to match the domain of your server. The *TCookieSessionsService* component will now function as the sessions service for your application.

Changing sessions service behavior

There are two important properties in the sessions service which you can use to change default server behavior. The *DefaultTimeout* property specifies the defaut time-out period in minutes. After *DefaultTimeout* minutes have passed without any user activity, the session is automatically terminated. If the user had logged in, all login information is lost.. The default value is 20. You can override the default value in any given session by changing its *TimeoutMinutes* property.

In *TSessionsService*, the *MaxSessions* property specifies how many users can be logged into the system at any given time. The default value for *MaxSessions* is -1, which places no software limitation on the number of allowed users. Of course, your server hardware can still run short of memory or processor cycles for new users, which can adversely affect system performance. If you are concerned that excessive numbers of users might overwhelm your server, be sure to set *MaxSessions* to an appropriate value.

Login pages

Of course, your application also needs a login page. Users enter their username and password for authentication, either while trying to access a restricted page or prior to such an attempt. The user can also specify which page they receive when authentication is completed. If the username and password match a user in the Web user list, the user acquires the appropriate access rights and is forwarded to the page specified on the login page. If the user isn't authenticated, the login page may be redisplayed (the default action) or some other action may occur.

Fortunately, WebSnap makes it easy to create a simple login page using a Web page module and the adapter page producer. To create a login page, start by creating a new Web page module. Select File | New | Other to display the New Items dialog box, then select WebSnap Page Module from the WebSnap tab. Select AdapterPageProducer as the page producer type. Fill in the other options however you like. Login tends to be a good name for the login page.

Now you should add the most basic login page fields: a username field, a password field, a selection box for selecting which page the user receives after logging in, and a Login button which submits the page and authenticates the user. To add these fields:

1. Add a LoginFormAdapter component (which you can find on the WebSnap tab of the component palette) to the Web page module you just created.
2. Double-click the *AdapterPageProducer* component to display a Web page editor window.
3. Right-click the *AdapterPageProducer* in the top left pane and select New Component. In the Add Web Component dialog box, select *AdapterForm* and click OK.
4. Add an *AdapterFieldGroup* to the AdapterForm. (Right-click the *AdapterForm* in the top left pane and select New Component. In the Add Web Component dialog box, select *AdapterFieldGroup* and click OK.)
5. Now go to the Object Inspector and set the *Adapter* property of your *AdapterFieldGroup* to your *LoginFormAdapter*. The UserName, Password and NextPage fields should appear automatically in the Browser tab of the Web page editor.

So, WebSnap takes care of most of the work in a few simple steps. The login page is still missing a Login button, which submits the information on the form for authentication. To add a Login button:

1. Add an *AdapterCommandGroup* to the *AdapterForm*.

2 Add an *AdapterActionButton* to the *AdapterCommandGroup*.

3 Click on the *AdapterActionButton* (listed in the upper right pane of the Web page editor) and change its *ActionName* property to *Login* using the Object Inspector. You can see a preview of your login page in the Web page editor's Browser tab.

Your Web page editor should look similar to the one shown below.

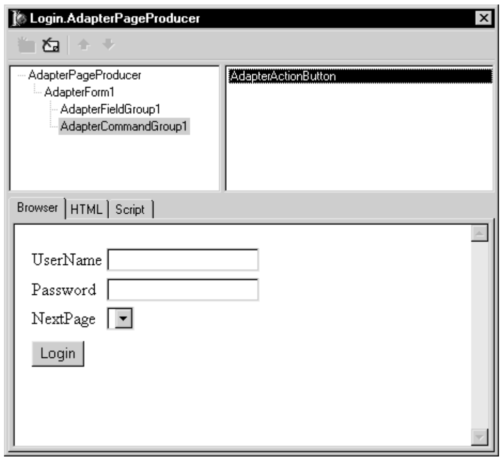


Figure 31.11 An example of a login page as seen from a Web page editor

If the button doesn't appear below the *AdapterFieldGroup*, make sure that the *AdapterCommandGroup* is listed below the *AdapterFieldGroup* on the Web page editor. If it appears above, select the *AdapterCommandGroup* and click the down arrow on the Web page editor. (In general, Web page elements appear vertically in the same order as they appear in the Web page editor.)

There is one more step necessary before your login page becomes functional. You need to specify which of your pages is the login page in your end user session adapter. To do so, select the *EndUserSessionAdapter* component in your Web application module. In the Object Inspector, change the *LoginPage* property to the name of your login page. Your login page is now enabled for all the pages in your Web server application.

Setting pages to require logins

Once you have a working login page, you must require logins for those pages which need controlled access. The easiest way to have a page require logins is to design that requirement into the page. When you first create a Web page module, check the Login Required box in the Page section of the New WebSnap Page Module dialog box.

If you create a page without requiring logins, you can change your mind later. To require logins after a Web page module has been created in Delphi:

D **1** Open the source code file associated with the Web page module in the editor.

2 Scroll down to the implementation section. In the parameters for the WebRequestHandler.AddWebModuleFactory command, find the creator of the TWebPageInfo object. It should look like this:

```
TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html')
```

3 Uncomment the *wpLoginRequired* portion of the parameter list by removing the curly braces. The TWebPageInfo creator should now look like this:

```
TWebPageInfo.Create([wpPublished , wpLoginRequired], '.html')
```

To require logins in C++:

1 Open the source code file associated with the Web page module in the editor.

2 Scroll down to the declaration of the static WebInit function.

3 Uncomment the *wpLoginRequired* portion of the parameter list by removing the /* and */ symbols surrounding it. The WebInit function should resemble the one shown below:

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished << wpLoginRequired, ".html", "", "", "", "");
```

To remove the login requirement from a page, reverse the process and recomment the *wpLoginRequired* portion of the creator (Delphi) or declaration (C++).

Note You can use the same processes to make the page published or not. Simply add or remove comment marks around the *wpPublished* portion as needed.

User access rights

User access rights are an important part of any Web server application. You need to be able to control who can view and modify the information your server provides. For example, let's say you are building a server application to handle online retail sales. It makes sense to allow users to view items in your catalog, but you don't want them to be able to change your prices! Clearly, access rights are an important issue.

Fortunately, WebSnap offers you several ways to control access to pages and server content. In previous sections, you saw how you can control page access by requiring logins. You have other options as well. For example:

- You can show data fields in an edit box to users with appropriate modify access rights; other users will see the field contents, but not have the ability to edit them.
- You can hide specific fields from users who don't have the correct view access rights.
- You can prevent unauthorized users from receiving specific pages.

Descriptions for implementing these behaviors are included in this section.

Dynamically displaying fields as edit or text boxes

If you use the adapter page producer, you can change the appearance of page elements for users with different access rights. For example, the Biolife demo (found in the WebSnap subdirectory of the Delphi demos directory) contains a form page which shows all the information for a given species. The form appears when the user clicks a Details button on the grid. A user logged in as Will sees data displayed as plain text. Will is not allowed to modify the data, so the form doesn't give him a mechanism to do so. User Ellen does have modify permissions, so when Ellen views the form page, she sees a series of edit boxes which allow her to change field contents. Using access rights in this manner can save you from creating extra pages.

The appearance of some page elements, such as *TAdapterDisplayField* and *TAdapterDisplayColumn*, is determined by its *ViewMode* property. If *ViewMode* is set to *vmToggleOnAccess*, the page element will appear as an edit box to users with modify access. Users without modify access will see plain text. Set the *ViewMode* property to *vmToggleOnAccess* to allow the page element's appearance and function to be determined dynamically.

A Web user list is a list of *TWebUserListItem* objects, one for each user who can login to the system. Permissions for users are stored in their Web user list item's *AccessRights* property. *AccessRights* is a text string, so you are free to specify permissions any way you like. Create a name for every kind of access right you want in your server application. If you want a user to have multiple access rights, separate items in the list with a space, semicolon or comma.

Access rights for fields are controlled by their *ViewAccess* and *ModifyAccess* properties. *ViewAccess* stores the name of the access rights needed to view a given field. *ModifyAccess* dictates what access rights are needed to modify field data. These properties appear in two places: in each field and in the adapter object that contains them.

Checking access rights is a two-step process. When deciding the appearance of a field in a page, the application first checks the field's own access rights. If the value is an empty string, the application then checks the access rights for the adapter which contains the field. If the adapter property is empty as well, the application will follow its default behavior. For modify access, the default behavior is to allow modifications by any user in the Web user list who has a non-empty *AccessRights* property. For view access, permission is automatically granted when no view access rights are specified.

Hiding fields and their contents

You can hide the contents of a field from users who don't have appropriate view permissions. First set the *ViewAccess* property for the field to match the permission you want users to have. Next, make sure that the *ViewAccess* for the field's page element is set to *vmToggleOnAccess*. The field caption will appear, but the value of the field won't.

Of course, it is often best to hide all references to the field when a user doesn't have view permissions. To do so, set the *HideOptions* for the field's page element to include *hoHideOnNoDisplayAccess*. Neither the caption nor the contents of the field will be displayed.

Preventing page access

You may decide that certain pages should not be accessible to unauthorized users. To grant check access rights before displaying pages, use the following steps.

In Delphi, alter your call to the *TWebPageInfo* constructor in the Web request handler's AddWebModuleFactory command. This command appears in the initialization section of the source code for your module.

The constructor for TWebPageInfo takes up to 6 arguments. WebSnap usually leaves four of them set to default values (empty strings), so the call generally looks like this:

```
TWebPageInfo.Create([wpPublished, wpLoginRequired], '.html')
```

To check permissions before granting access, you need to supply the string for the necessary permission in the sixth parameter. For example, let's say that the permission is called "Access". This is how you could modify the creator:

```
TWebPageInfo.Create([wpPublished, wpLoginRequired], '.html', '', '', '', 'Access')
```

In C++, alter your declaration of the module's WebInit function. This funciton appears in the source code for your module.

The WebInit function takes up to 9 arguments. WebSnap usually leaves four of them set to default values (empty strings), so the call generally looks like this:

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "");
```

To check permissions before granting access, you need to supply the string for the necessary permission in the ninth parameter. For example, let's say that the permission is called Access. This is how you could modify the WebInit function:

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "Access");
```

Access to the page will now be denied to anyone who lacks Access permission.

Server-side scripting in WebSnap

Page producer templates can include scripting languages such as JavaScript or VBScript. The page producer executes the script in response to a request for the producer's content. Because the Web server application evaluates the script, it is called server-side script, as opposed to client-side script (which is evaluated by the browser).

This section provides a conceptual overview of server-side scripting and how it is used by WebSnap applications. The "WebSnap server-side scripting reference"topic of the help files has much more detailed information about script objects and their properties and methods. You can think of it as an API reference for server-side scripting, similar to the object descriptions found in the IDE's help files. The server-side scripting refernece also contains detailed JavaScript examples which show you exactly how script can be used to generate HTML pages.

Although server-side scripting is a valuable part of WebSnap, it is not essential that you use scripting in your WebSnap applications. Scripting is used for HTML generation and nothing else. It allows you to insert application data into an HTML page. In fact, almost all of the properties exposed by adapters and other script-enabled objects are read-only. Server-side script isn't used to change application data, which is still managed by components and event handlers written in Delphi or C++.

There are other ways to insert application data into an HTML page. You can use Web Broker's transparent tags or some other tag-based solution, if you prefer. For example, several projects in the WebSnap demos directories use XML and XSL instead of scripting. Without scripting, however, you will be forced to write most of your HTML generation logic in Delphi or C++, which will increase your development time.

The scripting used in WebSnap is object-oriented and supports conditional logic and looping, which can greatly simplify your page generation tasks. For example, your pages may include a data field that can be edited by some users but not others. With scripting, conditional logic can be placed in your template pages which displays an edit box for authorized users and simple text for others. With a tag-based approach, you must program such decision-making into your HTML generating source code.

Script engine

The page producer's *ScriptEngine* property identifies the server-side scripting engine that evaluates the script within a template. Borland's Windows rapid application development tools use Microsoft's JScript as the default script engine. Linux products use the SpiderMonkey JavaScript engine (part of the Mozilla open-source browser project) as the default script engine. JScript and JavaScript are such similar languages that the script produced by WebSnap components is identical on both platforms, which makes WebSnap applications easy to port. In the documentation, the term JavaScript refers to either JScript or JavaScript.

Note WebSnap adapter page producer and adapter components are designed to produce JavaScript. Applications can support other scripting languages (such as VBScript in Windows), but you will need to provide your own script generation logic.

Script blocks

Script blocks, which appear in HTML templates, are delimited by <% and %>. The script engine evaluates any text inside script blocks. The result becomes part of the page producer's content. The page producer writes text outside of a script block after translating any embedded transparent tags. Script blocks can also enclose text, allowing conditional logic and loops to dictate the output of text. For example, the following JavaScript block generates a list of five numbered lines:

```
<ul>
<% for (i=0;i<5;i++) { %>
    <li>Item <%=i %></li>
<% } %>
</ul>
```

(The <%= ... %> tags are short for *Response.Write*.)

Creating script

Developers can take advantage of WebSnap features to automatically generate script.

Wizard templates

When creating a new WebSnap application or page module, WebSnap wizards provide a template field that is used to select the initial content for the page module template. For example, the Default template generates JavaScript which, in turn, displays the application title, page name, and links to published pages.

TAdapterPageProducer

The *TAdapterPageProducer* builds forms and tables by generating HTML and JavaScript. The generated JavaScript calls adapter objects to retrieve field values, field image parameters, and action parameters.

Editing and viewing script

Use the HTML Result tab to view the HTML resulting from the executed script. Use the Preview tab to view the result in a browser. The HTML Script tab is available when the Web Page module uses *TAdapterPageProducer*. The HTML Script tab displays the HTML and JavaScript generated by the *TAdapterPageProducer* object. Consult this view to see how to write script that builds HTML forms to display adapter fields and execute adapter actions.

Including script in a page

A template can include script from a file or from another page. To include script from a file, use the following code statement:

```
<!-- #include file="filename.html" -->
```

When the template includes script from another page, the script is evaluated by the including page. Use the following code statement to include the unevaluated content of page1.

```
<!-- #include page="page1" -- >
```

Script objects

Script objects are objects that script commands can reference. You make objects available for scripting by registering an *IDispatch* interface to the object with the active scripting engine. The following objects are available for scripting:

Table 31.4 Script objects

Script object	Description
Application	Provides access to the application adapter of the Web Application module.
EndUser	Provides access to the end user adapter of the Web Application module.
Session	Provides access to the session object of the Web Application module.
Pages	Provides access to the application pages.
Modules	Provides access to the application modules.

Table 31.4 Script objects (continued)

Script object	Description
Page	Provides access to the current page
Producer	Provides access to the page producer of the Web Page module.
Response	Provides access to the WebResponse. Use this object when tag replacement is not desired.
Request	Provides access to the WebRequest.
Adapter objects	All of the adapter components on the current page can be referenced without qualification. Adapters in other modules must be qualified using the Modules objects.

Script objects on the current page, which all use the same adapter, can be referenced without qualification. Script objects on other pages are part of another page module and have a different adapter object. They can be accessed by starting the script object reference with the name of the adapter object. For example,

```
<%= FirstName %>
```

displays the contents of the *FirstName* property of the current page's adapter. The following script line displays the *FirstName* property of Adapter1, which is in another page module:

```
<%= Adapter1.FirstName %>
```

For more complete descriptions of script objects, see the ”WebSnap server-side scripting reference”appendix.

Dispatching requests and responses

One reason to use WebSnap for your Web server application development is that WebSnap components automatically handle HTML requests and responses. Instead of writing event handlers for common page transfer chores, you can focus your efforts on your business logic and server design. Still, it can be helpful to understand how WebSnap applications handle HTML requests and responses. This section gives you an overview of that process.

Before handling any requests, the Web application module initializes the Web context object (of type *TWebContext*). The Web context object, which is accessed by calling the global WebContext function, provides global access to variables used by components servicing the request. For example, the Web context contains the *TWebRequest* and *TWebResponse* objects to represent the HTTP request message and the response that should be returned.

Dispatcher components

The dispatcher components in the Web application module control the flow of the application. The dispatchers determine how to handle certain types of HTTP request messages by examining the HTTP request.

The adapter dispatcher component (*TAdapterDispatcher)* looks for a content field, or a query field, that identifies an adapter action component or an adapter image field component. If the adapter dispatcher finds a component, it passes control to that component.

The Web dispatcher component (*TWebDispatcher*) maintains a collection of action items (of type *TWebActionItem*) that know how to handle certain types of HTTP request messages. The Web dispatcher looks for an action item that matches the request. If it finds one, it passes control to that action item. The Web dispatcher also looks for auto-dispatching components that can handle the request.

The page dispatcher component (*TPageDispatcher*) examines the *PathInfo* property of the *TWebRequest* object, looking for the name of a registered Web page module. If the dispatcher finds a Web page module name, it passes control to that module.

Adapter dispatcher operation

The adapter dispatcher component (*TAdapterDispatcher*) automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.

Using adapter components to generate content

For WebSnap applications to automatically execute adapter actions and retrieve dynamic images from adapter fields, the HTML content must be properly constructed. If the HTML content is not properly constructed, the resulting HTTP request will not contain the information that the adapter dispatcher needs to call adapter action and field components.

To reduce errors in constructing the HTML page, adapter components indicate the names and values of HTML elements. Adapter components have methods that retrieve the names and values of hidden fields that must appear on an HTML form designed to update adapter fields. Typically, page producers use server-side scripting to retrieve names and values from adapter components and then uses this information to generate HTML. For example, the following script constructs an <IMG> element that references the field called Graphic from Adapter1:

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>" alt="<%=Adapter1.Graphic.DisplayText%>">
```

When the Web application evaluates the script, the HTML src attribute will contain the information necessary to identify the field and any parameters that the field component needs to retrieve the image. The resulting HTML might look like this:

```
<img src="?_lSpecies No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

When the browser sends an HTTP request to retrieve this image to the Web application, the adapter dispatcher will be able to determine that the Graphic field of Adapter1, in the module DM, should be called with the parameter "Species No=90090". The adapter dispatcher will call the Graphic field to write an appropriate HTTP response.

The following script constructs an <A> element referencing the EditRow action of Adapter1 and creates a hyperlink to a page called Details:

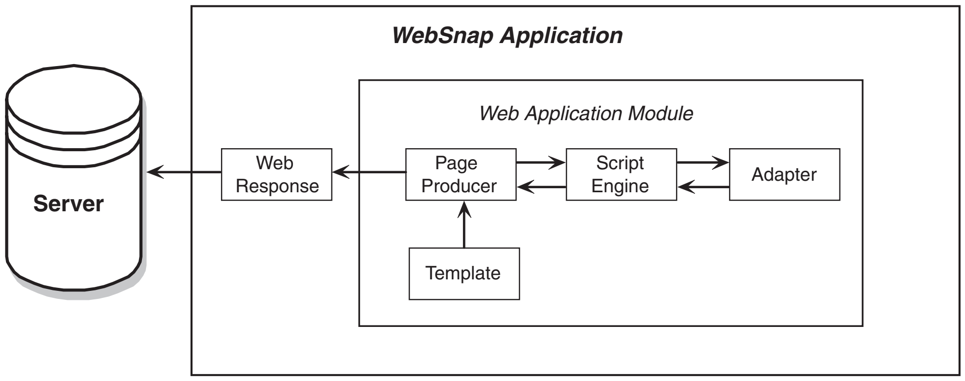
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

The resulting HTML might look like this:

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

The end user clicks this hyperlink, and the browser sends an HTTP request. The adapter dispatcher can determine that the EditRow action of Adapter1, in the module DM, should be called with the parameter Species No=903010. The adapter dispatcher also displays the Edit page if the action executes successfully, and displays the Grid page if action execution fails. It then calls the EditRow action to locate the row to be edited, and the page named Edit is called to generate an HTTP response. Figure 31.12 shows how adapter components are used to generate content.

Figure 31.12 Generating content flow



Receiving adapter requests and generating responses

When the adapter dispatcher receives a client request, the adapter dispatcher creates adapter request and adapter response objects to hold information about that HTTP request. The adapter request and adapter response objects are stored in the Web context to allow access during the processing of the request.

The adapter dispatcher creates two types of adapter request objects: action and image. It creates the action request object when executing an adapter action. It creates the image request object when retrieving an image from an adapter field.

The adapter response object is used by the adapter component to indicate the response to an adapter action or adapter image request. There are two types of adapter response objects, action and image.

Action requests

Action request objects are responsible for breaking the HTTP request down into information needed to execute an adapter action. The types of information needed for executing an adapter action may include the following request information:

Table 31.5 Request information found in action requests

Request informaton	Description
Component name	Identifies the adapter action component.
Adapter mode	Defines a mode. For example, *TDataSetAdapter* supports Edit, Insert, and Browse modes. An adapter action may execute differently depending on the mode.
Success page	Identifies the page displayed after successful execution of the action.
Failure page	Identifies the page displayed if an error occurs during execution of the action.
Action request parameters	Identifies the parameters need by the adapter action. For example, the *TDataSetAdapter* Apply action will include the key values identifying the record to be updated.
Adapter field values	Specifies values for the adapter fields passed in the HTTP request when an HTML form is submitted. A field value can include new values entered by the end user, the original values of the adapter field, and uploaded files.
Record keys	Specifies keys that uniquely identify each record.

Generating action responses

Action response objects generate an HTTP response on behalf of an adapter action component. The adapter action indicates the type of response by setting properties within the object, or by calling methods in the action response object. The properties include:

- *RedirectOptions*—The redirect options indicate whether to perform an HTTP redirect instead of returning HTML content.
- *ExecutionStatus*—Setting the status to success causes the default action response to be the content of the success page identified in the Action Request.

The action response methods include:

- *RespondWithPage* —The adapter action calls this method when a particular Web page module should generate the response.
- *RespondWithComponent*—The adapter action calls this method when the response should come from the Web page module containing this component.
- *RespondWithURL*—The adapter action calls this method when the response is a redirect to a specified URL.

When responding with a page, the action response object attempts to use the page dispatcher to generate page content. If it does not find the page dispatcher, it calls the Web page module directly.

Figure 31.15 illustrates how action request and action response objects handle a request.

Figure 31.13 Action request and response

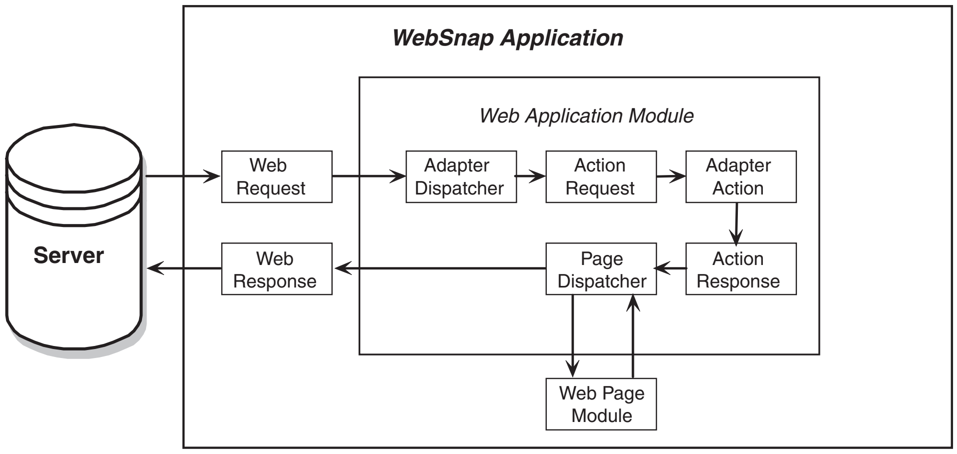


Image request

The image request object is responsible for breaking the HTTP request down into the information required by the adapter image field to generate an image. The types of information represented by the Image Request include:

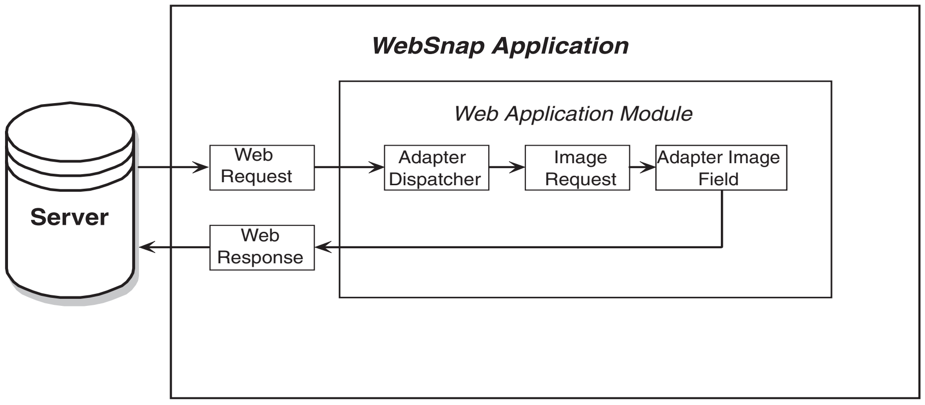
- Component name - Identifies the adapter field component.
- Image request parameters - Identifies the parameters needed by the adapter image. For example, the *TDataSetAdapterImageField* object needs key values to identify the record that contains the image.

Image response

The image response object contains the *TWebResponse* object. Adapter fields respond to an adapter request by writing an image to the Web response object.

Figure 31.14 illustrates how adapter image fields respond to a request.

Figure 31.14 Image response to a request



Dispatching action items

When responding to a request, the Web dispatcher (*TWebDispatcher*) searches through its list of action items for one that:

- matches the PathInfo portion of the target URL's request message, and
- can provide the service specified as the method of the request message.

It accomplishes this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds the appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response, or signals that the request has been completely handled.
- Adds to the response, and then allows other action items to complete the job.
- Defers the request to other action items.

After the dispatcher has checked all of its action items, if the message was not handled correctly, the dispatcher checks for specially registered auto-dispatching components that do not use action items. (These components are specific to multi-tiered database applications.) If the request message is still not fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

Page dispatcher operation

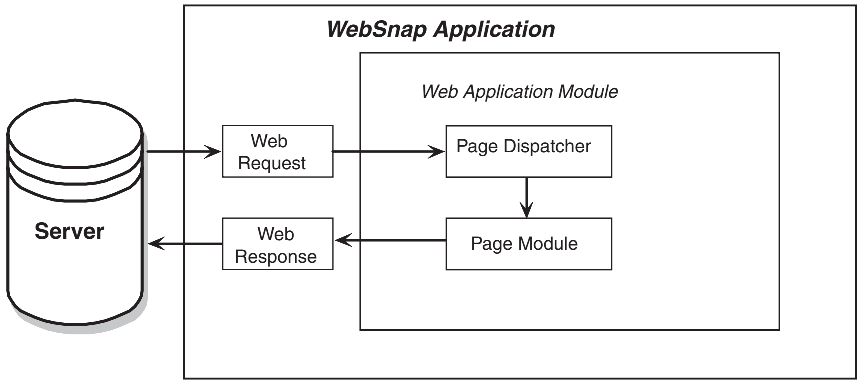
When the page dispatcher receives a client request, it determines the page name by checking the PathInfo portion of the target URL's request message. If the PathInfo portion is not blank, the page dispatcher uses the ending word of PathInfo as the page name. If the PathInfo portion is blank, the page dispatcher tries to determine a default page name.

If the page dispatcher's *DefaultPage* property contains a page name, the page dispatcher uses this name as the default page name. If the *DefaultPage* property is blank and the Web application module is a page module, the page dispatcher uses the name of the Web application module as the default page name.

If the page name is not blank, the page dispatcher searches for a Web page module with a matching name. If it finds a Web page module, it calls that module to generate a response. If the page name is blank, or if the page dispatcher does not find a Web page module, the page dispatcher raises an exception.

Figure 31.15 shows how the page dispatcher responds to a request.

Figure 31.15 Dispatching a page



Chapter

32

Working with XML documents

XML (Extensible Markup Language) is a markup language for describing structured data. It is similar to HTML, except that the tags describe the structure of information rather than its display characteristics. XML documents provide a simple, text-based way to store information so that it is easily searched or edited. They are often used as a standard, transportable format for data in Web applications, business-to-business communication, and so on.

XML documents provide a hierarchical view of a body of data. Tags in the XML document describe the role or meaning of each data element, as illustrated in the following document, which describes a collection of stock holdings:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

This example illustrates a number of typical elements in an XML document. The first line is a processing instruction called an XML declaration. The XML declaration is optional but you should include it because it supplies useful information about the document. In this example, the XML declaration says that the document conforms to version 1.0 of the XML specification, that it uses UTF-8 character encoding, and that it relies on an external file for its document type declaration (DTD).

The second line, which begins with the <!DOCType> tag, is a document type declaration (DTD). The DTD is how XML defines the structure of the document. It imposes syntax rules on the elements (tags) contained in the document. The DTD in this example references another file (sth.dtd). In this case, the structure is defined in an external file, rather than within the XML document itself. Other types of files that describe the structure of an XML document include Reduced XML Data (XDR) and XML schemas (XSD).

The remaining lines are organized into a hierarchy with a single root node (the <StockHoldings> tag). Each node in this hierarchy contains either a set of child nodes, or a text value. Some of the tags (the <Stock> and <shares> tags) include attributes, which are Name=Value pairs that provide details on how to interpret the tag.

Although it is possible to work directly with the text in an XML document, typically applications use additional tools for parsing and editing the data. W3C defines a set of standard interfaces for representing a parsed XML document called the Document Object Model (DOM). A number of vendors provide XML parsers that implement the DOM interfaces to let you interpret and edit XML documents more easily.

Special wizards and a number of CLX components and interfaces make it even easier to work with XML documents. These wizards and components that use a DOM parser that is provided by another vendor. This chapter describes those wizards and classes.

Note In addition to the tools described in this chapter, additional classes and tools let you convert XML documents to data packets that integrate into the CLX database architecture. For details on tools for integrating XML documents into database applications, see Chapter 28, “Using XML in database applications.”

Using the Document Object Model

The Document Object Model (DOM) is a set of standard interfaces for representing a parsed XML document. These interfaces are implemented by a number of different third-party vendors. If you do not want to use the default vendor that ships with CLX, there is a registration mechanism that lets you integrate additional DOM implementations by other vendors into the XML framework.

The XMLDOM unit includes declarations for all the DOM interfaces defined in the W3C XML DOM level 2 specification. Each DOM vendor provides an implementation for these interfaces.

- To use one of the DOM vendors supported by CLX, locate the unit that represents the DOM implementation. These units end in the ‘xmldom’. For example, the unit for the IMB implementation is IBMXMLDOM and the unit for the Open XML implementation is OXMLDOM. If you add the unit for the desired implementation to your project, the DOM implementation is automatically registered so that it is available to your code.
- To use another DOM implementation, you must create a unit that defines a descendant of the *TDOMVendor* class. This unit can then work like one of the

built-in DOM implementations, making your DOM implementation available when it is included in a project.

- In your descendant class, you must override two methods: the *Description* method, which returns a string identifying the vendor, and the *DOMImplementation* method, which returns the top-level interface (*IDOMImplementation*).
- Your new unit must register the vendor by calling the global *RegisterDOMVendor* procedure. In Delphi, this call typically goes in the initialization section of the unit. In C++, use the pragma startup directive.
- When your unit is unloaded, it needs to unregister itself to indicate that the DOM implementation is no longer available. Unregister the vendor by calling the global *UnRegisterDOMVendor* procedure. In Delphi, this call typically goes in the finalization section. In C++, use the pragma exit directive.

You can work directly with the DOM interfaces to parse and edit XML documents. Simply call the *GetDOM* function to obtain an *IDOMImplementation* interface, which you can use as a starting point.

Note For detailed descriptions of the DOM interfaces, see the declarations in the XMLDOM unit, the documentation supplied by your DOM Vendor, or the specifications provided on the W3C web site (www.w3.org).

You may find it more convenient to use special XML classes rather than working directly with the DOM interfaces. These are described below.

Working with XML components

CLX defines a number of classes and interfaces for working with XML documents. These simplify the process of loading, editing, and saving XML documents.

Using TXMLDocument

The starting point for working with an XML document is the *TXMLDocument* component. The following steps describe how to use *TXMLDocument* to work directly with an XML document:

1. Add a *TXMLDocument* component to your form or data module. *TXMLDocument* appears on the Internet page of the Component palette.
2. Set the *DOMVendor* property to specify the DOM implementation you want the component to use for parsing and editing an XML document. The Object Inspector lists all the currently registered DOM vendors. For information on DOM implementations, see "Using the Document Object Model" on page 32-2.
3. If you are working with an existing XML document, specify the document:
   - If the XML document is stored in a file, set the *FileName* property to the name of that file.

- You can specify the XML document as a string instead by using the *XML* property.

4 Set the *Active* property to true.

Once you have an active *TXMLDocument* object, you can traverse the hierarchy of its nodes, reading or setting their values. The root node of this hierarchy is available as the *DocumentElement* property.

Working with XML nodes

Once an XML document has been parsed by a DOM implementation, the data it represents is available as a hierarchy of nodes. Each node corresponds to a tagged element in the document. For example, given the following XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

TXMLDocument would generate a hierarchy of nodes as follows: The root of the hierarchy would be the *StockHoldings* node. *StockHoldings* would have two child nodes, which correspond to the two *Stock* tags. Each of these two child nodes would have four child nodes of its own (*name*, *price*, *symbol*, and *shares*). Those four child nodes would act as leaf nodes. The text they contain would appear as the value of each of the leaf nodes.

Note This division into nodes differs slightly from the way a DOM implementation generates nodes for an XML document. In particular, a DOM parser treats all tagged elements as internal nodes. Additional nodes (of type text node) are created for the values of the *name*, *price*, *symbol*, and *shares* nodes. These text nodes then appear as the children of the *name*, *price*, *symbol*, and *shares* nodes.

Each node in the hierarchy is accessed through an *IXMLNode* interface, starting with the root node, which is the value of the XML document component's *DocumentElement* property.

Working with a node's value

Given an *IXMLNode* interface, you can check whether it represents an internal node or a leaf node by checking the *IsTextElement* property.

- If it represents a leaf node, you can read or set its value using the *Text* property.
- If it represents an internal node, you can access its child nodes using the *ChildNodes* property.

Thus, for example, using the XML document above, you can read the price of Borland's stock as follows:

```
BorlandStock := XMLDocument1.DocumentElement.ChildNodes[0]; {Delphi}
Price := BorlandStock.ChildNodes['price'].Text;
```

```
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes->GetNode(0); // C++
AnsiString Price = BorlandStock->ChildNodes->Nodes[WideString("price")]->Text;
```

Working with a node's attributes

If the node includes any attributes, you can work with them using the *Attributes* property. You can read or change an attribute value by specifying an existing attribute name. You can add new attributes by specifying a new attribute name when you set the *Attributes* property:

```
BorlandStock := XMLDocument1.DocumentElement.ChildNodes[0]; { Delphi }
BorlandStock.ChildNodes['shares'].Attributes['type'] := 'common';
```

```
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes->GetNode(0); // C++
BorlandStock->ChildNodes->Nodes[WideString("shares")]->Attributes[WideString("type")] =
"common";
```

Adding and deleting child nodes

You can add child nodes using the *AddChild* method. *AddChild* creates new nodes that correspond to tagged elements in the XML document. Such nodes are called element nodes.

To create a new element node, specify the name that appears in the new tag and, optionally, the position where the new node should appear. For example, the following code adds a new stock listing to the document above:

Delphi example

```
var
  NewStock: IXMLNode;
  ValueNode: IXMLNode;
begin
  NewStock := XMLDocument1.DocumentElement.AddChild('stock');
  NewStock.Attributes['exchange'] := 'NASDAQ';
  ValueNode := NewStock.AddChild('name');
  ValueNode.Text := 'Cisco Systems'
  ValueNode := NewStock.AddChild('price');
  ValueNode.Text := '62.375';
  ValueNode := NewStock.AddChild('symbol');
  ValueNode.Text := 'CSCO';
  ValueNode := NewStock.AddChild('shares');
  ValueNode.Text := '25';
end;
```

C++ example

```
_di_IXMLNode NewStock = XMLDocument1->DocumentElement->AddChild(WideString("stock"));
NewStock->Attributes[WideString("exchange")] = "NASDAQ";
_di_IXMLNode ValueNode = NewStock->AddChild(WideString("name"));
ValueNode->Text = WideString("Cisco Systems");
ValueNode = NewStock->AddChild(WideString("price"));
ValueNode->Text = WideString("62.375");
ValueNode = NewStock->AddChild(WideString("symbol"));
ValueNode->Text = WideString("CSCO");
ValueNode = NewStock->AddChild(WideString("shares"));
ValueNode->Text = WideString("25");
```

An overloaded version of *AddChild* lets you specify the namespace URI in which the tag name is defined.

You can delete child nodes using the methods of the *ChildNodes* property. *ChildNodes* is an *IXMLNodeList* interface, which manages the children of a node. You can use its *Delete* method to delete a single child node that is identified by position or by name. For example, the following code deletes the last stock listed in the document above:

```
StockList := XMLDocument1.DocumentElement; { Delphi }
StockList.ChildNodes.Delete(StockList.ChildNodes.Count - 1);
```

```
_di_IXMLNode StockList = XMLDocument1->DocumentElement; // C++
StockList->ChildNodes->Delete(StockList->ChildNodes->Count - 1);
```

Abstracting XML documents with the Data Binding wizard

It is possible to work with an XML document using only the *TXMLDocument* component and the *IXMLNode* interface it surfaces for the nodes in that document, or even to work exclusively with the DOM interfaces (avoiding even *TXMLDocument*). However, you can write code that is much simpler and more readable by using the XML Data Binding wizard.

The Data Binding wizard takes an XML schema or data file and generates a set of interfaces that map on top of it.

Note In C++, an interface is a class with no data members and only pure virtual members. For information about interfaces in C++, see "Inheritance and interfaces" on page 14-3.

For example, given XML data that looks like the following:

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

The Data Binding wizard generates the following interface (along with a class to implement it):

Delphi example

```
ICustomer = interface(IXMLNode)
```

```
  property id: Integer read Getid write Setid;
  property name: DOMString read Getname write Setname;
  property phone: DOMString read Getphone write Setphone;
  function Getid: Integer;
  function Getname: DOMString;
  function Getphone: DOMString;
  procedure Setid(Value: Integer);
  procedure Setname(Value: DOMString);
  procedure Setphone(Value: DOMString);
end;
```

C++ example

```
__interface INTERFACE_UUID("{F3729105-3DD0-1234-80e0-22A04FE7B451}") ICustomer :
    public IXMLNode
{
public:
  virtual int __fastcall Getid(void) = 0 ;
  virtual DOMString __fastcall Getname(void) = 0 ;
  virtual DOMString __fastcall Getphone(void) = 0 ;
  virtual void __fastcall Setid(int Value)= 0 ;
  virtual void __fastcall Setname(DOMString Value)= 0 ;
  virtual void __fastcall Setphone(DOMString Value)= 0 ;
  __property int id = {read=Getid, write=Setid};
  __property DOMString name = {read=Getname, write=Setname};
  __property DOMString phone = {read=Getphone, write=Setphone};
};
```

Every child node is mapped to a property whose name matches the tag name of the child node and whose value is the interface of the child node (if the child is an internal node) or the value of the child node (for leaf nodes). Every node attribute is also mapped to a property, where the property name is the attribute name and the property value is the attribute value.

In addition to creating interfaces (and implementation classes) for each tagged element in the XML document, the wizard creates global functions for obtaining the interface to the root node. For example, if the XML above came from a document whose root node had the tag <Customers>, the Data Binding wizard would create the following global routines:

Delphi example

```
function GetCustomers(XMLDoc: IXMLDocument): ICustomers;

function LoadCustomers(const FileName: WideString): ICustomers;

function NewCustomers: ICustomers;
```

C++ example

```
extern PACKAGE _di_ICustomers __fastcall GetCustomers(TXMLDocument *XMLDoc);

extern PACKAGE _di_ICustomers __fastcall GetCustomers(_di_IXMLDocument XMLDoc);

extern PACKAGE _di_ICustomers __fastcall LoadCustomers(const WideString FileName);
extern PACKAGE _di_ICustomers __fastcall NewCustomers(void);
```

The Get... function takes the interface (Delphi) or interface wrapper (C++) for a *TXMLDocument* instance (or, in C++, a pointer to the *TXMLDocument* instance). The Load... function dynamically creates a *TXMLDocument* instance and loads the specified XML file as its value before returning an interface pointer. The New... function creates a new (empty) *TXMLDocument* instance and returns the interface to the root node.

Using the generated interfaces simplifies your code, because they reflect the structure of the XML document more directly. For example, instead of writing code such as the following:

```
CustIntf := XMLDocument1.DocumentElement;
CustName := CustIntf.ChildNodes[0].ChildNodes['name'].Value;
```

```
_di_IXMLNode CustIntf = XMLDocument1->DocumentElement;
CustName = CustIntf->ChildNodes->Nodes->GetNode(0)->ChildNodes->Nodes[WideString("name")]-
>Value;
```

Your code would look as follows:

```
CustIntf := GetCustomers(XMLDocument1);
CustName := CustIntf[0].Name;
```

```
_di_ICustomers CustIntf = GetCustomers(XMLDocument1);
CustName = CustIntf->Nodes->GetNode(0)->Name;
```

Note that the interfaces generated by the Data Binding wizard all descend from *IXMLNode*. This means you can still add and delete child nodes in the same way as when you do not use the Data Binding wizard. (See "Adding and deleting child nodes" on page 32-5.) In addition, when child nodes represent repeating elements (when all of the children of a node are of the same type), the parent node is given two methods, *Add*, and *Insert*, for adding additional repeats. These methods are simpler than using *AddChild*, because you do not need to specify the type of node to create.

Using the XML Data Binding wizard

To use the Data Binding wizard,

1 Choose File | New | Other and select the icon labeled XML Data Binding from the bottom of the New page.

2 The XML Data Binding wizard appears.

3 On the first page of the wizard, specify the XML document or schema for which you want to generate interfaces. This can be a sample XML document, a Document Type Definition (.dtd) file, a Reduced XML Data (.xdr) file, or an XML schema (.xsd) file.

4 Click the Options button to specify the naming strategies you want the wizard to use when generating interfaces and implementation classes and the default mapping of types defined in the schema to native data types.

5 Move to the second page of the wizard. This page lets you provide detailed information about every node type in the document or schema. At the left is a tree

view that shows all of the node types in the document. For complex nodes (nodes that have children), the tree view can be expanded to display the child elements. When you select a node in this tree view, the right-hand side of the dialog displays information about that node and lets you specify how you want the wizard to treat that node.

- The Source Name control displays the name of the node type in the XML schema.
- The Source Datatype control displays the type of the node's value, as specified in the XML schema.
- The Documentation control lets you add comments to the schema describing the use or purpose of the node.
- If the wizard generates code for the selected node (that is, if it is a complex type for which the wizard generates an interface and implementation class, or if it is one of the child elements of a complex type for which the wizard generates a property on the complex type's interface), you can use the Generate Binding check box to specify whether you want the wizard to generate code for the node. If you uncheck Generate Binding, the wizard does not generate the interface or implementation class for a complex type, or does not create a property in the parent interface for a child element or attribute.
- The Binding Options section lets you influence the code that the wizard generates for the selected element. For any node, you can specify the Identifier Name (the name of the generated interface or property). In addition, for interfaces, you must indicate which one represents the root node of the document. For nodes that represent properties, you can specify the type of the property and, if the property is not an interface, whether it is a read-only property.

6 Once you have specified what code you want the wizard to generate for each node, move to the third page. This page lets you choose some global options about how the wizard generates its code and lets you preview the code that will be generated, and lets you tell the wizard how to save your choices for future use.

- To preview the code the wizard generates, select an interface in the Binding Summary list and view the resulting interface definition in the Code Preview control.
- Use the Data Binding Settings to indicate how the wizard should save your choices. You can store the settings as annotations in a schema file that is associated with the document (the schema file specified on the first page of the dialog), or you can name an independent schema file that is used only by the wizard.

7 When you click Finish, the Data Binding wizard generates a new unit that defines interfaces and implementation classes for all of the node types in your XML document. In addition, it creates a global function that takes a *TXMLDocument* object and returns the interface for the root node of the data hierarchy.

Using code that the XML Data Binding wizard generates

Once the wizard has generated a set of interfaces and implementation classes, you can use them to work with XML documents that match the structure of the document or schema you supplied to the wizard. Just as when you are using only the built-in XML components, your starting point is the *TXMLDocument* component that appears on the Internet page of the Component palette.

To work with an XML document, use the following steps:

1 Obtain an interface for the root node of your XML document. You can do this in one of three ways:

- Place a *TXMLDocument* component in your form or data module. Bind the *TXMLDocument* to an XML document by setting the *FileName* property. (As an alternative approach, you can use a string of XML by setting the *XML* property at runtime.) Then, In your code, call the global function that the wizard created to obtain an interface for the root node of the XML document. For example, if the root element of the XML document was the tag <StockList>, by default, the wizard generates a function *GetStockListType*, which returns an *IStockListType* interface:

D
```
  var
StockList: IStockListType;
begin
  XMLDocument1.FileName := 'Stocks.xml';
  StockList := GetStockListType(XMLDocument1);
```

C++
```
    XMLDocument1->FileName := "Stocks.xml";
_di_IStockListType StockList = GetStockListType(XMLDocument1);
```

- Call the generated Load... function to create and bind the *TXMLDocument* instance and obtain its interface all in one step. For example, using the same XML document described above:

D
```
  var
StockList: IStockListType;
begin
  StockList := LoadStockListType('Stocks.xml');
```

C++
```
  _di_IStockListType StockList = LoadStockListType("Stocks.xml");
```

- Call the generated New... function to create the *TXMLDocument* instance for an empty document when you want to create all the data in your application:

D
```
    var
  StockList: IStockListType;
begin
  StockList := NewStockListType;
```

C++
```
    _di_IStockListType StockList = NewStockListType();
```

2 This interface has properties that correspond to the subnodes of the document's root element, as well as properties that correspond to that root element's attributes. You can use these to traverse the hierarchy of the XML document, modify the data in the document, and so on.

3 To save any changes you make using the interfaces generated by the wizard, call the *TXMLDocument* component's *SaveToFile* method or read its *XML* property.

Tip If you set the *Options* property of the *TXMLDocument* object to include *doAutoSave*, then you do not need to explicitly call the *SaveToFile* method.

Chapter

33

Using Web Services

Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications.

Web Services are designed to allow a loose coupling between client and server. That is, server implementations do not require clients to use a specific platform or programming language. In addition to defining interfaces in a language-neutral fashion, they are designed to allow multiple communications mechanisms as well.

Support for Web Services is designed to work using SOAP (Simple Object Access Protocol). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. It uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol. For more information about SOAP, see the SOAP specification available at

```
http://www.w3.org/TR/SOAP/
```

Note Although the components that support Web Services are built to use SOAP and HTTP, the framework is sufficiently general that it can be expanded to use other encoding and communications protocols.

In addition to letting you build SOAP-based Web Service applications (servers), special components and wizards let you build clients of Web Services that use either a SOAP encoding or a Document Literal style. The Document Literal style is used in .Net Web Services.

The components that support Web Services are available on both Windows and Linux, so you can use them as the basis of cross-platform distributed applications. There is no special client runtime software to install, as you must have when distributing applications using CORBA. Because this technology is based on HTTP messages, it has the advantage that it is widely available on a variety of machines. Support for Web Services is built on the Web server application architecture (Web Broker).

Web Service applications publish information on what interfaces are available and how to call them using a WSDL (Web Service Definition Language) document. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard or command-line utility can import a published WSDL document, providing you with the interface definitions and connection information you need. If you already have a WSDL document that describes the Web service you want to implement, you can generate the server-side code as well when importing the WSDL document.

Understanding invokable interfaces

Servers that support Web Services are built using invokable interfaces. Invokable interfaces are interfaces that are compiled to include runtime type information (RTTI). On the server, this RTTI is used when interpreting incoming method calls from clients so that they can be correctly marshaled. On clients, this RTTI is used to dynamically generate a method table for making calls to the methods of the interface.

Note In C++, an interface is a pure virtual class (a class that includes only pure virtual methods). For information about interface classes in C++, see "Inheritance and interfaces" on page 14-3.

To create an invokable interface in Delphi, you need only compile an interface with the {$M+} compiler option. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, your Web Service can only use the methods defined in the invokable interface and its descendants. Methods inherited from the noninvokeable ancestors are not compiled with type information and so can't be used as part of the Web Service.

To create an invokable interface in C++, you need to declare the interface class using the _declspec keyword, using the *delphirtti* modifier. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface class that is not invokable, your Web Service can only use the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be used as part of the Web Service.

When defining a Web service, you can derive an invokable interface from the base invokable interface, *IInvokable*. *IInvokable* is defined in the System unit in Delphi, or in the sysmac.h header file in C++. exposed to clients by a Web Service server. *IInvokable* is the same as the base interface (*IInterface*), except that it is compiled using the {$M+} compiler option in Delphi or the _declspec(delphirtti) option in C++. The {$M+} compiler option or _declspec(delphirtti) option ensures that the interface and all its descendants include RTTI.

For example, the following code defines an invokable interface that contains two methods for encoding and decoding numeric values:

Delphi example

```
IEncodeDecode = interface(IInvokable)
```

```
  ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;
```

C++ example

```
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}") IEncodeDecode :
    public IInvokable
{
public:
  virtual double __stdcall EncodeValue(int Value) = 0 ;
  virtual int __stdcall DecodeValue(double Value) = 0 ;
};
```

Note In the C++ example, notice the use of __interface in the declaration. This is not a true keyword, but rather a macro that is used by convention for interfaces. It maps to the class keyword. The INTERFACE_UUID macro assigns a globally unique identifier (GUID) to the interface. Note that the interface class contains only pure virtual methods.

Before a Web Service application can use this invokable interface, it must be registered with the invocation registry. On the server, the invocation registry entry allows the invoker component (*THTTPSOAPPascalInvoker* in Delphi or *THTTPSOAPCppInvoker* in C++) to identify an implementation class to use for executing interface calls. On client applications, an invocation registry entry allows remote interfaced objects (*THTTPRio*) to look up information that identifies the invokable interface and supplies information on how to call it.

Typically, your Web Service client or server creates the code to define invokable interfaces either by importing a WSDL document or using the Web Service wizard. By default, when the WSDL importer or Web Service wizard generates an interface, the definition is added to a unit with the same name as the Web Service. This unit includes both the interface definition and code to register the interface with the invocation registry. The invocation registry is a catalog of all registered invokable interfaces, their implementation classes, and any functions that create instances of the implementation classes. It is accessed using the global *InvRegistry* function, which is defined in the InvokeRegistry unit.

D In Delphi, the definition of the invokable interface is added to the interface section of the unit, and the code to register the interface goes in the initialization section. The registration code looks like the following:

```
initialization
  InvRegistry.RegisterInterface(TypeInfo(IEncodeDecode));
end.
```

Note The implementation section's uses clause must include the InvokeRegistry unit so that the call to the *InvRegistry* function is defined.

In C++, the interface definition is added to the header file and the corresponding .cpp file contains the code to register the interface (as well as an implementation class if you are writing a server). For the interface described above, the registration code looks like the following:

```
static void RegTypes()
```

```
{
  InvRegistry()->RegisterInterface(__delphirtti(IEncodeDecode), "", "");
}

#pragma startup RegTypes 32
```

Note The .cpp file must have an #include statement for the InvokeRegistry unit so that the call to the *InvRegistry* function is defined.

The interfaces of Web Services must have a namespace to identify them among all the interfaces in all possible Web Services. The previous example does not supply a namespace for the interface. When you do not explicitly supply a namespace, the invocation registry automatically generates one for you. This namespace is built from a string that uniquely identifies the application (the *AppNamespacePrefix* variable), the interface name, and the name of the unit in which it is defined. If you do not want to use the automatically-generated namespace, you can specify one explicitly using a second parameter to the *RegisterInterface* call.

In Delphi, you can use the same unit file to define an invokable interface for both client and server applications. If you are doing this, it is a good idea to keep the unit that defines your invokable interfaces separate from the unit in which you write the classes that implement them. Because the generated namespace includes the name of the unit in which the interface is defined, sharing the same unit in both client and server applications enables them to automatically use the same namespace, as long as they both use the same value for the *AppNamespacePrefix* variable.

For C++, do not be tempted to use the same header file that defines an invokable interface in both client and server applications. This can easily lead to mismatches in the namespaces, because when the header is included in another source file, the name of the unit changes to the including source file, changing the name of the generated namespace. Therefore, client applications should import the Web Service using a WSDL document.

Using nonscalar types in invokable interfaces

The Web Services architecture automatically includes support for marshaling the scalar types listed in Table 33.1:

Table 33.1 Scalar types automatically marshaled in Web Services applications

Delphi type	C++ type
Boolean	bool
Char	char
Byte	unsigned char
ShortInt	signed char
SmallInt	short
Word	unsigned short
Integer	int
Cardinal	unsigned int
LongInt	long

Table 33.1 Scalar types automatically marshaled in Web Services applications

Delphi type	C++ type
	unsigned long
Int64	__int64
	unsigned __int64
Single	float
Double	double
Extended	long double
string	AnsiString
WideString	WideString
Currency	Currency
Variant	Variant

Note The last four C++ types are not actually scalar types, but C++ classes that correspond to Delphi scalar types. However, they take advantage of the automatic marshaling provided for the Delphi scalar type.

You need do nothing special when you use these scalar types on an invokable interface. If your interface includes any properties or methods that use other types, however, your application must register those types with the remotable type registry. For more information on the remotable type registry, see "Registering nonscalar types" on page 33-6.

Dynamic arrays can be used in invokable interfaces, as long as you register them with the remotable type registry. The remotable type registry extracts all the information it needs from the type information that the compiler generates. In C++, you should try to avoid defining multiple dynamic array types with the same element type. Because the C++ compiler treats these as transparent types that can be implicitly cast one to another, it doesn't distinguish their runtime type information. As a result, the remotable type registry can't distinguish the types. This is not a problem for servers, but can result in clients using the wrong type definition.

Note The dynamic array types defined in the Types unit (Delphi) or sysdyn.h (C++) are automatically registered for you, so your application does not need to add any special registration code for them. One of these in particular, *TByteDynArray*, deserves special notice because it maps to a 'base64' block of binary data, rather than mapping each array element separately the way the other dynamic array types do.

Enumerated types and types that map directly to one of the automatically-marshaled scalar types can also be used in an invokable interface. As with dynamic array types, they must be registered so that the remotable type registry can generate the appropriate logic for marshaling them. In Delphi, the only extra step required is registration. In C++, the compiler does not automatically generate all the necessary information, and you must first create a holder class for the type. The use of holder classes is described in "Using holder classes to register C++ types" on page 33-7.

For any other types, such as static arrays, structs or records, sets, interfaces, or classes, you must map the type to a remotable class. A remotable class is a class that includes runtime type information (RTTI). Your interface must then use the remotable class instead of the corresponding static array, struct or record, set, interface, or class. Any remotable classes you create must be registered with the remotable type registry.

Important In C++, all types should be declared explicitly with a typedef statement rather than declared inline. This is necessary so that the remotable type registry can determine the native C++ type name.

Registering nonscalar types

Before an invokable interface can use any types other than the built-in scalar types listed in "Using nonscalar types in invokable interfaces" on page 33-4, the application must register the type with the remotable type registry. To access the remotable type registry, you must add the InvokeRegistry unit to your uses clause (Delphi) or include InvokeRegistry.hpp in your source file (C++). This unit or header declares a global function, *RemTypeRegistry*(), which returns a reference to the remotable type registry.

Note On clients, the code to register types with the remotable type registry is generated automatically when you import a WSDL document. For servers, remotable types are registered for you automatically when you register an interface that uses them. You only need to explicitly add code to register types if you want to specify the namespace or type name rather than using the automatically-generated values.

The remotable type registry has two methods that you can use to register types: *RegisterXSInfo* and *RegisterXSClass*. The first (*RegisterXSInfo*) lets you register a dynamic array or other type definition. The second (*RegisterXSClass*) is for registering remotable classes that you define to represent other types.

If you are using dynamic arrays or (in Delphi) enumerated types, the invocation registry can get the information it needs from the compiler-generated type information. Thus, for example, your interface may use a type such as the following:

```
type
  TDateTimeArray = array of TXSDateTime; {Delphi}
```

```
typedef DynamicArray<TXSDateTime> TDateTimeArray; // C++
```

This type is registered automatically when you register the invokable interface. However, if you want to specify the namespace in which the type is defined or the name of the type, you must add code to explicitly register the type using the *RegisterXSInfo* method of the remotable type registry.

In Delphi, the registration goes in the initialization section of the unit where you declare or use the dynamic array:

```
RemTypeRegistry.RegisterXSInfo(TypeInfo(TDateTimeArray), MyNameSpace, 'DTarray', 'DTarray');
```

In C++, the registration is added to the RegTypes function where you register the invokable interface that uses the dynamic array:

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(__arraytypeinfo(TDateTimeArray),
        MyNameSpace, "DTarray", "DTarray");
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

The first parameter of *RegisterXSInfo* is the type information for the type you are registering. The second parameter is the namespace URI for the namespace in which the type is defined. If you omit this parameter or supply an empty string, the registry generates a namespace for you. The third parameter is the name of the type as it appears in native code. In C++, you must supply a value for this parameter. In Delphi, if you omit this parameter or supply an empty string, the registry uses the type name from the type information you supplied as the first parameter. The final parameter is the name of the type as it appears in WSDL documents. If you omit this parameter or supply an empty string, the registry uses the native type name (the third parameter).

Registering a remotable class is similar, except that you supply a class reference rather than a type information pointer. For example, the following line comes from the XSBuiltIns unit. It registers *TXSDateTime*, a *TRemotable* descendant that represents *TDateTime* values:

```
RemClassRegistry.RegisterXSClass(TXSDateTime, XMLSchemaNameSpace, 'dateTime', '',True);
```

In C++, the equivalent registration would look something like the following

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TXSDateTime), XMLSchemaNameSpace, "dateTime",
"", true);
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

The first parameter is class reference for the remotable class that represents the type. The second is a uniform resource identifier (URI) that uniquely identifies the namespace of the new class. If you supply an empty string, the registry generates a URI for you. The third and fourth parameters specify the native and external names of the data type your class represents. If you omit the fourth parameter, the type registry uses the third parameter for both values. If you supply an empty string for both parameters, the registry uses the class name. The fifth parameter indicates whether the value of class instances can be transmitted as a string. You can optionally add a sixth parameter (not shown here) to control how multiple references to the same object instance should be represented in SOAP packets.

Using holder classes to register C++ types

In C++, the __delphirtti function can’t extract type information directly from enumerated types. To work around this, you need to generate a CLX-style holder class that can be used to extract the type information:

```
class MyEnumType_TypeInfoHolder : public TObject {
  MyEnumType __instanceType;
public:
__published:
  __property MyEnumType __propType = {read=__instanceType };
};
```

In this case, the class *MyEnumType_TypeInfoHolder* is defined to extract the type information from an enumerated type called *MyEnumType*. It has a single published property, *__propType*, whose type is the enumerated type you want to register. Once you have defined the holder class, you can obtain the type information for the enumerated type by calling the global *GetClsMemberTypeInfo* function. The following code illustrates how to register an enumerated type, given its holder class:

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(
          GetClsMemberTypeInfo(__classid(MyEnumType_TypeInfoHolder), "__propType"),
          MyNameSpace, "MyEnumType");
}
```

RegisterClsMemberTypeInfo has two parameters: The type information of the holder class, and the name of the published property from whose type you want to extract type information. If the holder class has only one published property (as in the previous example), you can omit the second parameter.

This technique of using a holder class must also be used if your type is declared using a typedef statement that maps it to a built-in C++ scalar type. (A built-in C++ scalar type is any of the types listed in "Using nonscalar types in invokable interfaces" on page 33-4 except for the ones that are classes that emulate Delphi types.) Thus, for example, given the following type:

```
typedef int CardNumber;
```

You would create a holder class such as the following:

```
class CardNumberType_TypeInfoHolder : public TObject {
  CardNumber __instanceType;
public:
__published:
  __property CardNumber __propType = {read=__instanceType };
};
```

and then register it as follows:

```
RemTypeRegistry()->RegisterXSInfo(GetClsMemberTypeInfo(
          __classid(CardNumber_TypeInfoHolder), "__propType"),
          MyNameSpace, "CardNumber");
```

For any other type declared using a typedef statement, call *RegisterXSInfo* if it maps to a dynamic array class, and *RegisterXSClass* if it maps to a class. For more information about registering dynamic arrays and classes, see "Registering nonscalar types" on page 33-6.

Note You should try to avoid typedef statements that map multiple types to the same underlying type. The runtime type information for these is not distinct.

Using remotable objects

Use *TRemotable* as a base class when defining a class to represent a complex data type on an invokable interface. For example, in the case where you would ordinarily pass a record or struct as a parameter, you would instead define a *TRemotable* descendant where every member of the record or struct is a published property on your new class.

You can control whether the published properties of your *TRemotable* descendant appear as element nodes or attributes in the corresponding SOAP encoding of the type. To make the property an attribute, use the stored directive on the property definition, assigning a value of AS_ATTRIBUTE:

```
property MyAttribute: Boolean read FMyAttribute write FMyAttribute stored AS_ATTRIBUTE;
```

```
__property bool MyAttribute =
                          {read=FMyAttribute, write=FMyAttribute, stored= AS_ATTRIBUTE;
```

Note If you do not include a stored directive, or if you assign any other value to the stored directive (even a function that returns AS_ATTRIBUTE), the property is encoded as a node rather than an attribute.

If the value of your new *TRemotable* descendant represents a scalar type in a WSDL document, you should use *TRemotableXS* as a base class instead. *TRemotableXS* is a *TRemotable* descendant that introduces two methods for converting between your new class and its string representation. Implement these methods by overriding the *XSToNative* and *NativeToXS* methods.

For certain commonly-used XML scalar types, the XSBuiltIns unit already defines and registers remotable classes for you. These are listed in the following table:

Table 33.2 Remotable classes

XML type	remotable class
dateTime timeInstant	TXSDateTime
date	TXSDate
time	TXSTime
duration timeDuration	TXSDuration
decimal	TXSDecimal
hexBinary	TXSHexBinary

After you define a remotable class, it must be registered with the remotable type registry, as described in "Registering nonscalar types" on page 33-6.This registration happens automatically on servers when you register the interface that uses the class. On clients, the code to register the class is generated automatically when you import the WSDL document that defines the type.

Tip It is a good idea to implement and register *TRemotable* descendants in a separate unit from the rest of your server application, including from the units that declare and register invokable interfaces. In this way, you can use the type for more than one interface.

Representing attachments

One important *TRemotable* descendant is *TSoapAttachment*. This class represents an attachment. It can be used as the value of a parameter or the return value of a method on an invokable interface. Attachments are sent with SOAP messages as separate parts in a multipart form.

When a Web Service application or the client of a Web Service receives an attachment, it writes the attachment to a temporary file. *TSoapAttachment* lets you access that temporary file or save its content to a permanent file or stream. When the application needs to send an attachment, it creates an instance of *TSoapAttachment* and assigns its content by specifying the name of a file, supplying a stream from which to read the attachment, or providing a string that represents the content of the attachment.

Managing the lifetime of remotable objects

One issue that arises when using *TRemotable* descendants is the question of when they are created and destroyed. Obviously, the server application must create its own local instance of these objects, because the caller's instance is in a separate process space. To handle this, Web Service applications create a data context for incoming requests. The data context persists while the server handles the request, and is freed after any output parameters are marshaled into a return message. When the server creates local instances of remotable objects, it adds them to the data context, and those instances are then freed along with the data context.

In some cases, you may want to keep an instance of a remotable object from being freed after a method call. For example, if the object contains state information, it may be more efficient to have a single instance that is used for every message call. To prevent the remotable object from being freed along with the data context, change its *DataContext* property.

Remotable object example

This example shows how to create a remotable object for a parameter on an invokable interface where you would otherwise use an existing class. In this example, the existing class is a string list (*TStringList*). To keep the example small, it does not reproduce the *Objects* property of the string list.

Because the new class is not scalar, it descends from *TRemotable* rather than *TRemotableXS*. It includes a published property for every property of the string list you want to communicate between the client and server. Each of these remotable properties corresponds to a remotable type. In addition, the new remotable class includes methods to convert to and from a string list.

D Delphi example

```
TRemotableStringList = class(TRemotable)
  private
    FCaseSensitive: Boolean;
    FSorted: Boolean;
    FDuplicates: TDuplicates;
    FStrings: TStringDynArray;
  public
    procedure Assign(SourceList: TStringList);
    procedure AssignTo(DestList: TStringList);
  published
    property CaseSensitive: Boolean read FCaseSensitive write FCaseSensitive;
    property Sorted: Boolean read FSorted write FSorted;
    property Duplicates: TDuplicates read FDuplicates write FDuplicates;
    property Strings: TStringDynArray read FStrings write FStrings;
end;
```

C++ example

```
class TRemotableStringList: public TRemotable
{
  private:
    bool FCaseSensitive;
    bool FSorted;
    Classes::TDuplicates FDuplicates;
    System::TStringDynArray FStrings;
  public:
    void __fastcall Assign(Classes::TStringList *SourceList);
    void __fastcall AssignTo(Classes::TStringList *DestList);
__published:
    __property bool CaseSensitive = {read=FCaseSensitive, write=FCaseSensitive};
    __property bool Sorted = {read=FSorted, write=FSorted};
    __property Classes::TDuplicates Duplicates = {read=FDuplicates, write=FDuplicates};
    __property System::TStringDynArray Strings = {read=FStrings, write=FStrings};
}
```

Note that *TRemotableStringList* exists only as a transport class. Thus, although it has a *Sorted* property (to transport the value of a string list's *Sorted* property), it does not need to sort the strings it stores, it only needs to record whether the strings should be sorted. This keeps the implementation very simple. You only need to implement the *Assign* and *AssignTo* methods, which convert to and from a string list:

Delphi example

```
procedure TRemotableStringList.Assign(SourceList: TStrings);
var I: Integer;
begin
  SetLength(Strings, SourceList.Count);
  for I := 0 to SourceList.Count - 1 do
    Strings[I] := SourceList[I];
  CaseSensitive := SourceList.CaseSensitive;
  Sorted := SourceList.Sorted;
  Duplicates := SourceList.Duplicates;
end;

procedure TRemotableStringList.AssignTo(DestList: TStrings);
var I: Integer;
begin
  DestList.Clear;
  DestList.Capacity := Length(Strings);
  DestList.CaseSensitive := CaseSensitive;
  DestList.Sorted := Sorted;
  DestList.Duplicates := Duplicates;
  for I := 0 to Length(Strings) - 1 do
    DestList.Add(Strings[I]);
end;
```

C++ example

```
void __fastcall TRemotableStringList::Assign(Classes::TStringList *SourceList)
{
  SetLength(Strings, SourceList->Count);
  for (int i = 0; i < SourceList->Count; i++)
```

```
    Strings[i] = SourceList->Strings[i];
  CaseSensitive = SourceList->CaseSensitive;
  Sorted = SourceList->Sorted;
  Duplicates = SourceList->Duplicates;
}
void __fastcall TRemotableStringList::AssignTo(Classes::TStringList *DestList)
{
  DestList->Clear();
  DestList->Capacity = Length(Strings);
  DestList->CaseSensitive = CaseSensitive;
  DestList->Sorted = Sorted;
  DestList->Duplicates = Duplicates;
  for (int i = 0; i < Length(Strings); i++)
    DestList->Add(Strings[i]);
}
```

Optionally, you may want to register the new remotable class so that you can specify its class name. If you do not register the class, it is registered automatically when you register the interface that uses it. Similarly, if you register the class but not the *TDuplicates* and *TStringDynArray* types that it uses, they are registered automatically. This code shows how to register the *TRemotableStringList* class and (in Delphi) the *TDuplicates* type. *TStringDynArray* is registered automatically because it is one of the built-in dynamic array types declared in the Types unit (Delphi) or sysdyn.h (C++). For details on explicitly registering an enumerated type such as *TDuplicates* in C++, see "Using holder classes to register C++ types" on page 33-7.

In Delphi, this registration code goes in the initialization section of the unit where you define the remotable class:

```
RemClassRegistry.RegisterXSInfo(TypeInfo(TDuplicates), MyNameSpace, 'duplicateFlag');
RemClassRegistry.RegisterXSClass(TRemotableStringList, MyNameSpace, 'stringList', '',False);
```

In C++, the registration code goes in the startup code (the *RegTypes* function):

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TRemotableStringList), MyNameSpace,
"stringList", "", false);
}
#pragma startup initServices 32
```

Writing servers that support Web Services

In addition to the invokable interfaces and the classes that implement them, your server requires two components: a dispatcher and an invoker. The dispatcher (*THTTPSoapDispatcher*) receives incoming SOAP messages and passes them on to the invoker. The invoker (*THTTPSoapPascalInvoker* in Delphi or *THTTPCppInvoker* in C++) interprets the SOAP message, identifies the invokable interface it calls, executes the call, and assembles the response message.

Note *THTTPSoapDispatcher*, *THTTPSoapPascalInvoker*, and *THTTPSoapCppInvoker* are designed to respond to HTTP messages containing a SOAP request. The underlying architecture is sufficiently general, however, that it can support other protocols with the substitution of different dispatcher and invoker components.

Once you register your invokable interfaces and their implementation classes, the dispatcher and invoker automatically handle any messages that identify those interfaces in the SOAP Action header of the HTTP request message.

Web services also include a publisher (*TWSDLHTMLPublish*). Publishers respond to incoming client requests by creating the WSDL documents that describe how to call the Web Services in the application.

Building a Web Service server

Use the following steps to build a server application that implements a Web Service:

1. Choose File | New | Other and on the WebServices tab, double-click the Soap Server Application icon to launch the SOAP Server Application wizard. The wizard creates a new Web server application that includes the components you need to respond to SOAP requests. For details on the SOAP application wizard and the code it generates, see "Using the SOAP application wizard" on page 33-14.
2. When you exit the SOAP Server Application wizard, it asks you if you want to define an interface for your Web Service. If you are creating a Web Service from scratch, click yes, and you will see the Add New Web Service wizard. The wizard adds code to declare and register a new invokable interface for your Web Service. Edit the generated code to define and implement your Web Service. If you want to add additional interfaces (or you want to define the interfaces at a later time), choose File | New | Other, and on the WebServices tab, double-click the SOAP Web Service interface icon. For details on using the Add New Web Service wizard and completing the code it generates, see "Adding new Web Services" on page 33-15.
3. If you are implementing a Web Service that has already been defined in a WSDL document, you can use the Web Services Importer to generate the interfaces, implementation classes, and registration code that your application needs. You need only fill in the body of the methods the importer generates for the implementation classes. For details on using the Web Services Importer, see "Using the Web Services Importer" on page 33-17.
4. If your application raises an exception when attempting to execute a SOAP request, the exception will be automatically encoded in a SOAP fault packet, which is returned instead of the results of the method call. If you want to convey more information than a simple error message, you can create your own exception classes that are encoded and passed to the client. This is described in "Creating custom exception classes for Web Services" on page 33-18.
5. The SOAP Server Application wizard adds a publisher component (*TWSDLHTMLPublish)* to new Web Service applications. This enables your application to publish WSDL documents that describe your Web Service to clients. For information on the WSDL publisher, see "Generating WSDL documents for a Web Service application" on page 33-18.

Using the SOAP application wizard

Web Service applications are a special form of Web Server application. Because of this, support for Web Services is built on top of the Web Broker architecture. To understand the code that the SOAP Application wizard generates, therefore, it is helpful to understand the Web Broker architecture. Information about Web Server applications in general, and Web Broker in particular, can be found in Chapter 29, "Creating Internet server applications" and Chapter 30, "Using Web Broker."

To launch the SOAP application wizard, choose File | New | Other, and on the WebServices page, double-click the Soap Server Application icon. Choose the type of Web server application you want to use for your Web Service. For information about different types of Web Server applications, see "Types of Web server applications" on page 29-6.

The wizard generates a new Web server application that includes a Web module which contains three components:

- An invoker component (*THTTPSOAPPascalInvoker* in Delphi or *THTTPSOAPCppInvoker* in C++). The invoker converts between SOAP messages and the methods of any registered invokable interfaces in your Web Service application.
- A dispatcher component (*THTTPSoapDispatcher*). The dispatcher automatically responds to incoming SOAP messages and forwards them to the invoker. You can use its *WebDispatch* property to identify the HTTP request messages to which your application responds. This involves setting the *PathInfo* property to indicate the path portion of any URL directed to your application, and the *MethodType* property to indicate the method header for request messages.
- A WSDL publisher (*TWSDLHTMLPublish*). The WSDL publisher publishes a WSDL document that describes your interfaces and how to call them. The WSDL document tells clients that how to call on your Web Service application. For details on using the WSDL publisher, see "Generating WSDL documents for a Web Service application" on page 33-18.

The SOAP dispatcher and WSDL publisher are auto-dispatching components. This means they automatically register themselves with the Web module so that it forwards any incoming requests addressed using the path information they specify in their *WebDispatch* properties. If you right-click on the Web module, you can see that in addition to these auto-dispatching components, it has a single Web action item named *DefaultHandler*.

DefaultHandler is the default action item. That is, if the Web module receives a request for which it can't find a handler (can't match the path information), it forwards that message to the default action item. *DefaultHandler* generates a Web page that describes your Web Service. To change the default action, edit this action item's *OnAction* event handler.

Adding new Web Services

To add a new Web Service interface to your server application, choose File | New | Other, and on the WebServices tab double-click on the icon labeled SOAP Server Interface.

The Add New Web Service wizard lets you specify the name of the invokable interface you want to expose to clients, and generates the code to declare and register the interface and its implementation class. By default, the wizard also generates comments that show sample methods and additional type definitions, to help you get started in editing the generated files.

Editing the generated code

The interface definitions appear in the interface section of the generated unit (Delphi) or in its header file (in C++). This generated unit has the name you specified using the wizard. You will want to change the interface declaration, replacing the sample methods with the methods you are making available to clients.

The wizard generates an implementation class that descends from *TInvokableClass* and that supports the invokable interface). If you are defining an invokable interface from scratch, you must edit the declaration of the implementation class to match any edits you made to the generated invokable interface.

When adding methods to the invokable interface and implementation class, remember that the methods must only use remotable types. For information on remotable types and invokable interfaces, see "Using nonscalar types in invokable interfaces" on page 33-4.

Using a different base class

The Add New Web Service wizard generates implementation classes that descend from *TInvokableClass*. This is the easiest way to create a new class to implement a Web Service. You can, however, replace this generated class with an implementation class that has a different base class (for example, you may want to use an existing class as a base class.) There are a number of considerations to take into account when you replace the generated implementation class:

- Your new implementation class must support (in C++ descend from) the invokable interface directly. The invocation registry, with which you register invokable interfaces and their implementation classes, keeps track of what class implements each registered interface and makes it available to the invoker component when the invoker needs to call the interface. It can only detect that a class implements an interface if the interface is directly included in the class declaration. It does not detect support an interface if it is inherited along with a base class.
- Your new implementation class must include support for the *IUnknown* or *IInterface* methods that are part of any interface. This point may seem obvious, but it is an easy one to overlook. For more information about implementing *IUnknown* in C++, see "Creating classes that support IUnknown" on page 14-6.
- You must change the generated code that registers the implementation class to include a factory method to create instances of your implementation class.

This last point takes a bit of explanation. When the implementation class descends from *TInvokableClass* and does not replace the inherited constructor with a new constructor that includes one or more parameters, the invocation registry knows how to create instances of the class when it needs them. When you write an implementation class that does not descend from *TInvokableClass*, or when you change the constructor, you must tell the invocation registry how to obtain instances of your implementation class.

You can tell the invocation registry how to obtain instances of your implementation class by supplying it with a factory procedure. Even if you have an implementation class that descends from *TInvokableClass* and that uses the inherited constructor, you may want to supply a factory procedure anyway. For example, you can use a single global instance of your implementation class rather than requiring the invocation registry to create a new instance every time your application receives a call to the invokable interface.

The factory procedure must be of type *TCreateInstanceProc*. It returns an instance of your implementation class. If the procedure creates a new instance, the implementation object should free itself when the reference count on its interface drops to zero, as the invocation registry does not explicitly free object instances. The following code illustrates another approach, where the factory procedure returns a single global instance of the implementation class:

Delphi example

```
procedure CreateEncodeDecode(out obj: TObject);
begin
  if FEncodeDecode = nil then
  begin
    FEncodeDecode := TEncodeDecode.Create;
    {save a reference to the interface so that the global instance doesn't free itself }
    FEncodeDecodeInterface := FEncodeDecode as IEncodeDecode;
  end;
  obj := FEncodeDecode; { return global instance }
end;
```

C++ example

```
void __fastcall CreateEncodeDecode(System::TObject* &obj)
{
  if (!FEncodeDecode)
  {
    FEncodeDecode = new TEncodeDecodeImpl();
    // save a reference to the interface so that the global instance doesn't free itself
    TEncodeDecodeImpl->QueryInterface(FEncodeDecodeInterface);
  }
  obj = FEncodeDecode;
}
```

Note In the C++ example, *FEncodeDecodeInterface* is a variable of type *_di_IEncodeDecode*. In the Delphi example, it is of type *IEncodeDecode*.

You register the factory procedure with an implementation class by supplying it as a second parameter to the call that registers the class with the invocation registry. First, locate the call the wizard generated to register the implementation class.

In Delphi, this appears in initialization section of the unit that defines the class. It looks something like the following:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode);
```

In C++, it appears in the RegTypes method at the bottom of the unit, and looks something like the following:

```
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl));
```

Add a second parameter to this call that specifies the factory procedure:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode, CreateEncodeDecode);
```

```
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl), &CreateEncodeDecode);
```

Using the Web Services Importer

To use the Web Services Importer, choose File | New | Other, and on the WebServices page double-click the icon labeled Web Services Importer. In the dialog that appears, specify the file name of a WSDL document (or XML file) or provide the URL where that document is published.

If the WSDL document is on a server that requires authentication (or must be reached using a proxy server that requires authentication), you need to provide a user name and password before the wizard can retrieve the WSDL document. To supply this information, click the Options button and provide the appropriate connection information.

When you click the Next button, the Web Services Importer displays the code it generates for every definition in the WSDL document that is compatible with the Web Services framework. That is, it only uses those port types that have a SOAP binding. You can configure the way the importer generates code by clicking the Options button and choosing the options you want.

You can use the Web Services Importer when writing either a server or a client application. When writing a server, click the Options button and in the resulting dialog, check the option that tells the importer to generate server code. When you select this option, the importer generates implementation classes for the invokable interfaces, and you need only fill in the bodies of the methods.

Warning If you import a WSDL document to create a server that implements a Web Service that is already defined, you must still publish your own WSDL document for that service. There may be minor differences in the imported WSDL document and the generated implementation. For example, If the WSDL document or XML schema file uses identifiers that are also keywords, the importer automatically adjusts their names so that the generated code can compile.)

When you click Finish, the importer creates new units that define and register invokable interfaces for the operations defined in the document, and that define and register remotable classes for the types that the document defines.

As an alternate approach, you can use the command line WSDL importer instead. For a server, call the command line importer with the -S option, as follows:

```
WSDLIMP -S -P -V MyWSDLDoc.wsdl
WSDLIMP -S -C -V MyWSDLDoc.wsdl
```

For a client application, call the command line importer without the -S option:

```
WSDLIMP -P -V MyWSDLDoc.wsdl
WSDLIMP -C -V MyWSDLDoc.wsdl
```

Creating custom exception classes for Web Services

When your Web Service application raises an exception in the course of trying to execute a SOAP request, it automatically encodes information about that exception in a SOAP fault packet, which it returns instead of the results of the method call. The client application then raises the exception.

By default, the client application raises a generic exception of type *ERemotableException*with the information from the SOAP fault packet. You can transmit additional, application-specific information by deriving an *ERemotableException* descendant. The values of any published properties you add to the exception class are included in the SOAP fault packet so that the client can raise an equivalent exception.

To use an *ERemotableException* descendant, you must register it with the remotable type registry. Thus, in the unit that defines your *ERemotableException* descendant, you must add the InvokeRegistry unit to the uses clause (Delphi) or include InvokeRegistry.hpp (C++) and add a call to the *RegisterXSClass* method of the object that the global *RemTypeRegistry* function returns.

If the client also defines and registers your *ERemotableException* descendant, then when it receives the SOAP fault packet, it automatically raises an instance of the appropriate exception class, with all properties set to the values in the SOAP fault packet.

Generating WSDL documents for a Web Service application

To allow client applications to know what Web Services your application makes available, you can publish a WSDL document that describes your invokable interfaces and indicates how to call them.

In C++, you must always publish a WSDL document for your Web Service application, even if you implemented an imported service or if your client application can use the same header files as your server.

To publish a WSDL document that describes your Web Service, include a *TWSDLHTMLPublish* component in your Web Module. (The SOAP Server Application wizard adds this component by default.) *TWSDLHTMLPublish* is an auto-dispatching component, which means it automatically responds to incoming messages that request a list of WSDL documents for your Web Service. Use the *WebDispatch* property to specify the path information of the URL that clients must use to access the list of WSDL documents. The Web browser can then request the list of WSDL documents by specifying an URL that is made up of the location of the server application followed by the path in the *WebDispatch* property. This URL looks something like the following:

```
http://www.myco.com/MyService.dll/WSDL
```

Tip If you want to use a physical WSDL file instead, you can display the WSDL document in your Web browser and then save it to generate a WSDL document file.

It is not necessary to publish the WSDL document from the same application that implements your Web Service. To create an application that simply publishes the WSDL document, omit the code that implements and registers the implementation objects and only include the code that defines and registers invokable interfaces, remotable classes that represent complex types, and any remotable exceptions.

By default, when you publish a WSDL document, it indicates that the services are available at the same URL as the one where you published the WSDL document (but with a different path). If you are deploying multiple versions of your Web Service application, or if you are publishing the WSDL document from a different application than the one that implements the Web Service, you will need to change the WSDL document so that it includes updated information on where to locate the Web Service.

To change the URL, use the WSDL administrator. The first step is to enable the administrator. You do this by setting the *AdminEnabled* property of the *TWSDLHTMLPublish* component to true. Then, when you use your browser to display the list of WSDL documents, it includes a button to administer them as well. Use the WSDL administrator to specify the locations (URLs) where you have deployed your Web Service application.

Writing clients for Web Services

You can write clients that access Web Services that you have written, or any other Web Service that is defined in a WSDL document. There are two steps to writing an application that is the client of a Web Service: importing the definitions from a WSDL document, and using a remote interfaced object to call the Web Service.

Importing WSDL documents

Before you can use a Web Service, your application must define and register the invokable interfaces and types that are included in the Web Service application. To obtain these definitions, you can import a WSDL document (or XML file) that defines the service. The Web Services importer creates a unit that defines and registers the interfaces and types you need to use. For details on using the Web Services Importer, see “Using the Web Services Importer” on page 33-17.

Calling invokable interfaces

To call an invokable interface, your client application must include any definitions of the invokable interfaces and any remotable classes that implement complex types.

If the server is written in Delphi, you can use the same units that the server application uses to define and register these interfaces and classes instead of the files generated by importing a WSDL file. Be sure that the unit uses the same namespace URI and SOAPAction header when it registers invokable interfaces. These values can be explicitly specified in the code that registers the interfaces, or it can be automatically generated. If it is automatically generated, the unit that defines the interfaces must have the same name in both client and server, and both client and server must define the global *AppSpacePrefix* variable to have the same value.

Once the client application has the declaration of an invokable interface, create an instance of *THTTPRio* for the desired interface:

```
X := THTTPRio.Create(nil);
```

```
X = new THTTPRio(NULL);
```

Note It is important that you do not explicitly destroy the *THTTPRio* instance. If it is created without an *Owner* (as in the previous line of code), it automatically frees itself when its interface is released. If it is created with an *Owner*, the *Owner* is responsible for freeing the *THTTPRio* instance.

Next, provide the *THTTPRio* object with the information it needs to identify the server interface and locate the server. There are two ways to supply this information:

- If you do not expect the URL for the Web Service or the namespaces and soap Action headers it requires to change, you can simply specify the URL for the Web Service you want to access. *THTTPRio* uses this URL to look up the definition of the interface, plus any namespace and header information, based on the information in the invocation registry. Specify the URL by setting the *URL* property to the location of the server:

```
X.URL := 'http://www.myco.com/MyService.dll/SOAP/IServerInterface';
```

```
X->URL = "http://www.myco.com/MyService.dll/SOAP/IServerInterface";
```

- If you want to look up the URL, namespace, or Soap Action header from the WSDL document dynamically at runtime, you can use the *WSDLLocation*, *Service*, and *Port* properties, and it will extract the necessary information from the WSDL document:

```
X.WSDLLocation := 'Cryptography.wsdl';
X.Service := 'Cryptography';
X.Port := 'SoapEncodeDecode';
```

```
X.WSDLLocation = "Cryptography.wsdl";
X.Service = "Cryptography";
X.Port = "SoapEncodeDecode";
```

Once you have specified how to locate the server and identify the interface, you can obtain an interface pointer for the invokable interface from the *THTTPRio* object.

In Delphi, you obtain this interface pointer using the as operator. Simply cast the *THTTPRio* instance to the invokable interface:

```
InterfaceVariable := X as IEncodeDecode;
Code := InterfaceVariable.EncodeValue(5);
```

In C++, you use the *QueryInterface* method to obtain an interface pointer for the invokable interface. Note that the call to *QueryInterface* takes as an argument the *DelphiInterface* wrapper for the invokable interface rather than the invokable interface itself.

```
_di_IEncodeDecode InterfaceVariable;
X->QueryInterface(InterfaceVariable);
if (InterfaceVariable)
{
  Code = InterfaceVariable->EncodeValue(5);
}
```

When you obtain the interface pointer, *THTTPRio* creates a vtable for the associated interface dynamically in memory, enabling you to make interface calls.

THTTPRio relies on the invocation registry to obtain information about the invokable interface. If the client application does not have an invocation registry, or if the invokable interface is not registered, *THTTPRio* can't build its in-memory vtable.

Warning If you assign the interface you obtain from *THTTPRio* to a global variable, you must change that assignment to nil (Delphi) or NULL (C++) before shutting down your application. For example, if *InterfaceVariable* in the previous code sample is a global variable, rather than stack variable, you must release the interface before the *THTTPRio* object is freed. Typically, this code goes in the *OnDestroy* event handler of the form or data module:

Delphi example

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  InterfaceVariable := nil;
end;
```

C++ example

```
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  InterfaceVariable = NULL;
}
```

The reason you must reassign a global interface variable to nil (Delphi) or NULL (C++) is because *THTTPRio* builds its vtable dynamically in memory. That vtable must still be present when the interface is released. If you do not release the interface along with the form or data module, it is released when the global variable is freed on shutdown. The memory for global variables may be freed after the form or data module that contains the *THTTPRio* object, in which case the vtable will not be available when the interface is released.

Chapter

34

Working with sockets

This chapter describes the socket components that let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the underlying networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as User Datagram Protocol (UDP), Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family.

Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is usually dedicated to a single service such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Using server sockets, an application that provides one of these services can link to client applications that want to use that service. Client sockets allow an application that uses one of these services to link to server applications that provide the service.

Implementing services

Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with distributed data sets, you may want to write a layer to communicate with databases on other systems.

Understanding service protocols

Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using.

If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for the servers and clients of this service. What messages are sent? How are these messages coordinated? How is the information encoded?

Communicating with applications

Often, your network server or client application provides a layer between the networking software and an application that uses the service. For example, an HTTP server sits between the Internet and a Web server application that provides content and responds to HTTP request messages.

Sockets provide the interface between your network server or client application and the networking software. You must provide the interface between your application and the clients that use it. You can copy the API of a standard third party server (such as Apache), or you can design and publish your own API.

Services and ports

Most standard services are associated, by convention, with specific port numbers. We will discuss port numbers in greater detail later. For now, consider the port number a numeric code for the service.

If you are implementing a standard service for use in cross-platform applications, Linux socket objects provide methods for you to look up the port number for the service. If you are providing a new service, you can specify the associated port number in the /etc/services file. See your Linux documentation for more information on the services file.

Types of socket connections

Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are

- Client connections.
- Listening connections.
- Server connections.

Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

Client connections

Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket to which it wishes to connect. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description of the server socket to which it is connecting, and the connection is completed by the client.

Listening connections

Server sockets do not locate clients. Instead, they form passive "half connections" that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.

Server connections

Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.

Describing sockets

Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies:

- The system on which it is running.
- The types of interfaces it understands.
- The port it is using for the connection.

A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host and the port number.

Before you can make a socket connection, you must fully describe the sockets that form its endpoints. Some of the information is available from the system your application is running on. For instance, you do not need to describe the local IP address of a client socket—this information is available from the operating system.

The information you must provide depends on the type of socket you are working with. Client sockets must describe the server they want to connect to. Listening server sockets must describe the port that represents the service they provide.

Describing the host

The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as

```
123.197.1.2
```

A single system may support more than one IP address.

IP addresses are often difficult to remember and easy to mistype. An alternative is to use the host name. Host names are aliases for the IP address that you often see in Uniform Resource Locators (URLs). They are strings containing a domain name and service, such as

```
http://www.ASite.com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. You can learn the host name associated with any IP address (if one already exists) by executing the following command from a command prompt:

```
nslookup IPADDRESS
```

where *IPADDRESS* is the IP address you're interested in. If your local IP address doesn't have a host name and you decide you want one, contact your network administrator. It is common for computers to refer to themselves with the name *localhost* and the IP number 127.0.0.1.

Server sockets do not need to specify a host. The local IP address can be read from the system. If the local system supports more than one IP address, server sockets will listen for client requests on all IP addresses simultaneously. When a server socket accepts a connection, the client socket provides the remote IP address.

Client sockets must specify the remote host by providing either its host name or IP address.

Choosing between a host name and an IP address

Most applications use the host name to specify a system. Host names are easier to remember, and easier to check for typographical errors. Further, servers can change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If the host name is unknown, the client socket must specify the server system using its IP address. Specifying the server system by giving the IP address is faster. When you provide the host name, the socket must search for the IP address associated with the host name, before it can locate the server system.

Using ports

While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number.

Earlier, we described port numbers as numeric codes for the services implemented by network applications. This is actually just a convention that allows listening server connections to make themselves available on a fixed port number so that they can be found by client sockets. Server sockets listen on the port number associated with the service they provide. When they accept a connection to a client socket, they create a separate socket connection that uses a different, arbitrary, port number. This way, the listening connection can continue to listen on the port number associated with the service.

Client sockets use an arbitrary local port number, as there is no need for them to be found by other sockets. They specify the port number of the server socket to which they want to connect so that they can find the server application. Often, this port number is specified indirectly, by naming the desired service.

Using socket components

The Internet palette page includes three socket components that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. These are:

- *TcpServer*
- *TcpClient*
- *UdpSocket*

Associated with each of these socket components are socket objects, which represent the endpoint of an actual socket connection. The socket components use the socket objects to encapsulate the socket server calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages.

If you want to customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the socket objects.

Getting information about the connection

After completing the connection to a client or server socket, you can use the client or server socket object associated with your socket component to obtain information about the connection. Use the *LocalHost* and *LocalPort* properties to determine the address and port number used by the local client or server socket, or use the *RemoteHost* and *RemotePort* properties to determine the address and port number used by the remote client or server socket. Use the *GetSocketAddr* method to build a valid socket address based on the host name and port number. You can use the *LookupPort* method to look up the port number. Use the *LookupProtocol* method to look up the protocol number. Use the *LookupHostName* method to look up the host name based on the host machine's IP address.

To view network traffic in and out of the socket, use the *BytesSent* and *BytesReceived* properties.

Using client sockets

Add a *TcpClient* or *UdpSocket* component to your form or data module to turn your application into a TCP/IP or UDP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server.

Each client socket component uses a single client socket object to represent the client endpoint in a connection.

Specifying the desired server

Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. Use the *RemoteHost* property to specify the remote host server by either its host name or IP address.

In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can use the *RemotePort* property to specify the server port number directly or indirectly by naming the target service.

Forming the connection

Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the *Open* method. If you want your application to form the connection automatically when it starts up, set the *Active* property to true at design time, using the Object Inspector.

Getting information about the connection

After completing the connection to a server socket, you can use the client socket object associated with your client socket component to obtain information about the connection. Use the *LocalHost* and *LocalPort* properties to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the *Handle* property to obtain a handle to the socket connection to use when making socket calls.

Closing the connection

When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the *Close* method. The connection may also be closed from the server end. If that is the case, you will receive notification in an *OnDisconnect* event.

Using server sockets

Add a server socket component (*TcpServer* or *UdpSocket*) to your form or data module to turn your application into an IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests.

Each server socket component uses a single server socket object to represent the server endpoint in a listening connection. It also uses a server client socket object for the server endpoint of each active connection to a client socket that the server accepts.

Specifying the port

Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the *LocalPort* property. If your server application is providing a standard service that is associated by convention with a specific port number, you can also specify the service name using the *LocalPort* property. It is a good idea to use the service name instead of a port number, because it is easy to introduce typographical errors when specifying the port number.

Listening for client requests

Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the *Open* method. If you want your application to form the listening connection automatically when it starts up, set the *Active* property to true at design time, using the Object Inspector.

Connecting to clients

A listening server socket component automatically accepts client connection requests when they are received. You receive notification every time this occurs in an *OnAccept* event.

Closing server connections

When you want to shut down the listening connection, call the *Close* method or set the *Active* property to false. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections.

When TCP clients shut down their individual connections to your server socket, you are informed by an *OnDisconnect* event.

Responding to socket events

When writing applications that use sockets, you can write or read to the socket anywhere in the program. You can write to the socket using the *SendBuf*, *SendStream*, or *Sendln* methods in your program after the socket has been opened. You can read from the socket using the similarly-named methods *ReceiveBuf* and *Receiveln.* The *OnSend* and *OnReceive* events are triggered every time something is written or read from the socket. They can be used for filtering. Every time you read or write, a read or write event is triggered.

Both client sockets and server sockets generate error events when they receive error messages from the connection.

Socket components also receive two events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds, you must use the *SendBuf* and *ReceiveBuf* methods to respond to these client events or server events.

Error events

Client and server sockets generate *OnError* events when they receive error messages from the connection. You can write an *OnError* event handler to respond to these error messages. The event handler is passed information about

- What socket object received the error notification.
- What the socket was trying to do when the error occurred.
- The error code that was provided by the error message.

You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.

Client events

When a client socket opens a connection, the following events occur:

- The socket is set up and initialized for event notification.
- An *OnCreateHandle* event occurs after the server and server socket is created. At this point, the socket object available through the *Handle* property can provide information about the server or client socket that will form the other end of the connection. This is the first chance to obtain the actual port used for the connection, which may differ from the port of the listening sockets that accepted the connection.

- The connection request is accepted by the server and completed by the client socket.
- When the connection is established, the *OnConnect* notification event occurs.

Server events

Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

Events when listening

Just before the listening connection is formed, the *OnListening* event occurs. You can use its *Handle* property to make changes to the socket before it is opened for listing. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnListening* event handler.

Events with client connections

When a server socket accepts a client connection request, the following events occur:

- An *OnAccept* event occurs, passing in the new *TTcpClient* object to the event handler. This is the first point when you can use the properties of *TTcpClient* to obtain information about the server endpoint of the connection to a client.
- If *BlockMode* is *bmThreadBlocking* an *OnGetThread* event occurs. If you want to provide your own customized descendant of *TServerSocketThread*, you can create one in an *OnGetThread* event handler, and that will be used instead of *TServerSocketThread*. If you want to perform any initialization of the thread, or make any socket API calls before the thread starts reading or writing over the connection, you should use the *OnGetThread* event handler for these tasks as well.
- The client completes the connection and an *OnAccept* event occurs. With a non-blocking server, you may want to start reading or writing over the socket connection at this point.

Reading and writing over socket connections

The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection.

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection. You can also form blocking connections, where your application waits for the reading or writing to be completed before executing the next line of code.

Non-blocking connections

Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in you network application. To create a non-blocking connection for client or server sockets, set the *BlockMode* property to *bmNonBlocking.*

When the connection is non-blocking, reading and writing events inform your socket when the socket on the other end of the connection tries to read or write information.

Reading and writing events

Non-blocking sockets generate reading and writing events when they need to read or write over the connection. You can respond to these notifications in an *OnReceive* or *OnSend* event handler.

The socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This socket object provides a number of methods to allow you to read or write over the connection.

To read from the socket connection, use the *ReceiveBuf* or *Receiveln* method. To write to the socket connection, use the *SendBuf*, *SendStream*, or *Sendln* method.

Blocking connections

When the connection is blocking, your socket must initiate reading or writing over the connection. It cannot wait passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place.

For client or server sockets, set the *BlockMode* property to *bmBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading or writing over the connection to be completed.

For server sockets, set the *BlockMode* property to *bmBlocking* or *bmThreadBlocking* to form a blocking connection. Because blocking connections hold up the execution of all other code while the socket waits for information to be written or read over the connection, server socket components always spawn a new execution thread for every client connection when the *BlockMode* is *bmThreadBlocking*. When the *BlockMode* is *bmBlocking*, program execution is blocked until a new connection is established.

Part

IV

Creating custom components

The chapters in “Creating custom components” present concepts necessary for designing and implementing custom components.

Chapter

35

Overview of component creation

This chapter provides an overview of component design and the process of writing components for CLX applications. The material here assumes that you are familiar with Kylix and its standard components.

- Component Library for Cross-Platform (CLX)
- Components and classes
- Creating components
- What goes into a component?
- Creating a new component
- Installing a component on the Component palette
- Testing installed components

For information on installing new components, see "Installing component packages" on page 16-7.

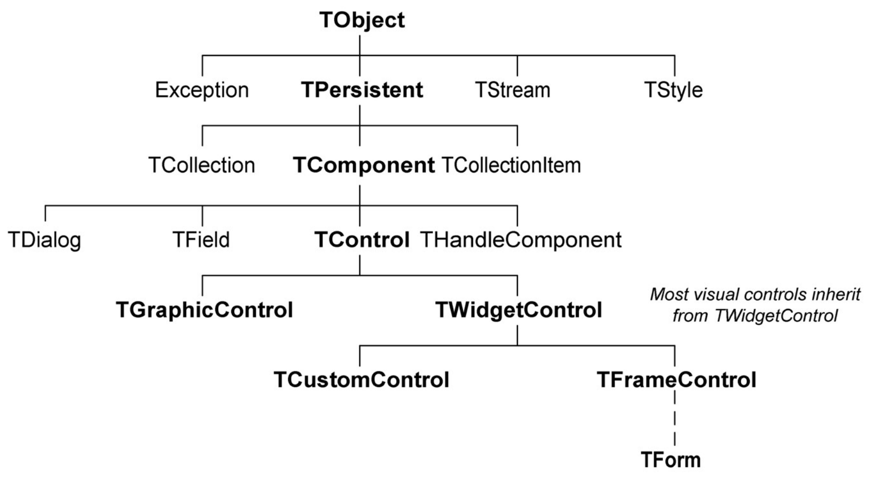
Component Library for Cross-Platform (CLX)

Kylix's components are part of a class hierarchy called the Component Library for Cross-Platform (CLX). Figure 35.1 shows the relationship of selected classes that make up CLX.

For a more detailed discussion of the class hierarchy and the inheritance relationships among classes, see Chapter 36, "Object-oriented programming for component writers" and refer to the CLX online reference for details on the components. Refer to Chapter 3, Understanding the class libraries for more information about CLX basics.

The *TComponent* class is the shared ancestor of every component in CLX. *TComponent* provides the minimal properties and events necessary for a component to work in the IDE. The various branches of the library provide other, more specialized capabilities.

Figure 35.1 CLX class hierarchy



When you create a component, you add to CLX by deriving a new class from one of the existing class types in the hierarchy.

Components and classes

Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes.

Briefly, there are two main differences between creating components and using them in applications. When creating components,

- You access parts of the class that are inaccessible to application programmers.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.

Creating components

A component can be almost any program element that you want to manipulate at design time. Creating a component means deriving a new class from an existing one. You can derive a new component in several ways:

- Modifying existing components
- Creating controls
- Creating graphic controls
- Subclassing controls
- Creating nonvisual components

Table 35.1 summarizes the different kinds of components and the classes you use as starting points for each.

Table 35.1 Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as *TButton* or *TListBox*, or an abstract component type, such as *TCustomListBox*
Create a widget-based control	*TWidgetControl*
Create a graphic control	*TGraphicControl*
Subclassing a control	Any widget-based control
Create a nonvisual component	*TComponent*

You can also derive classes that are not components and cannot be manipulated on a form, such as *TFont*.

Modifying existing components

The simplest way to create a component is to customize an existing one. You can derive a new component from any of the existing components provided.

Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, CLX includes an abstract class (with the word “custom” in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide) a property inherited from an ancestor class, so you need to derive your component from something above *TListBox* in the hierarchy. Rather than force you to start from the abstract *TWidgetControl* class and reinvent all the list box functions, CLX provides *TCustomListBox*, which implements the properties of a list box but does not publish all of them. When you derive a component from an abstract class like *TCustomListBox*, you publish only the properties you want to make available in your component and leave the rest protected.

Chapter 37, “Creating properties,” explains publishing inherited properties. Chapter 43, “Modifying an existing component,” and Chapter 45, “Customizing a grid,” show examples of modifying existing controls.

Creating controls

Controls in CLX are objects that appear at runtime and that the user can interact with.In CLX, these controls are widget-based controls. Each widget-based control has a handle, accessed through its *Handle* property, that identifies the underlying widget.

The *TWidgetControl* class is the base class for all of the user interface widgets. All widget-based controls descend from the *TWidgetControl* class. These include most standard controls, such as pushbuttons, list boxes, and edit boxes. While you could derive an original control (one that’s not related to any existing control) directly from

TWidgetControl, CLX provides the *TCustomControl* component for this purpose. *TCustomControl* is a specialized control that makes it easier to draw complex visual images.

You can either create a custom control based on *TCustomControl* or use a widget (such as a widget that is not already encapsulated by CLX) and encapsulate it yourself by descending from a related object (or *TWidgetControl* itself).

Chapter 45, "Customizing a grid," presents an example of creating a control.

Creating graphic controls

If your control does not need to receive input focus, you can make it a graphic control. Components like *TLabel*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to system events.

Kylix supports the creation of custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a canvas to paint on.

Chapter 44, "Creating a graphic control," presents an example of creating a graphic control.

Subclassing controls

You create custom controls by defining a new widget. The widget contains information shared among instances of the same sort of control; you can base a new widget on an existing class, which is called *subclassing*. You then put your control in a shared object file, much like the standard controls, and provide an interface to it.

Using Kylix, you can create a component "wrapper" around any existing widget class. So if you already have a library of custom controls that you want to use in Kylix applications, you can create Kylix components that behave like your controls, and derive new controls from them just as you would with any other component.

For examples of the techniques used in subclassing controls, see the components in the QStdCtls unit (Delphi) or header file (C++) that represent standard Windows controls, such as *TEdit*.

Creating nonvisual components

Nonvisual components are used as interfaces for elements like databases (*TDataSet* or *TSQLConnection*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TDialog* and its descendants). Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components. *THandleComponent* is the base class for components that require a handle to the underlying widget, such as menus.

What goes into a component?

To make your components reliable parts of the Kylix environment, you need to follow certain conventions in their design. This section discusses the following topics:

- Removing dependencies
- Setting properties, methods, and events
- Encapsulating graphics
- Registering components

Removing dependencies

One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions.

An example of removing dependencies is the *Handle* property of widget controls. If you have written GUI applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you do not try to access a window or control until you have created a unique identifier for the instance of the underlying widget. You can use *Handle* when making low-level function calls, for example, into the Qt shared libraries, where the function call requires a unique identifier for the QWidget object. If the widget does not exist, reading *Handle* causes the component to create the underlying widget. Thus, whenever an application's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing background tasks like creating the a widget, Kylix components allow developers to focus on what they really want to do. Before passing a handle to a widget, you do not need to verify that the handle exists or to create the window. The application developer can assume that things will work, instead of constantly checking for things that might go wrong.

Although it can take time to create components that are free of dependencies, it is generally time well spent. It not only spares application developers from repetition and drudgery, but it reduces your documentation and support burdens.

Setting properties, methods, and events

Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a chapter devoted to it in this book, but the discussion that follows explains some of the motivation for their use.

Properties

Properties give the application developer the illusion of setting or reading the value of a variable, while allowing the component writer to hide the underlying data structure or to implement special processing when the value is accessed.

There are several advantages to using properties:

- Properties are available at design time. The application developer can set or change initial values of properties without having to write code.
- Properties can check values or formats as the application developer assigns them. Validating input at design time prevents errors.
- The component can construct appropriate values on demand. Perhaps the most common type of error programmers make is to reference a variable that has not been initialized. By representing data with a property, you can ensure that a value is always available on demand.
- Properties allow you to hide data under a simple, consistent interface. You can alter the way information is structured in a property without making the change visible to application developers.

Chapter 37, "Creating properties," explains how to add properties to your components.

Methods

Application developers use methods to direct a component to perform a specific action or return a value not contained by any property. There are two types of methods: class methods and component methods. Class methods are procedures and functions that operate on a class rather than on specific instances of the class. For example, every component's *Create* constructor is a class method. Component methods are procedures and functions that operate on the component instances themselves. Application developers use methods to direct a component to perform a specific action or return a value not contained by any property.

Because they require execution of code, methods can be called only at runtime. Methods are useful for several reasons:

- Methods encapsulate the functionality of a component in the same object where the data resides.
- Methods can hide complicated procedures under a simple, consistent interface. An application developer can call a component's *AlignControls* method without knowing how the method works or how it differs from the *AlignControls* method in another component.
- Methods allow updating of several properties with a single call.

Chapter 39, "Creating methods," explains how to add methods to your components.

Events

An event is a special property that invokes code in response to input or other activity at runtime. Events give the application developer a way to attach specific blocks of code to specific runtime occurrences, such as mouse actions and keystrokes. The code that executes when an event occurs is called an *event handler*.

Events allow application developers to specify responses to different kinds of input without defining new components.

Chapter 38, "Creating events," explains how to implement standard events and how to define new ones.

Encapsulating graphics

Kylix simplifies graphics by encapsulating various graphics tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, such as a pen, a brush, and a font. A canvas is a painter that takes care of all the bookkeeping for you.

To draw on a form or other component, you access the component's *Canvas* property. *Canvas* is a property and it is also an object called *TCanvas*. *TCanvas* is a wrapper around a Qt painter that is accessible through the *Handle* property. You can use the handle to access low-level Qt graphics library functions.

If you want to customize a pen or brush, you set its color or style. When you finish, Kylix disposes of the resources. CLX also caches the resources to avoid recreating them if your application frequently uses the same kinds of resource.

You can use the canvas built into CLX components by descending from them. How graphics images work in the component depends on the canvas of the object from which your component descends. Graphics features are detailed in Chapter 40, "Using graphics in components."

Registering components

Before you can install your components in the IDE, you have to register them. Registration tells Kylix where to place the component on the Component palette. You can also customize the way Kylix stores your components in the form file. For information on registering a component, see Chapter 42, "Making components available at design time."

Creating a new component

You can create a new component two ways:

- Creating a component with the Component wizard
- Creating a component manually

You can use either of these methods to create a minimally functional component ready to install on the Component palette. After installing, you can add your new component to a form and test it at both design time and runtime. You can then add more features to the component, update the Component palette, and continue testing.

There are several basic steps that you perform whenever you create a new component. These steps are described below; other examples in this document assume that you know how to perform them.

1 Create a unit for the new component.
2 Derive your component from an existing component type.
3 Add properties, methods, and events.
4 Register your component with the IDE.
5 Create a bitmap for the component.
6 Create a package (a special shared object file) so that you can install your component in the IDE.

When you finish, the complete component includes the following files:

D

Table 35.2 Delphi component files

Type	Description of file
bpl<*packagename*>.so	Package
.dcp	Compiled package
.dcu and .dpu	Compiled unit
.dcr and .res	Component or project icon and version information

Table 35.3 C++ component files

Type	Description of file
bpl<*packagename*>.so	Package
.a	Static library file for package
.bpi	Borland import library for package
.o	Compiled unit
.dcr and .res	Component or project icon and version information

You can also create a bitmap to represent your new component. See "Creating a bitmap for a component" on page 35-15.

The chapters in the rest of Part IV explain all the aspects of building components and provide several complete examples of writing different kinds of components.

Creating a component with the Component wizard

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify:

- The class from which the new component is derived.
- The class name for the new component.
- The Component palette page where you want it to appear.
- The name of the unit in which the component is created.
- The search path where the unit is found.

- The name of the package in which you want to place the component.

The Component wizard performs the same tasks you would when creating a component manually, as described on page 35-11.

The Component wizard cannot add new components to an existing unit (in C++, consisting of a .cpp file and an associated header file). If you want to add new components, you must add them to existing units manually.

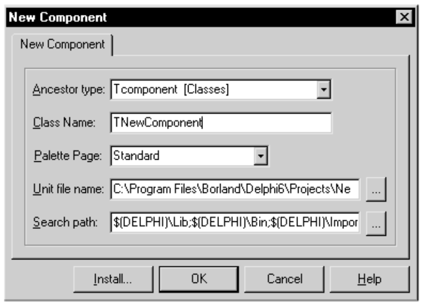
1 To start the Component wizard, choose one of these two methods:

- Choose Component | New Component.
- Choose File | New | Other and double-click Component.

2 Fill in the fields in the Component wizard:

- In the Ancestor Type field, specify the class from which you are deriving your new component.
- In the Class Name field, specify the name of your new component class.
- In the Palette Page field, specify the page on the Component palette on which you want the new component to be installed.
- In the Unit file name field, specify the name of the unit you want the component class declared in. If the unit is not on the search path, edit the search path in the Search Path field as necessary.

Figure 35.2 Component wizard



Ancestor type: Tcomponent [Classes]
Class Name: TNewComponent
Palette Page: Standard
Unit file name: C:\Program Files\Borland\Delphi6\Projects\Ne
Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Impor

3 After you fill in the fields in the Component wizard, either:

- Choose Install. To place the component in a new or existing package, click Component | Install and use the dialog box that appears to specify a package. See "Installing a component on the Component palette" on page 36-25.
- Choose OK. The IDE creates a new unit. In C++, the unit consists of a .cpp file and an associated header file.

Warning If you derive a component from a CLX class whose name begins with "custom" (such as *TCustomControl*), do not try to place the new component on a form until you have overridden any abstract methods in the original component. CLX cannot create instance objects of a class that has abstract properties or methods.

Delphi example

In Delphi, the Code editor displays a new unit containing the class declaration and the *Register* procedure, and adds a uses clause that includes all the standard Delphi units. For example:

```
unit NewComponent;

interface

uses
  SysUtils, Classes;

type
  TNewComponent = class(TComponent)
  private
  { Private declarations }
  protected
  { Protected declarations }
  public
  { Public declarations }
  published
  { Published declarations }
end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TNewComponent]);
end;

end.
```

C++ example

In C++, the Code editor displays the .cpp file. It contains a constructor for the component and the *Register* function that registers the component, informing the C++ IDE which component to add to the component library and on which page of the Component palette it should appear. The file also contains an include statement that specifies the header file that was created. For example:

```
#include <clx.h>
#pragma hdrstop
#include "NewComponent.h"
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//

static inline void ValidCtrCheck(TNewComponent *)
{
        new TNewComponent(NULL);
}
//---------------------------------------------------------------------------
```

```
__fastcall TNewComponent::TNewComponent(TComponent* Owner)
   : TComponent(Owner)
{
}
//---------------------------------------------------------------------------
namespace Newcomponent
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] = {__classid(TNewComponent)};
      RegisterComponents("Samples", classes, 0);
   }
}
```

To open the header file in the Code editor, place your cursor on the header file name, click your right mouse button to display the context menu, and choose Open File at Cursor on the menu.

The header file contains the new class declaration, including a constructor declaration, and several **#include** statements to support the new class. For example:

```
#ifndef NewComponentH
#define NewComponentH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Classes.hpp>
//---------------------------------------------------------------------------
class PACKAGE TNewComponent : public TComponent
{
private:
protected:
public:
   __fastcall TNewComponent(TComponent* Owner);
__published:
};
//---------------------------------------------------------------------------
#endif
```

Save the .cpp file with a meaningful name before proceeding.

Note To see the source code for your unit, choose View | Units or File | Open.

Creating a component manually

The easiest way to create a new component is to use the Component wizard. You can, however, perform the same steps manually.

To create a component manually, follow these steps:

1. Creating a unit file.
2. Deriving the component.
3. Declaring a new constructor in C++.
4. Registering the component.

Creating a unit file

A Delphi unit is a separately compiled module of code (.pas file). A C++ unit is a .cpp file and .h file combination that is compiled into an .o file. The IDE uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well.

When you create a component, you either create a new unit for the component or add the new component to an existing unit.

To create a new unit for a component:

1 Choose either:

- File | New | Unit.
- File | New | Other to display the New Items dialog box, select Unit, and choose OK.

The IDE creates a new unit file (Delphi) or .cpp and .h file (C++) and opens it in the Code editor. In C++, the .cpp file opens in the Code editor. To display the header file, click the .h tab.

2 Save the file with a meaningful name.

3 Derive the component class.

To open an existing unit:

1 Choose File | Open and select the source code unit to which you want to add your component.

Note When adding a component to an existing unit, make sure that the unit contains only component code. For example, adding component code to a unit that contains a form causes errors in the Component palette.

2 Derive the component class.

Deriving the component

Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. "Creating components" on page 35-2 describes which class to derive different kinds of components from.

A simple component class is a nonvisual component descended directly from *TComponent*.

To derive a simple component class, add the following class declarations.

D Delphi example

In Delphi, add a class declaration to the interface part of your component unit.

```
type
  TNewComponent = class(TComponent)
  end;
```

C++ example

In C++, add a class declaration to the header file.

```
class PACKAGE TNewComponent : public TComponent
{
};
```

In C++, the PACKAGE macro expands to a statement that allows classes to be imported and exported. You should also add the necessary include statements that specify the .hpp files needed by the new component. These are the most common include statements you need:

```
#include <SysUtils.hpp>
#include <QControls.hpp>
#include <Classes.hpp>
#include <QForms.hpp>
```

So far the new component does nothing different from *TComponent*. You have created a framework on which to build your new component.

Deriving classes is explained in more detail in "Defining new classes" on page 36-1. Also, review the CLX object hierarchy to determine the best object to use.

Declaring a new constructor in C++

Each new C++ component must have a constructor that overrides the constructor of the class from which it was derived. When you write the constructor for your new component, it must *always* call the inherited constructor.

Within the class declaration, declare a virtual constructor in the public section of the class. You can learn more about the public section in "Controlling access" on page 36-4.

For example:

```
class PACKAGE TNewComponent : public TComponent
{
public:
  virtual __fastcall TNewComponent(TComponent* AOwner);
};
```

In the .cpp file, implement the constructor:

```
__fastcall TNewComponent::TNewComponent(TComponent* AOwner): TComponent(AOwner)
{
}
```

Within the constructor, you add the code you want to execute when the component is created.

Registering the component

Registration is a simple process that tells the IDE which components to add to its component library, and on which pages of the Component palette they should appear. For a more detailed discussion of the registration process, see Chapter 42, "Making components available at design time."

Delphi example

To register a component in Delphi:

1 Add a procedure named *Register* to the interface part of the component's unit. *Register* takes no parameters, so the declaration is very simple:

```
procedure Register;
```

If you are adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you do not need to change the declaration.

2 Write the *Register* procedure in the implementation part of the unit, calling *RegisterComponents* for each component you want to register. *RegisterComponents* is a procedure that takes two parameters: the name of a Component palette page and a set of component types. If you are adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

C++ example

To register a component in C++:

1 Add a function named *Register* to the unit's .cpp file, placing it within a namespace. The namespace is the name of the file the component is in, minus the file extension, with all lowercase letters except the first letter.

For example, this code exists within a *Newcomp* namespace, whereas *Newcomp* is the name of the .cpp file:

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
  }
}
```

2 Within the *Register* function, declare an open array of type *TComponentClass* that holds the array of components you are registering. The syntax should look like this:

```
TComponentClass classes[1] = {__classid(TNewComponent)};
```

In this case, the array of classes contains just one component, but you can add all the components you want to register to the array.

3 Within the *Register* function, call *RegisterComponents* for each component you want to register.

RegisterComponents is a function that takes three parameters: the name of a Component palette page, the array of component classes, and the size – 1 of the component classes. If you're adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

You can register multiple components with just one *RegisterComponents* call if all components go on the same page on the Component palette.

To register a component named *TNewComponent* and place it on the Samples page of the Component palette:

1 Add the following *Register* procedure to the unit (Delphi) or .cpp file (C++) that declares *TNewComponent*:

Delphi example

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewComponent]);
end;
```

C++ example

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TNewComponent)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

This *Register* procedure places *TNewComponent* on the Samples page of the Component palette.

2 Once you register a component, you can compile (Delphi) or build (C++) it into a package and install it on the Component palette. See "Installing a component on the Component palette" on page 35-16.

See Chapter 42, "Making components available at design time".

Creating a bitmap for a component

Every component needs a bitmap to represent it on the Component palette. If you don't specify your own bitmap, the IDE uses a default bitmap. Because the palette bitmaps are needed only at design time, you don't compile them into the component's compilation unit. Instead, you supply them in a resource file with the same name as the unit (Delphi) or .cpp file (C++), but with the extension .dcr (dynamic component resource).

Each bitmap should be 24x24 pixels. There are a number of tools available on Linux for generating bitmap files. One approach is to use a tool such as Gimp to create the images as a .ppm file and then convert them to the .bmp format using ppmtobmp.

Note For each unit that contains components you want to install, supply a palette bitmap resource file, and within each palette bitmapresource file, supply a bitmap for each component you register. The bitmap image has the same name as the component.

For example, if you create a component named *TMyControl* in a unit named *TMyControl*, you need to create a resource file called TMYCONTROL. The resource names are not case-sensitive, but by convention they are usually in uppercase letters.

Keep the resource file in the same directory with the compiled files, so Kylix can find the bitmaps when it installs the components on the Component palette.

Installing a component on the Component palette

To install components in a package and onto the Component palette:

1. Choose Component | Install Component.

   The Install Component dialog box appears.
2. Install the new component into either an existing or a new package by selecting the applicable page.
3. Enter the name of the .pas (Delphi) or .cpp (C++) file containing the new component or choose Browse to find the unit.
4. Adjust the search path if the .pas (Delphi) or .cpp (C++) file for the new component is not in the default location shown.
5. Enter the name of the package into which to install the component or choose Browse to find the package.
6. If the component is installed into a new package, optionally enter a meaningful description of the package.
7. Choose OK to close the Install Component dialog box. This compiles/rebuilds the package and installs the component on the Component palette.

Note Newly installed components initially appear on the page of the Component palette that was specified by the component writer. You can move the components to a different page after they have been installed on the palette with the Component | Configure Palette dialog box.

For component writers who need to distribute their components to users to install on the Component palette, see "Making source files available" on page 35-16Making source files available.

Making source files available

Component writers should make all source files used by a component should be located in the same directory. These files include source code files (.pas (Delphi) or .cpp (C++)) and additional project files (.xfm, .res, .rc, and .dcr). In C++, header files (.h and .hpp) should be located in the include directory (or in a location on the search path for the IDE or a project).

The process of adding a component results in the creation of a number of files. These files are automatically put in directories specified in the IDE environment options (choose the menu command Tools | Environment Options, and click the Library tab).

The library files are placed in the DCP output directory (Delphi) or BPI/LIB output directory (C++). If adding the component entails creating a new package (as opposed to installing it into an existing package), the package file is put in the BPL output directory (Delphi) and .bpi files in the BPI/LIB output directory (C++).

Testing uninstalled components

You can test the runtime behavior of a component before you install it on the Component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Component palette. For information on testing already installed components, see "Testing installed components" on page 35-20.

You test an uninstalled component by emulating the actions performed by the IDE when the component is selected from the palette and placed on a form.

To test an uninstalled component:

1. Create a new application or open an existing one.
2. Choose Project | Add to Project to add the component unit to your project.
3. Include the name of the component's unit (.h file in C++) to the form unit's **uses** clause (Delphi) or header file (C++).
4. Add a data member to the form to represent the component.

   This is one of the main differences between the way you add components and the way the IDE does it. You add the data member to the public part at the bottom of the form's class declaration. The IDE would add it above, in the part of the class declaration that it manages.

   Never add fields to the IDE-managed part of the form's class declaration. The items in that part of the class declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.
5. Construct the component in the form's *OnCreate* handler.

   When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You nearly always pass *Self* (Delphi) or **this** (C++) as the owner. In a method, *Self* (Delphi) or **this** (C++) is a reference to the class that contains the method. In this case, in the form's *OnCreate* handler, *Self* (Delphi) or **this** (C++) refers to the form.
6. Assign the *Parent* property.

   Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the component that visually contains the control; which is most often the form, but it might be a group box or panel. Normally, you'll set *Parent* to *Self* (Delphi) or **this** (C++), that is, the form. Always set *Parent* before setting other properties of the control.

In Delphi, if your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property (instead of the component's) to *Self*, you can cause an operating system problem.

7 Set any other component properties as desired.

In C++, testing your components without installing has the added benefit of generating compile-time errors that are seen only when the class is instantiated. For example, trying to create an instance of an abstract class yields an error directing you to the pure virtual that must be overloaded.

Delphi example

Suppose you want to test a new component of type *TMyControl* in a unit named *MyControl*. Create a new project, then follow the steps to end up with a form unit that looks like this:

```
unit Unit1;
interface

uses
  SysUtils, Classes, QGraphics, QControls, QForms, QDialogs,
  MyControl;                               { 1. Add NewTest to uses clause }

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);              { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    MyControl1: TMyControl1;                                      { 2. Add an object field }
  end;

var
  Form1: TForm1;

implementation
{$R *.xfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  MyControl1 := TMyControl.Create(Self);                     { 4. Construct the component }
  MyControl1.Parent := Self;          { 5. Set Parent property if component is a control }
  MyControl1.Left := 12;                                     { 6. Set other properties... )
  ⋮                                                                   ...continue as needed }
end;
end.
```

C++ example

Suppose you want to test a new component of class *TNewControl* in a unit named *NewCtrl*. Create a new project, then follow the steps to end up with a header file for the form that looks like this:

```
//---------------------------------------------------------------------------
#ifndef TestFormH
```

```
#define TestFormH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include "NewCtrl.h"
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:          // IDE-managed Components
private:              // User declarations
public:               // User declarations
    TNewControl* NewControl1;
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

In C++, the **#include** statement that includes the newctrl.h file assumes that the component resides in the directory of the current project or in a directory that is on the include path of the project.

This is the .cpp file of the form unit:

```
#include <clx.h>
#pragma hdrstop
#include "TestForm.h"
#include "NewCtrl.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
static inline TNewControl *ValidCtrCheck()
{
    return new TNewControl(NULL);
}
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    NewControl1 = new TNewControl(this);
    NewControl1->Parent = this;
    NewControl1->Left = 12;
}
//---------------------------------------------------------------------------
namespace Newctrl
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TNewControl)};
        RegisterComponents("Samples", classes, 0);
```

```
          }
      }
```

Testing installed components

You can test the design-time behavior of a component after you install it on the Component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Component palette. For information on testing components that have not yet been installed, see "Installing a component on the Component palette" on page 35-16.

Testing your components after installing allows you to debug the component that only generates design-time exceptions when dropped on a form.

Test an installed component using a second running instance of the IDE:

1 From the IDE menu select Project | Options | and on the Directories/Conditionals page, set the Debug Source Path to the component's source file.
2 Then select Tools | Debugger Options. On the Language Exceptions page, enable the exceptions you want to track.
3 Open the component source file and set breakpoints.
4 Select Run | Parameters and set the Host Application field to the name and location of the Kylix executable file.
5 In the Run Parameters dialog, click the Load button to start a second instance of Kylix.
6 Then drop the components to be tested on the form, which should break on your breakpoints in the source.

Chapter

36

Object-oriented programming for component writers

If you have written applications with CLX, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user.

When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in this chapter, you can create versatile, reusable components.

Before you start creating components, you should be familiar with these topics, which are related to object-oriented programming (OOP):

- Defining new classes
- Ancestors, descendants, and class hierarchies
- Controlling access
- Dispatching methods
- Abstract class members
- Classes and pointers

Defining new classes

The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes.

A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as an integer. Classes are usually more complex than simple data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks.

For example, it is quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of the class *TButton*, but by assigning different values to their *Caption* properties and different handlers to their *OnClick* events, you make the two instances behave differently.

Deriving new classes

There are two reasons to derive a new class:

- To change class defaults to avoid repetition
- To add new capabilities to a class

In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.

To change class defaults to avoid repetition

Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine (Delphi) or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default.

For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to recreate the dialog each time, it is also not necessary. You can design the dialog once, set its properties, and install a wrapper component associated with it onto the Component palette. By making the dialog into a reusable component, you not only eliminate a repetitive task, but you encourage standardization and reduce the likelihood of errors each time the dialog is recreated.

Chapter 43, "Modifying an existing component," shows an example of changing a component's default properties.

Note If you want to modify only the published properties of an existing component, or to save specific event handlers for a component or group of components, you may be able to accomplish this more easily by creating a *component template*.

To add new capabilities to a class

A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*.

Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do *not* want to include in yours, you should derive from that component's ancestor.

For example, if you want to add features to a list box, you could derive your component from *TListBox*. However, if you want to add new features but exclude some capabilities of the standard list box, you need to derive your component from *TCustomListBox*, the ancestor of *TListBox*. Then you can recreate (or make visible) only the list-box capabilities you want, and add your new features.

Chapter 45, "Customizing a grid," shows an example of customizing an abstract component class.

Declaring a new component class

In addition to standard components, CLX provides many abstract classes designed as bases for deriving new components. Table 35.1 on page 35-3 shows the classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component's unit file (Delphi) or header file (C++).

A finished component declaration usually includes property, event (Delphi) or data member (C++), and method declarations before the end (Delphi) or final brace (C++). But an empty declaration is also valid, and provides a starting point for the addition of component features.

The declaration of a simple graphical component follows.

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
public:
    virtual __fastcall TSampleShape(TComponent* Owner);
};
```

In C++, do not forget to include the PACKAGE macro (defined in Sysmac.h), which allows classes to be imported and exported.

Ancestors, descendants, and class hierarchies

Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them.

The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its immediate ancestor, and from the immediate ancestor of its immediate ancestor, and so forth. All of the classes from which a component inherits are called its *ancestors*; the component is a *descendant* of its ancestors.

Together, all the ancestor-descendant relationships in an application constitute a hierarchy of classes. Each generation in the hierarchy contains more than its ancestors, since a class inherits everything from its ancestors, then adds new properties and methods or redefines existing ones.

If you do not specify an immediate ancestor, CLX derives your component from the default ancestor, *TObject*. *TObject* is the ultimate ancestor of all classes in the object hierarchy.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

Controlling access

There are several levels of *access control*—also called *visibility*—on properties, methods, and data members. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components.

Table 36.1 shows the levels of visibility, from most restrictive to most accessible:

Table 36.1 Levels of visibility within an object

Visibility	Meaning	Used for
private	In Delphi, accessible only to code in the unit where the class is defined. In C++, accessible only to the class where it is defined.	Hiding implementation details.
protected	In Delphi, accessible to code in the unit(s) where the class and its descendants are defined. In C++, accessible to the class where it is defined and its descendants.	Defining the component writer's interface.
public	Accessible to all code.	Defining the runtime interface.
published	Accessible to all code and accessible from the Object Inspector. Saved in a form file.	Defining the design-time interface.

In Delphi, declare members as private if you want them to be available only within the class where they are defined; declare them as protected if you want them to be available only within that class and its descendants. Remember, though, that if a member is available anywhere within a unit file, it is available *everywhere* in that file. Thus, if you define two classes in the same unit, the classes will be able to access each other's private methods. And if you derive a class in a different unit from its ancestor, all the classes in the new unit will be able to access the ancestor's protected methods.

Hiding implementation details

Private parts of a class are mostly useful for hiding details of implementation from users of the class. Because users of the class cannot access the private parts, you can change the internal implementation of the class without affecting user code. How this is done varies slightly in Delphi and C++

In Delphi, declaring part of a class as private makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

In C++, declaring part of a class as private makes that part invisible to code outside the class unless the functions are friends of the class. If you do not specify any access control on a data member, method, or property, that part is private.

Delphi example

The following Delphi example shows how declaring a field as private hides it from application developers. The listing shows two CLX form units. Each form has a handler for its *OnCreate* event which assigns a value to a private field. The compiler allows assignment to the field only in the form where it is declared.

```
unit HideInfo;
interface

uses SysUtils, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;

type
  TSecretForm = class(TForm)                                           { declare new form }
    procedure FormCreate(Sender: TObject);
  private                                                          { declare private part }
    FSecretCode: Integer;                                       { declare a private field }
  end;

var
  SecretForm: TSecretForm;

implementation
{$R *.xfm}
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42;                                            { this compiles correctly }
end;
end.                                                                        { end of unit }

unit TestHide;                                               { this is the main form file }

interface
uses SysUtils, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls,
HideInfo;                                       { use the unit with TSecretForm }

type
  TTestForm = class(TForm)
```

```
    procedure FormCreate(Sender: TObject);
  end;
var
  TestForm: TTestForm;

implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;         { compiler stops with "Field identifier expected" }
end;
end.                                                                        { end of unit }
```

Although a program using the *HideInfo* unit can use objects of type *TSecretForm*, it cannot access the *FSecretCode* field in any of those objects.

C++ example

The following C++ example illustrates how declaring a data member as private prevents users from accessing information.

The first part is a form unit made up of a header file and a .cpp file that assigns a value to a private data member in the form's *OnCreate* event handler. Because the event handler is declared within the *TSecretForm* class, the unit compiles without error.

```
#ifndef HideInfoH
#define HideInfoH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSecretForm : public TForm
{
__published:    // IDE-managed Components
    void __fastcall FormCreate(TObject *Sender);
private:
    int FSecretCode;                                  // declare a private data member
public:         // User declarations
  __fastcall TSecretForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TSecretForm *SecretForm;
//---------------------------------------------------------------------------
#endif
```

This is the accompanying .cpp file:

```
#include <clx.h>
#pragma hdrstop
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.xfm"
TSecretForm *SecretForm;
```

```
//---------------------------------------------------------------------------
__fastcall TSecretForm::TSecretForm(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TSecretForm::FormCreate(TObject *Sender)
{
  FSecretCode = 42;                             // this compiles correctly
}
```

The second part of this example is another form unit that attempts to assign a value to the *FSecretCode* data member in the *SecretForm* form. This is the header file for the unit:

```
#ifndef TestHideH
#define TestHideH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TTestForm : public TForm
{
__published:    // IDE-managed Components
    void __fastcall FormCreate(TObject *Sender);
public:         // User declarations
    __fastcall TTestForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TTestForm *TestForm;
//---------------------------------------------------------------------------
#endif
```

This is the accompanying .cpp file. Because the *OnCreate* event handler attempts to assign a value to a data member private to the *SecretForm* form, the compilation fails with the error message `'TSecretForm::FSecretCode' is not accessible`.

```
#include <clx.h>
#pragma hdrstop
#include "testHide.h"
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.xfm"
TTestForm *TestForm;
//---------------------------------------------------------------------------
__fastcall TTestForm::TTestForm(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TTestForm::FormCreate(TObject *Sender)
{
  SecretForm->FSecretCode = 13;               //compiler stops here with error message
}
```

Although a program using the *HideInfo* unit can use classes of type *TSecretForm*, it cannot access the *FSecretCode* data member in any of those classes.

Defining the component writer's interface

Declaring part of a class as protected makes that part visible only to the class itself and its descendants (and in Delphi, to other classes that share their unit files).

You can use protected declarations to define a *component writer's interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers.

Note A common mistake is trying to access protected methods from an event handler. Event handlers are typically methods of the form, not the component that receives the event. As a result, they do not have access to the component's protected methods (unless the component is declared in the same unit as the form).

Defining the runtime interface

Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole.

Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.

Here is an example that shows two read-only properties declared as part of a component's runtime interface:

D Delphi example

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;                      { implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;             { properties are public }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
⋮
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

C++ example

```
class PACKAGE TSampleComponent : public TComponent
{
private:
    int FTempCelsius;                                  // implementation details are private
    int GetTempFahrenheit();
public:
    ⋮
    __property int TempCelsius = {read=FTempCelsius};          // properties are public
    __property int TempFahrenheit = {read=GetTempFahrenheit};
};
```

This is the *GetTempFahrenheit* method in the .cpp file:

```
int TSampleComponent::GetTempFahrenheit()
{
  return FTempCelsius * (9 / 5) + 32;
}
```

Defining the design-time interface

Declaring part of a class as published makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events.

Because they show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that an application developer might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment.

Here is an example of a published property called *Temperature*. Because it is published, it appears in the Object Inspector at design time.

Delphi example

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;                           { implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature;       { writable! }
  end;
```

C++ example

```
class PACKAGE TSampleComponent : public TComponent
{
private:
    int FTemperature;
  ⋮
__published:
    __property int Temperature = {read=FTemperature, write=FTemperature};
};
```

Dispatching methods

Dispatch is the term used to describe how your application determines which class method should be invoked when it encounters a class method call. When you write code that calls a class method, it looks like any other procedure or function call. But classes have different ways of dispatching methods.

These types of method dispatch include:

- Static (Delphi)
- Regular (not virtual) methods (C++)
- Virtual
- Dynamic (Delphi)

Static methods in Delphi

In Delphi, all methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at runtime, which takes somewhat longer.

A static method does not change when inherited by a descendant class. If you declare a class that includes a static method, then derive a new class from it, the derived class shares exactly the same method at the same address. This means that you cannot override static methods; a static method always does exactly the same thing no matter what class it is called in. If you declare a method in a derived class with the same name as a static method in the ancestor class, the new method simply replaces the inherited one in the derived class.

An example of static methods in Delphi

In the following Delphi code, the first component declares two static methods. The second declares two static methods with the same names that replace the methods inherited from the first component.

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;      { different from the inherited method, despite same declaration }
    function Flash(HowOften: Integer): Integer;                { this is also different }
  end;
```

Regular methods in C++

C++ class methods are regular (or nonvirtual) unless you specifically declare them as virtual, or unless they override a virtual method in a base class. The compiler can determine the exact address of a regular class member at compile time. This is known as compile-time binding.

A base class regular method is inherited by derived classes. In the following example, an object of type *Derived* can call the method *Regular()* as it were its own method. Declaring a method in a derived class with the same name and parameters as a regular method in the class's ancestor *replaces* the ancestor's method. In the following example, when *d->AnotherRegular()* is called, it is being dispatched to the *Derived* class replacement for *AnotherRegular()*.

```
class Base
{
public:
    void Regular();
    void AnotherRegular();
    virtual void Virtual();
};

class Derived : public Base
{
public:
    void AnotherRegular();           // replaces Base::AnotherRegular()
    void Virtual();                  // overrides Base::Virtual()
};

void FunctionOne()
{
  Derived *d;
  d = new Derived;
  d->Regular();                    // Calling Regular() as it were a member of Derived
                                   // The same as calling d->Base::Regular()
  d->AnotherRegular();             // Calling the redefined AnotherRegular(), ...
                                   // ... the replacement for Base::AnotherRegular()
  delete d;
}

void FunctionTwo(Base *b)
{
  b->Virtual();
  b->AnotherRegular();
}
```

Virtual methods

Unlike regular methods, which are bound at compile time, virtual methods are bound at runtime. That is, the address of the method is determined by the object on which the method is called. This makes it possible to redefine virtual methods in descendant classes. (In addition, it is still possible to call the method in the ancestor class.)

To declare a method virtual, add the virtual directive after the method declaration (Delphi) or preface the method declaration with the virtual keyword (C++). When the compiler encounters the virtual keyword, it creates an entry in the class's virtual method table (VMT). The VMT holds the addresses of all the virtual methods in a class and is used at runtime to determine the address of the method to call.

When you derive a new class from an existing class, the new class creates its own VMT. This includes the entries from its ancestor's VMT, as well as entries for any additional virtual methods declared in the new class. The descendant class can override any of the inherited methods.

To see how all this works, consider the previous C++ example. (Delphi works in a similar fashion.) Suppose you call *FunctionTwo()* with a pointer to a *Derived* object. When *FunctionTwo()* calls *Virtual()* on the *Derived* object, the *Derived* object notices that *Virtual()* is a virtual method and looks up its address in the VMT. Because the *Derived* class overrides the *Virtual()* method, it finds the address of the *Virtual()* method, which is then called. On the other hand, when *FunctionTwo()* calls *AnotherRegular()* on the *Derived* object, this results in a call to the *Base* object's *AnotherRegular()*. This is because *AnotherRegular()* is not virtual, so its address (in *Base*) was bound at compile time.

Overriding methods

Overriding a method means extending or refining an ancestor's method, rather than replacing it.

In Delphi, to override a method in a descendant class, add the directive override to the end of the method declaration. Overriding a method causes a compilation error if:

- The method does not exist in the ancestor class.
- The ancestor's method of that name is static.
- The declarations are not otherwise identical (number and type of arguments parameters differ).

In C++, to override a method in a descendant class, redeclare the method in the derived class, ensuring that the number and type of arguments are the same.

The following code example shows the declaration of two simple components.

Delphi example

The first component declares three methods, each with a different kind of dispatching. The second component, derived from the first, replaces the static method and overrides the virtual methods.

```
type
  TFirstComponent = class(TCustomControl)
    procedure Move;               { static method }
    procedure Flash; virtual;     { virtual method }
    procedure Beep; dynamic;      { dynamic virtual method }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;               { declares new method }
    procedure Flash; override;     { overrides inherited method }
    procedure Beep; override;      { overrides inherited method }
  end;
```

C++ example

The first component declares two methods, each with a different kind of dispatching. The second component, derived from the first, replaces the nonvirtual method and overrides the virtual method.

```
class PACKAGE TFirstComponent : public TComponent
{
public:
  void Move();                             // regular method
  virtual void Flash();                    // virtual method
};

class PACKAGE TSecondComponent : public TFirstComponent
{
public:
  void Move();               // declares new method "hiding" TFirstComponent::Move()
  void Flash();              // overrides virtual TFirstComponent::Flash in TFirstComponent
};
```

D Dynamic methods in Delphi

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a method is called frequently, or if its execution is time-critical, you should probably declare it as virtual rather than dynamic.

Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic methods are listed separately. The dynamic method list contains entries only for methods introduced or overridden by a particular class. (The virtual method table, in contrast, includes all of the object's virtual methods, both inherited and introduced.) Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, working backwards through the inheritance tree.

To make a method dynamic, add the dynamic directive after the method declaration.

Abstract class members

When a method is declared as abstract in an ancestor class, you must surface it (by redeclaring and implementing it) in any descendant component before you can use the new component in applications. CLX cannot create instances of a class that contains abstract members. For more information about surfacing inherited parts of classes, see Chapter 37, "Creating properties," and Chapter 39, "Creating methods."

Classes and pointers

Every class (and therefore every component) is really a pointer. In Delphi, the compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this.

The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.

Chapter

37

Creating properties

Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form Designer. Well-designed properties make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining properties
- Creating array properties
- Storing and loading properties

Why create properties?

From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties as if they were data members. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter (Delphi) or an argument to a method by reference (C++).)

Properties provide more power than simple data members because:

- Application developers can set properties at design time. Unlike methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct an object, the Object Inspector supplies the values. The Object Inspector also validates property assignments as soon as they are made.
- Properties can hide implementation details. For example, data stored internally in an encrypted form can appear unencrypted as the value of a property; although

the value is a simple number, the component may look up the value in a database or perform complex calculations to arrive at it. Properties let you attach complex effects to outwardly simple assignments; what looks like an assignment to a data member can be a call to a method which implements elaborate processing.

- Properties can be virtual. Hence, what looks like a single property to an application developer may be implemented differently in different components.

A simple example is the *Top* property of all controls. Assigning a new value to *Top* does not just change a stored value; it repositions and repaints the control. And the effects of setting a property need not be limited to an individual component; for example, setting the *Down* property of a speed button to true sets *Down* property of all other speed buttons in its group to false.

Types of properties

A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time.

Table 37.1 How properties appear in the Object Inspector

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear as numbers, characters, and strings. The application developer can edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) appear as editable strings. The developer can also cycle through the possible values by double-clicking the value column, and there is a drop-down list that shows all possible values.
Set	Properties of set types appear as sets. By double-clicking on the property, the developer can expand the set and treat each element as a Boolean value (true if it is included in the set).
Object	Properties that are themselves classes often have their own property editors, specified in the component's registration procedure. If the class held by a property has its own published properties, the Object Inspector lets the developer to expand the list (by double-clicking) to include these properties and edit them individually. Object properties must descend from *TPersistent*.
Interface	Properties that are interfaces can appear in the Object Inspector as long as the value is an interface that is implemented by a component (a descendant of *TComponent*). Interface properties often have their own property editors.
Array	Array properties must have their own property editors; the Object Inspector has no built-in support for editing them. You can specify a property editor when you register your components.

Publishing inherited properties

All components inherit properties from their ancestor classes. When you derive a new component from an existing one, your new component inherits all the properties of its immediate ancestor. If you derive from one of the abstract classes, many of the inherited properties are either protected or public, but not published.

To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as published. Redeclaring means adding a declaration for the inherited property to the declaration of the descendant class.

If you derive a component from *TWidgetControl*, for example, it inherits the protected *Brush* property. By redeclaring *Brush* in your new component, you can change the level of protection to either public or published.

The following code shows a redeclaration of *Brush* as published, making it available at design time.

Delphi example

```
type
  TSampleComponent = class(TWidgetControl)
  published
    property Brush;
  end;
```

C++ example

```
class PACKAGE TSampleComponent : public TWidgetControl
{
__published:
    __property Brush;
};
```

When you redeclare a property, you specify only the property name, not the type and other information described in “Defining properties”. You can also declare new default values and specify whether to store the property.

Redeclarations can make a property less restricted, but not more restricted. Thus you can make a protected property public, but you cannot hide a public property by redeclaring it as protected.

Defining properties

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include:

- Property declarations
- Internal data storage
- Direct access
- Access methods

- Default property values

Property declarations

A property is declared in the declaration of its component class. To declare a property, you specify three things:

- The name of the property.
- The type of the property.
- The methods used to read and write the value of the property. If no write method is declared, the property is read-only.

Properties declared in a published section of the component's class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a public section are available at runtime and can be read or set in program code.

Here is a typical declaration for a property called *Count*.

Delphi example

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;                      { used for internal storage }
    procedure SetCount (Value: Integer);  { write method }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

C++ example

```
class PACKAGE TYourComponent : public TComponent
{
private:
    int FCount;                                             // data member for storage
    int __fastcall GetCount();                              // read method
    void __fastcall SetCount( int ACount );                 // write method
public:
    __property int Count = {read=GetCount, write=SetCount};   // property declaration
  ⋮
};
```

Internal data storage

There are no restrictions on how you store the data for a property. In general, however, CLX components follow these conventions:

- Property data is stored in class data members.

- The data members used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage.
- Identifiers for these data members consist of the letter *F* followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a data member called *FWidth*.

The principle that underlies these conventions is that only the implementation methods for a property should access the data behind it. If a method or another property needs to change that data, it should do so through the property, not by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating derived components.

Direct access

The simplest way to make property data available is *direct access*. That is, the read and write parts of the property declaration specify that assigning or reading the property value goes directly to the internal-storage data member without calling an access method. Direct access is useful when you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing.

It is common to have direct access for the read part of a property declaration but use an access method for the write part. This allows the status of the component to be updated when the property value changes.

The following component-type declaration shows a property that uses direct access for both the read and the write parts.

Delphi example

```
type
  TSampleComponent = class(TComponent)
  private                                                { internal storage is private}
    FMyProperty: Boolean;                           { declare field to hold property value }
  published                                       { make property available at design time }
    property MyProperty: Boolean read FMyProperty write FMyProperty;
  end;
```

C++ example

```
class PACKAGE TSampleComponent : public TComponent
{
private:                                       // internal storage is private
    bool FReadOnly;                            // declare data member to hold value
  ⋮
__published:                                   // make property available at design time
    __property bool ReadOnly = {read=FReadOnly, write=FReadOnly};
};
```

Access methods

You can specify an access method instead of a data member in the read and write parts of a property declaration. Access methods should be protected, and are usually declared as virtual; this allows descendant components to override the property's implementation.

Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.

Here is a class that declares three properties using the index specifier, which allows all three properties to have the same read and write access methods:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
```

C++ example

```
  class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
  int __fastcall GetDateElement(int Index);    // note Index parameter
  void __fastcall SetDateElement(int Index, int Value);
public:
  __property int Day = {read=GetDateElement, write=SetDateElement, index=3, nodefault};
  __property int Month = {read=GetDateElement, write=SetDateElement, index=2, nodefault};
  __property int Year = {read=GetDateElement, write=SetDateElement, index=1, nodefault};
};
```

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, the code avoids duplication by sharing the read and write methods for all three properties. You need only one method to read a date element, and another to write the date element.

Here is the read method that obtains the date element:

Delphi example

```
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);            { break encoded date into elements }
  case Index of
```

```
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

C++ example

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &Aday); // break date into elements
  switch (Index)
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}
```

This is the write method that sets the appropriate date element:

Delphi example

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                           { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);   { get current date elements }
    case Index of                             { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);   { encode the modified date }
    Refresh;                                    { update the visible calendar }
  end;
end;
```

C++ example

```
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0) // all elements must be positive
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay); // get date elements
```

```
      switch (Index)
      {
        case 1: AYear = Value;  break;
        case 2: AMonth = Value; break;
        case 3: ADay = Value;   break;
        default: return;
      }
   }
   FDate = TDateTime(AYear, AMonth, ADay); // encode the modified date
   Refresh();// update the visible calendar
  }
```

The read method

The read method for a property is a function that takes no parameters (except as noted below) and returns a value of the same type as the property. By convention, the function's name is *Get* followed by the name of the property. For example, the read method for a property called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see "Creating array properties" on page 37-11), both of which pass their index values as parameters. Use index specifiers to create a single read method that is shared by several properties. For more information about index specifiers, see the *Delphi Language Guide*.

If you do not declare a read method, the property is write-only. Write-only properties are seldom used.

The write method

The write method for a property is a procedure that takes a single parameter (except as noted below) of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's internal storage.

The only exceptions to the single-parameter rule are for array properties and properties that use index specifiers, both of which pass their index values as a second parameter. Use index specifiers to create a single write method that is shared by several properties.

If you do not declare a write method, the property is read-only.

Write methods commonly test whether a new value differs from the current value before changing the property. For example, here is a simple write method for an integer property called *Count* that stores its current value in a data member called *FCount*.

Delphi example

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

C++ example

```
void __fastcall TMyComponent::SetCount( int Value )
{
  if ( Value != FCount )
  {
    FCount = Value;
    Update();
  }
```

Default property values

When you declare a property, you can specify a *default value* for it. CLX uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, CLX always stores the property.

To specify a default value for a property:

- In Delphi, append the default directive to the property's declaration (or redeclaration), followed by the default value. For example,

  ```
  property Cool Boolean read GetCool write SetCool default True;
  ```

- In C++, append an equal sign after the property name and a set of braces that holds the **default** keyword and the default value. For example,

  ```
  __property bool IsTrue = {default=true};
  ```

Note Declaring a default value does not set the property to that value. The component's constructor method should initialize property values when appropriate. However, since objects always initialize their data members to 0, it is not strictly necessary for the constructor to set integer properties to 0, string properties to null, or Boolean properties to false.

Specifying no default value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value:

- In Delphi, append the **nodefault** directive to the property's declaration. For example:

  ```
  property FavoriteFlavor string nodefault;
  ```

- In C++, append an equal sign after the property name and a set of braces that holds the nodefault keyword. For example:

```
__property int NewInteger = {nodefault};
```

When you declare a property for the first time, there is no need to include nodefault. The absence of a declared default value means that there is no default.

Here is the declaration of a component that includes a single Boolean property named *IsTrue* with a default value of true.

Delphi example

Below the declaration (in the implementation section of the unit) is the constructor that initializes the property.

```
type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
⋮
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);              { call the inherited constructor }
  FIsTrue := True;                       { set the default value }
end;
```

C++ example

The declaration includes the constructor that sets the default value.

```
class PACKAGE TSampleComponent : public TComponent
{
private:
     bool FIsTrue;
public:
    virtual __fastcall TSampleComponent( TComponent* Owner );
__published:
    __property bool IsTrue = {read=FIsTrue, write=FIsTrue, default=true};
};

__fastcall TSampleComponent::TSampleComponent ( TComponent* Owner )
  : TComponent ( Owner )
{
  FIsTrue = true;
}
```

Creating array properties

Some properties lend themselves to being indexed like arrays. For example, the *Lines* property of *TMemo* is an indexed list of the strings that make up the text of the memo; you can treat it as an array of strings. *Lines* provides natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties are declared like other properties, except that

- The declaration includes one or more indexes with specified types. The indexes can be of any type.
- The read and write parts of the property declaration, if specified, must be methods. They cannot be data members.

The read and write methods for an array property take additional parameters that correspond to the indexes. The parameters must be in the same order and of the same type as the indexes specified in the declaration.

There are a few important differences between array properties and arrays. Unlike the index of an array, the index of an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can reference only individual elements of an array property, not the entire range of the property.

The following example shows the declaration of a property that returns a string based on an integer index.

Delphi example

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
⋮
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

C++ example

```
class PACKAGE TDemoComponent : public TComponent
{
private:
    System::AnsiString __fastcall GetNumberSize(int Index);
public:
    __property System::AnsiString NumberSize[int Index] = {read=GetNumberSize};
  ⋮
};
```

This is the *GetNumberSize* method in the .cpp file:

```
System::AnsiString __fastcall TDemoComponent::GetNumberSize(int Index)
{
  System::AnsiString Result;
  switch (Index)
  {
    case 0:
      Result = "Zero";
      break;
    case 100:
      Result = "Medium";
      break;
    case 1000:
      Result = "Large";
      break;
    default: Result = "Unknown size";
  }
  return Result;
}
```

Creating properties for subcomponents

By default, when a property's value is another component, you assign a value to that property by adding an instance of the other component to the form or data module and then assigning that component as the value of the property. However, it is also possible for your component to create its own instance of the object that implements the property value. Such a dedicated component is called a subcomponent.

Subcomponents can be any persistent object (any descendant of *TPersistent*). Unlike separate components that happen to be assigned as the value of a property, the published properties of subcomponents are saved with the component that creates them. In order for this to work, however, the following conditions must be met:

- The *Owner* of the subcomponent must be the component that creates it and uses it as the value of a published property. For subcomponents that are descendants of *TComponent*, you can accomplish this by setting the *Owner* property of the subcomponent. For other subcomponents, you must override the *GetOwner* method of the persistent object so that it returns the creating component.
- If the subcomponent is a descendant of *TComponent*, it must indicate that it is a subcomponent by calling the *SetSubComponent* method. Typically, this call is made either by the owner when it creates the subcomponent or by the constructor of the subcomponent.

Typically, properties whose values are subcomponents are read-only. If you allow a property whose value is a subcomponent to be changed, the property setter must free the subcomponent when another component is assigned as the property value. In addition, the component often re-instantiates its subcomponent when the property is set to nil (Delphi) or NULL (C++). Otherwise, once the property is changed to another component, the subcomponent can never be restored at design time. The following example illustrates such a property setter for a property whose value is a *TTimer*:

Delphi example

```
procedure TDemoComponent.SetTimerProp(Value: TTimer);
begin
  if Value <> FTimer then
  begin
    if Value <> nil then
    begin
      if Assigned(FTimer) and (FTimer.Owner = self) then
        FTimer.Free;
      FTimer := Value;
      FTimer.FreeNotification(self);
    end
    else //nil value
    begin
      if Assigned(FTimer) and (FTimer.Owner <> self) then
      begin
        FTimer := TTimer.Create(self);
        FTimer.Name := 'Timer'; //optional bit, but makes result much nicer
        FTimer.SetSubComponent(True);
        FTimer.FreeNotification(self);
      end;
    end;
  end;
end;
```

C++ example

```
void __fastcall TDemoComponent::SetTimerProp(ExtCtrls::TTimer *Value)
{
  if (Value != FTimer)
  {
    if (Value)
    {
      if (FTimer && FTimer->Owner == this)
        delete FTimer;
      FTimer = Value;
      FTimer->FreeNotification(this);
    }
    else // NULL value
    {
      if (FTimer && FTimer->Owner != this)
      {
        FTimer = new ExtCtrls::TTimer(this);
        FTimer.SetSubComponent(true);
        FTimer->FreeNotification(this);
        }
      }
    }
  }
```

Note that the property setter above called the *FreeNotification* method of the component that is set as the property value. This call ensures that the component that is the value of the property sends a notification if it is about to be destroyed. It sends this notification by calling the *Notification* method. You handle this call by overriding the *Notification* method, as follows:

Delphi example

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = FTimer) then
    FTimer := nil;
end;
```

C++ example

```
void __fastcall TDemoComponent::Notification(Classes::TComponent *AComponent,
Classes::TOperation Operation)
{
  TComponent::Notification(AComponent, Operation); { call inherited method }
  if ((Operation == opRemove) && (AComponent == (TComponent *)FTimer))
    FTimer = NULL;
}
```

Creating properties for interfaces in Delphi

You can use an interface as the value of a published property, much as you can use an object. However, the mechanism by which your component receives notifications from the implementation of that interface differs. In the previous topic, the property setter called the *FreeNotification* method of the component that was assigned as the property value. This allowed the component to update itself when the component that was the value of the property was freed. When the value of the property is an interface, however, you don't have access to the component that implements that interface. As a result, you can't call its *FreeNotification* method.

To handle this situation, you can call your component's *ReferenceInterface* method:

```
procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;
```

Calling *ReferenceInterface* with a specified interface does the same thing as calling another component's *FreeNotification* method. Thus, after calling *ReferenceInterface* from the property setter, you can override the *Notification* method to handle the notifications from the implementor of the interface:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.IsImplementorOf(MyInftProp)) then
    MyIntfProp := nil;
end;
```

Note that the *Notification* code assigns **nil** to the *MyIntfProp* property, not to the private field (*FIntfField*). This ensures that *Notification* calls the property setter, which calls *ReferenceInterface* to remove the notification request that was established when the property value was set previously. All assignments to the interface property must be made through the property setter.

Storing and loading properties

The Form Designer stores forms and their components in form (.xfm) files. A form file stores the properties of a form and its components. When CLX developers add the components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into the IDE or executed as part of an application, the components must restore themselves from the form file.

Most of the time you will not need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

These are the aspects of property storage you need to understand:

- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading
- Storing and loading unpublished properties

Using the store-and-load mechanism

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

Specifying default values

CLX components save their property values only if those values differ from the defaults. If you do not specify otherwise, CLX assumes a property has no default value, meaning the component always stores the property, whatever its value.

In Delphi, to specify a default value for a property, add the default directive and the new default value to the end of the property declaration.

In C++, to specify a default value for a property:

1 Add an equal sign (=) after the property name.

2 After the equal sign, add braces({}).

3 Within the braces, type the default keyword, followed by another equal sign.

4 Specify the new default value. For example:

```
__property Alignment = {default=taCenter};
```

In either language, you can also specify a default value when redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.

Note Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's constructor assigns the necessary value. A property whose value is not set by a component's constructor assumes a zero value—that is, whatever value the property assumes when its storage memory is set to 0. Thus numeric values default to 0, Boolean values to false, pointers to nil (Delphi) or NULL (C++), and so on. If there is any doubt, assign a value in the constructor method.

The following code shows a component declaration that specifies a default value for the *Align* property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

Delphi example

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;       { override to set new default }
  published
    property Align default alBottom;                   { redeclare with new default value }
  end;
⋮
constructor TStatusBar.Create(AOwner: TComponent);
begin
```

```
  inherited Create(AOwner);                          { perform inherited initialization }
  Align := alBottom;                                { assign new default value for Align }
end;
```

C++ example

```
class PACKAGE TMyStatusBar : public TPanel
{
public:
  virtual __fastcall TMyStatusBar(TComponent* AOwner);
__published:
  __property Align = {default=alBottom};
};
```

The constructor of the *TMyStatusBar* component is in the .cpp file:

```
__fastcall TMyStatusBar::TMyStatusBar (TComponent* AOwner)
  : TPanel(AOwner)
{
   Align = alBottom;
}
```

Determining what to store

You can control whether the Form Designer stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose not to store a given property at all, or you can designate a function that determines dynamically whether to store the property.

D To control whether the Delphi IDE stores a property, add the stored directive to the property declaration, followed by true, false, or the name of a Boolean function.

To control whether the C++ IDE stores a property:

1 Add an equal sign (=) after the property name.

2 After the equal sign, add braces({}).

3 Within the braces, type the stored keyword, followed by true, false, or the name of a Boolean function.

The following code shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean function:

D Delphi example

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
  ⋮
  published
    property Important: Integer stored True;         { always stored }
    property Unimportant: Integer stored False;       { never stored }
    property Sometimes: Integer stored StoreIt;       { storage depends on function value }
  end;
```

C++ example

```
class PACKAGE TSampleComponent : public TComponent
{
protected:
  bool __fastcall StoreIt();
public:
   ⋮
__published:
  __property int Important = {stored=true};         // always stored
  __property int Unimportant = {stored=false};      // never stored
  __property int Sometimes = {stored=StoreIt};      // storage depends on function value
};
```

Initializing after loading

After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the *Loaded* method.

Note The first thing to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize your own component.

Storing and loading unpublished properties

By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that the Form Designer does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on automatic behavior to store and load the strings it represents and must use the following mechanism.

You can save unpublished properties by adding code that indicates how to load and save your property's value.

To write your own code to load and save properties, use the following steps:

1 Create methods to store and load the property value.

2 Override the *DefineProperties* method, passing those methods to a filer object.

Creating methods to store and load property values

To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices:

- Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Kylix's built-in capabilities for saving and loading simple types. If your property value is built out of types that Kylix knows how to save and load, use this approach.
- Create two methods of type *TStreamProc*, one to store and one to load your property's value. *TStreamProc* takes a stream as an argument, and you can use the stream's methods to write and read your property values.

For example, consider a property that represents a component that is created at runtime. Kylix knows how to write this value, but does not do so automatically because the component is not created in the form designer. Because the streaming system can already load and save components, you can use the first approach. The following methods load and store the dynamically created component that is the value of a property named *MyCompProperty*:

Delphi example

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
  if Reader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
```

C++ example

```
void __fastcall TSampleComponent::LoadCompProperty(TReader *Reader)
{
  if (Reader->ReadBoolean())
    MyCompProperty = Reader->ReadComponent(NULL);
}
void __fastcall TSampleComponent::StoreCompProperty(TWriter *Writer)
{
  if (MyCompProperty)
  {
    Writer->WriteBoolean(true);
    Writer->WriteComponent(MyCompProperty);
  }
  else
    Writer->WriteBoolean(false);
}
```

Overriding the DefineProperties method

Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Kylix calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead.

No matter which method you use to define the property, you pass it the methods that store and load your property value as well as a boolean value indicating whether the property value needs to be written. If the value can be inherited or has a default value, you do not need to write it.

For example, given the *LoadCompProperty* method of type *TReaderProc* and the *StoreCompProperty* method of type *TWriterProc*, you would override *DefineProperties* as follows:

Delphi example

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else if MyCompProperty = nil or
         TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name then
        Result := True
      else Result := False;
    end
    else { no inherited value -- check for default (nil) value }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;
```

C++ example

```
void __fastcall TSampleComponent::DefineProperties(TFiler *Filer)
{
  // before we do anything, let the base class define its properties.
  // Note that this example assumes that TSampleComponent derives directly from TComponent
  TComponent::DefineProperties(Filer);
  bool WriteValue;
  if (Filer->Ancestor) // check for inherited value
  {
    if ((TSampleComponent *)Filer->Ancestor)->MyCompProperty == NULL)
      WriteValue = (MyCompProperty != NULL);
    else if ((MyCompProperty == NULL) ||
             (((TSampleComponent *)Filer->Ancestor)->MyCompProperty->Name !=
                                                     MyCompProperty->Name))
      WriteValue = true;
    else WriteValue = false;
  }
  else // no inherited value, write property if not null
    WriteValue = (MyCompProperty != NULL);
  Filer->DefineProperty("MyCompProperty ",LoadCompProperty,StoreCompProperty, WriteValue);
end;
```

Chapter

38

Creating events

An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*.

Events for the most common user actions (such as mouse actions) are built into all the standard components, but you can also define new events. To create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

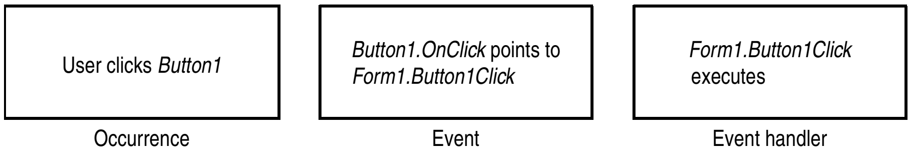
Events are implemented as properties, so you should already be familiar with the material in Chapter 37, “Creating properties,” before you attempt to create or change a component’s events.

What are events?

An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer (Delphi) or closure (C++) that points to a method in a specific class instance.

From the application developer’s perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. For example, a push button called *Button1* has an *OnClick* method. By default, when you assign a value to the *OnClick* event, the Form Designer generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick*. When a click event occurs in the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.

To write an event, you need to understand the following:


User clicks *Button1*
Button1.OnClick points to *Form1.Button1Click*
Form1.Button1Click executes
Occurrence
Event
Event handler

- Events are method pointers or closures.
- Events are properties.
- Event types are method pointer or closure types.
- Event-handler types are procedures in Delphi.
- Event handlers have a return type of void in C++.
- Event handlers are optional.

Events are method pointers or closures

CLX uses method pointers (Delphi) or closures (C++) to implement events. A method pointer or closure is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the method pointer or closure as a place holder: When your code detects that an event occurs, you call the method (if any) specified by the user for that event.

Method pointers or closures maintain a hidden pointer to a class instance. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a method of a specific class instance. That instance is usually the form that contains the component, but it need not be.

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
procedure Click; dynamic;
```

```
virtual void __fastcall Click(void);
```

In C++, dynamic methods map to virtual methods, but in CLX, which is written in Delphi, there is a distinction. Dynamic methods and virtual methods store the virtual method table differently, so that virtual methods are more time-efficient and dynamic methods more space-efficient.

The implementation of *Click* calls the user's click-event handler, if one exists. If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

Events are properties

Components use properties to implement their events. Unlike most other properties, events do not use methods to implement their read and write parts. Instead, event properties use a private data member of the same type as the property.

By convention, the data member's name is the name of the property preceded by the letter *F*. For example, the *OnClick* method's pointer (Delphi) or closure (C++) is stored in a data member called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

D

Delphi example

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;                { declare a data member to hold the method
pointer }
    ⋮
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

C++ example

```
class PACKAGE TControl : public TComponent
{
private:
    TNotifyEvent FOnClick;
  ⋮
protected:
    __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
  ⋮
};
```

To learn about *TNotifyEvent* and other event types, see the next section, "Event types are method pointer or closure types."

As with any other property, you can set or change the value of an event at runtime. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

Event types are method pointer or closure types

Because an event is a pointer to an event handler, the type of the event property must be a method pointer (Delphi) or closure (C++) type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class.

D To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

CLX defines method types or closures for all its standard events. When you create your own events, you can use an existing method pointer or closure if that is appropriate, or define one of your own.

Event-handler types are procedures in Delphi

In Delphi, although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers should be procedures.

Although an event handler cannot be a function, you can still get information from the application developer's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code to change the value.

An example of passing **var** parameters to an event handler is the *OnKeyPress* event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component may want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.

Event handlers have a return type of void in C++

Event handlers must have a return type of void only. Even though the handler can return only void, you can still get information back from the user's code by passing arguments by reference. When you do this, make sure you assign a valid value to the argument before calling the handler so you do not require the user's code to change the value.

An example of passing arguments by reference to an event handler is the key-pressed event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two arguments, one to indicate which object generated the event, and one to indicate which key was pressed:

```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender, Char &Key);
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component might want to change the character. One example might be to force all characters to uppercase in an edit control. In that case, the user could define the following handler for keystrokes:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char &Key)
{
  Key = UpCase(Key);
}
```

You can also use arguments passed by reference to let the user override the default handling.

Event handlers are optional

When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in "Calling the event" on page 38-10.)

Events happen almost constantly in a GUI application. Just moving the mouse pointer across a visual component sends numerous mouse-move messages, which the component translates into *OnMouseMove* events. In most cases, developers do not want to handle the mouse-move events, and this should not cause a problem. So the components you create should not require handlers for their events.

Moreover, application developers can write any code they want in an event handler. CLX components have events that are written in such a way as to minimize the chance of an event handler generating errors. Obviously, you cannot protect against logic errors in application code, but you can ensure that data structures are initialized before calling events so that application developers do not try to access invalid data.

Implementing the standard events

The controls that come with CLX inherit events for the most common occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often protected, meaning developers cannot attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- Identifying standard events
- Making events visible
- Changing the standard event handling

Identifying standard events

There are two categories of standard events: those defined for all controls and those defined only for the standard widget-based controls.

Standard events for all controls

The most basic events are defined in the class *TControl*. All controls, whether windowed, graphical, or custom, inherit these events. The following events are available in all controls:

OnClick	*OnDragDrop*	*OnEndDrag*	*OnMouseMove*
OnDblClick	*OnDragOver*	*OnMouseDown*	*OnMouseUp*

The standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names. For example, *OnClick* events call a method named *Click*, and *OnEndDrag* events call a method named *DoEndDrag*.

Standard events for widget-based controls

In addition to the events common to all controls, standard windowed controls (those that descend from *TWidgetControl* in CLX) have the following events:

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

Like the standard events in *TControl*, the widget-based control events have corresponding methods.

Making events visible

The declarations of the standard events in *TControl* and *TWidgetControl* are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either public or published.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.

For example, to create a component that surfaces the *OnClick* event at design time, add the following to the component's class declaration.

Delphi example

```
type
  TMyControl = class(TCustomControl)
  ⋮
  published
    property OnClick;
  end;
```

C++ example

```
class PACKAGE TMyControl : public TCustomControl
{
  ⋮
__published:
    __property OnClick;            // Makes OnClick available in the Object Inspector
};
```

Changing the standard event handling

If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component.

This is the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling, including the event for the application developer's code.

The order in which you call the methods is significant. As a rule, call the inherited method first, allowing the application developer's event-handler to execute before your customizations (and in some cases, to keep the customizations from executing). There may be times when you want to execute your code before calling the inherited method, however. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to them.

Suppose you are writing a component and you want to modify the way it responds to mouse clicks. Instead of assigning a handler to the *OnClick* event as a application developer would, you override the protected method *Click*:

Delphi example

```
procedure click override        { forward declaration }
⋮
procedure TMyControl.Click;
begin
  inherited Click;              { perform standard handling, including calling handler }
...                             { your customizations go here }
end;
```

C++ example

```
void __fastcall TMyControl::Click()
{
  TWidgetControl::Click();        // perform standard handling, including calling handler
  // your customizations go here
}
```

Defining your own events

Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it.

There are the issues you will need to consider when defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

Triggering the event

You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse. When notified of an event by the system, the component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event.

However, some events are less clearly tied to specific external occurrences. For example, a scroll bar has an *OnChange* event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the method that triggers the event.

See“Responding to system notifications using CLX” on page 41-1.

Two kinds of events

Events can be widget events, such as highlighting a menu item, or system events, such as working with timers or key presses.

Widget events are actions that are generated by user interaction with a widget. Widget events generate a signal that is passed onto the CLX component for processing. The CLX component has an associated event handler installed for the signal being passed to it. Examples of widget events are *OnChange* (the user changed text in an edit control), *OnHightlighted* (the user highlighted a menu item on a menu), and *OnReturnPressed* (the user pressed *Enter* in a memo control). These events are always tied to specific widgets and are defined within those widgets.

System events are events that the operating system generates. For example, the *OnTimer* event (the *TTimer* component issues one of these events whenever a predefined interval has elapsed), the *OnCreate* event (the component is being created), the *OnPaint* event (a component or widget needs to be redrawn), *OnKeyPress* event (a key was pressed on the keyboard), and so on. These are events that the application programmer must respond to if they are not handled as you want them to be.

Defining the handler type

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type *TNotifyEvent*, which carries only one parameter, the sender of the event. All a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it is not enough to know which event happened and what component it happened to. For example, if the event is a key-press event, it is likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters for additional information.

If your event was generated in response to a message, it is likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because event handlers do not return a value (in Delphi terms, procedures), the only way to pass information back from a handler is using a parameter that is passed by reference (in Delphi terms, a var parameter).Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass by reference the value of the key pressed in a parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to force typed characters to uppercase, for example.

Declaring the event

Once you have determined the type of your event handler, you are ready to declare the method pointer (Delphi) or closure (C++) and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of most events in CLX begin with "On." This is just a convention; the compiler does not enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method pointer (Delphi) or closure (C++) properties are assumed to be events and appear on the Events page.

Developers expect to find events in the alphabetical list of names starting with "On." Using other kinds of names is likely to confuse them.

Note The main exception to this rule is that many events that occur before and after some occurrence begin with "Before" and "After."

Calling the event

You should centralize calls to an event. That is, create a virtual method in your component that calls the application's event handler (if it assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from yours can customize event handling by overriding a single method, rather than searching through your code for places where you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid.
- Users can override default handling.

Empty handlers must be valid

You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.

An empty handler should produce the same result as no handler at all. So the code for calling an application's event handler should look like this:

Delphi example

```
if Assigned(OnClick) then OnClick(Self);
...  { perform default handling }
```

C++ example

```
if (OnClick)
  OnClick(this);
// perform default handling }
```

You should *never* have something like this:

Delphi example

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

C++ example

```
if (OnClick)
  OnClick(this);
else
  // perform default handling
```

Users can override default handling

For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

When handling key-press events, for example, application developers can suppress the component's default handling of the keystroke by returning a null character (#0 in Delphi or 0x0 in C++) as the Key parameter. The logic for supporting this looks like:

Delphi example

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ...  { perform default handling }
```

C++ example

```
if (OnKeyPress)
  OnKeyPress(this, &Key);
if (Key != NULL)
  //perform default handling
```

The actual code is a little different from this because it deals with system events, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets *Key* to a null character, the component skips the default handling.

Chapter

39

Creating methods

Component methods are no different from any other class's methods. That is, they are procedures built into the structure of a component class. Although there are essentially no restrictions on what you can do with the methods of a component, CLX does use some standards you should follow. These guidelines include:

- Avoiding dependencies.
- Naming methods.
- Protecting methods.
- Making methods virtual.
- Declaring methods.

In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that interacts well with the IDE and is accessible at design time.

Avoiding dependencies

At all times when writing components, minimize the preconditions imposed on the developer. To the greatest extent possible, developers should be able to do anything they want to a component, whenever they want to do it. There will be times when you cannot accommodate that, but your goal should be to come as close as possible.

This list gives you an idea of the kinds of dependencies to avoid:

- Methods that the user *must* call to use the component.
- Methods that must execute in a particular order.
- Methods that put the component into a state or mode where certain events or methods could be invalid.

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that second method so that if an application calls it when the component is in a bad state, the method corrects the state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause problems. A warning message, for example, is preferable to a system failure if the user does not accommodate your dependencies.

Naming methods

There are no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for application developers, however. Keep in mind that the nature of a component architecture dictates that many different kinds of people can use your components.

If you are accustomed to writing code that only you or a small group of programmers use, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive. Use meaningful verbs.

  A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.
- Function names should reflect the nature of what they return.

  Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.
- In C++, if a function return type is void, the function name should be active. Use active verbs in your function names. For example, *ReadFileNames* is much more helpful than *DoFiles*.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

Protecting methods

All parts of classes, including data members, methods, and properties, have a level of protection or “visibility,” as explained in “Controlling access” on page 36-4. Choosing the appropriate visibility for a method is simple.

Most methods you write in your components are public or protected. You rarely need to make a method private, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.

In C++, there is generally no reason for declaring a method (other than an event handler) as **__published**. Doing so looks to the end user exactly as if the method were public.

Methods that should be public

Any method that application developers need to call must be declared as public. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting the operating system in a state where it cannot respond to the user.

Note Constructors and destructors should always be public.

Methods that should be protected

Any implementation methods for the component should be protected so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them as protected.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method public, there is a chance that applications will call it before setting up the data. On the other hand, by making it protected, you ensure that applications cannot call it directly. You can then set up other, public methods that ensure that data setup occurs before calling the protected method.

Property-implementation methods should be declared as virtual protected methods. Methods that are so declared allow the application developers to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that developers do not accidentally call them, inadvertently modifying a property.

Abstract methods in Delphi

Sometimes a method is declared as abstract in a CLX component. Abstract methods usually occur in classes whose names begin with “custom,” such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes.

While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception.

The abstract directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces component writers to redeclare the abstract member in descendant classes before actual instances of the class can be created.

Making methods virtual

You make methods **virtual** when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

Declaring methods

Declaring a method in a component is the same as declaring any class method.

To declare a new method in a component, you do two things.

D In Delphi:

- Add the declaration to the component's object-type declaration.
- Implement the method in the implementation part of the component's unit.

In C++:

- Add the declaration to the component's class declaration in the component's header file.
- Write the code that implements the method in the .cpp file of the unit.

The following code shows a component that defines two new methods, one protected method and one public virtual method.

D Delphi example

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                             { declare protected static method }

  public
    function CalculateArea: Integer; virtual;           { declare public virtual method }
  end;
⋮
implementation
⋮
procedure TSampleComponent.MakeBigger;                          { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
```

```
end;

function TSampleComponent.CalculateArea: Integer;              { implement second method }
begin
  Result := Width * Height;
end;
```

C++ example

This code is the interface definition in the .h file:

```
class PACKAGE TSampleComponent : public TControl
{
protected:
    void __fastcall MakeBigger();
public:
    virtual int __fastcall CalculateArea();
  ⋮
};
```

This code is in the .cpp file of the unit that implements the methods:

```
void __fastcall TSampleComponent::MakeBigger()
{
  Height = Height + 5;
  Width = Width + 5;
}

int __fastcall TSampleComponent::CalculateArea()
{
  return Width * Height;
}
```

Chapter

40

Using graphics in components

Instead of forcing you to deal with graphics at a detailed level, CLX provides a simple yet complete interface: your component's *Canvas* property. The canvas has its own properties that represent the current pen, brush, and font.

The canvas manages all these resources for you, so you need not concern yourself with creating, selecting, and releasing things like pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Kylix manage graphic resources is that it can cache resources for later use, which can speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Kylix caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Kylix uses an existing one.

An example of this is an application that has dozens of forms open, with hundreds of controls. Each of these controls might have one or more *TFont* properties. Though this could result in hundreds or thousands of instances of *TFont* objects, most applications wind up using only two or three font handles.

Using the canvas

The canvas class encapsulates graphics controls at several levels, including high-level functions for drawing individual lines, shapes, and text; intermediate properties for manipulating the drawing capabilities of the canvas.

Table 40.1 summarizes the capabilities of the canvas.

Table 40.1 Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as *MoveTo*, *LineTo*, *Rectangle*, and *Ellipse*
	Displaying and measuring text	*TextOut*, *TextHeight*, *TextWidth*, and *TextRect* methods
	Filling areas	*FillRect* and *FloodFill* methods
Intermediate	Customizing text and graphics	*Pen*, *Brush*, and *Font* properties
	Manipulating pixels	*Pixels* property.
	Copying and merging images	*Draw*, *StretchDraw*, *BrushCopy*, and *CopyRect* methods; *CopyMode* property
Low	Calling Windows GDI functions	*Handle* property

For detailed information on canvas classes and their methods and properties, see online Help.

Working with pictures

Most of the graphics work you do in Kylix is limited to drawing directly on the canvases of components and forms. Kylix also provides for handling stand-alone graphic images, such as bitmaps, drawings (recorded drawing instructions), and icons.

There are three important aspects to working with pictures in Kylix:

- Using a picture, graphic, or canvas
- Loading and storing graphics

Using a picture, graphic, or canvas

There are three kinds of classes in Kylix that deal with graphics:

- A *canvas* represents a drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class.
- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or drawing. Kylix defines classes *TBitmap*, *TIcon*, and *TDrawing*, all descended from a generic *TGraphic*. You can also define your own graphic classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.
- A *picture* is a container for a graphic, meaning it could contain any of the graphic classes. That is, an item of type *TPicture* can contain a bitmap, an icon, a drawing, or a user-defined graphic type, and an application can access them all in the same way through the picture class. For example, the image control has a property

called *Picture*, of type *TPicture*, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture class always has a graphic, and a graphic might have a canvas. (The only standard graphics that has a canvas is *TBitmap* and *TDrawing*.) Normally, when dealing with a picture, you work only with the parts of the graphic class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

Loading and storing graphics

All pictures and graphics in Kylix can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

To load an image into a picture from a file, call the picture's *LoadFromFile* method. To save an image from a picture into a file, call the picture's *SaveToFile* method.

LoadFromFile and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

Note You can also load images from and save them to a Qt MIME source, or a stream object.

To load a bitmap into an image control's picture, for example, pass the name of a bitmap file to the picture's *LoadFromFile* method:

Delphi example

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('random.bmp');
end;
```

C++ example

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Image1->Picture->LoadFromFile("c:\\windows\\athena.bmp");
}
```

The picture recognizes .bmp as the standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Because the graphic is a bitmap, it loads the image from the file as a bitmap.

Off-screen bitmaps

When drawing complex graphic images, a common technique in graphics programming is to create an off-screen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an off-screen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap class in Kylix, which represents bitmapped images in resources and files, can also work as an off-screen image.

There are two main aspects to working with off-screen bitmaps:

- Creating and managing off-screen bitmaps.
- Copying bitmapped images.

Creating and managing off-screen bitmaps

When creating complex graphic images, avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas. The most common use of an offscreen bitmap is in the *Paint* method of a graphic control.

Delphi example

In Delphi, as with any temporary object, the bitmap should be protected with a try..finally block:

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;                              { override the Paint method }
  end;

procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                       { temporary variable for the off-screen bitmap }
begin
  Bitmap := TBitmap.Create;                                { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                                             { destroy the bitmap object }
  end;
end;
```

Copying bitmapped images

Kylix provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

Table 40.2 summarizes the image-copying methods in canvas objects.

Table 40.2 Image-copying methods

To create this effect	Call this method
Copy an entire graphic.	Draw
Copy and resize a graphic.	StretchDraw
Copy part of a canvas.	CopyRect
Copy a graphic repeatedly to tile an area.	TiledDraw

Responding to changes

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the class's *OnChange* event.

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the *OnChange* event of each, causing the component to refresh its image if either the pen or brush changes:

Delphi example

```
type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                              { construct the pen }
  FPen.OnChange := StyleChanged;                       { assign method to OnChange event }
  FBrush := TBrush.Create;                                        { construct the brush }
  FBrush.OnChange := StyleChanged;                     { assign method to OnChange event }
end;
```

```
procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                                    { erase and repaint the component }
end;
```

C++ example

Although the shape component is written in Delphi, the following is a C++ translation of the shape component with a new name, *TMyShape*.

This is the class declaration in the header file:

```
class PACKAGE TMyShape : public TGraphicControl
{
private:
protected:
public:
    virtual __fastcall TMyShape(TComponent* Owner);
__published:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall StyleChanged(TObject *Sender);
};
```

This is the code in the .cpp file:

```
__fastcall TMyShape::TMyShape(TComponent* Owner)
  : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FPen = new TPen;
  FPen->OnChange = StyleChanged;
  FBrush = new TBrush;
  FBrush->OnChange = StyleChanged;
}

void __fastcall TMyShape::StyleChanged(TObject *Sender)
{
  Invalidate();
}
```

Chapter

41

Handling system notifications

Components often need to respond to notifications from the underlying operating system. The operating system informs the application of occurrences such as what the user does with the mouse and keyboard. Some controls also generate notifications, such as the results from user actions such as selecting an item in a list box. CLX handles most of the common notifications already. It is possible, however, that you will need to write your own code for handling such notifications.

In CLX, notifications arrive in the form of *signals* and *system events* instead of Windows messages.

Responding to system notifications using CLX

When using Windows, the operating system sends notifications directly to your application and the controls it contains using Windows messages. This approach, however, is not appropriate for CLX, because CLX is a cross-platform library, and Windows messages are not used on Linux. Instead, CLX uses a platform-neutral way to respond to system notifications

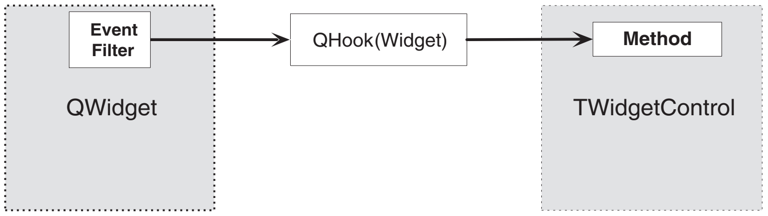
On CLX, the analog to Windows messages is a system of signals from the underlying widget layer. Whereas in WinCLX, Windows messages can originate either from the operating system or from the native Windows controls that WinCLX wraps, the widget layer that CLX uses makes a distinction between these two. If the notification originates from a widget, it is called a signal. If the notification originates with the operating system, it is called a system event. The widget layer communicates system events to your CLX components as a signal of type *event*.

Responding to signals

The underlying widget layer emits a variety of signals, each of which represents a different type of notification. These signals include system events (the event signal) as well as notifications that are specific to the widget that generates them. For example, all widgets generate a destroyed signal when the widget is freed, trackbar widgets generate a *valueChanged* signal, header controls generate a *sectionClicked* signal, and so on.

Each CLX component responds to signals from its underlying widget by assigning a method as the handler for the signal. It does this using a special hook object that is associated with the underlying widget. The hook object is a lightweight object that is really just a collection of method pointers, each method pointer specific to a particular signal. When a method of the CLX component has been assigned to the hook object as the handler for a specific signal, then every time the widget generates the specific signal, the method on the CLX component gets called. This is illustrated in Figure 41.1.

Figure 41.1 Signal routing



Note The methods for each hook object are declared in the Qt unit (in C++, check qt.hpp to see the methods available for a given hook object). The methods are flattened into global routines with names that reflect the hook object to which they belong. For example, all methods on the hook object associated with the application widget (QApplication) begin with 'QApplication_hook.' This flattening is necessary so that the Delphi CLX object can access the methods of the C++ hook object.

Assigning custom signal handlers

Many CLX controls already assign methods to handle signals from the underlying widget. Typically, these methods are private and not virtual. Thus, if you want to write your own method to respond to a signal, you must assign your own method to the hook object associated with your widget. To do this, override the *HookEvents* method.

Note If the signal to which you want to respond is a system event notification, you must not use an override of the HookEvents method. For details on how to respond to system events, see "Responding to system events" later.

In your override of the *HookEvents* method, declare a variable of type *TMethod*. Then for each method you want to assign to the hook object as a signal handler, do the following:

1. Initialize the variable of type *TMethod* to represent a method handler for the signal.
2. Assign this variable to the hook object. You can access the hook object using the *Hooks* property that your component inherits from *THandleComponent* or *TWidgetControl*.

In your override, always call the inherited *HookEvents* method so that the signal handlers that base classes assign are also hooked up.

The following code is the *HookEvents* method of *TTrackBar* (or in C++, a translation of the *HookEvents* method). It illustrates how to override the *HookEvents* method to add custom signal handlers.

Delphi example

```
procedure TTrackBar.HookEvents;
var
  Method: TMethod;
begin
  // initialize Method to represent a handler for the QSlider valueChanged signal
  // ValueChangedHook is a method of TTrackBar that responds to the signal.
  QSlider_valueChanged_Event(Method) := ValueChangedHook;
  // Assign Method to the hook object. Note that you can cast Hooks to the
  // type of hook object associated with the underlying widget.
  QSlider_hook_hook_valueChanged(QSlider_hookH(Hooks), Method);
  // Repeat the process for the sliderMoved event:
  QSlider_sliderMoved_Event(Method) := ValueChangedHook;
  QSlider_hook_hook_valueChanged(QSlider_hookH(Hooks), Method);
  // Call the inherited method so that inherited signal handlers are hooked up:
  inherited HookEvents;
end;
```

C++ example

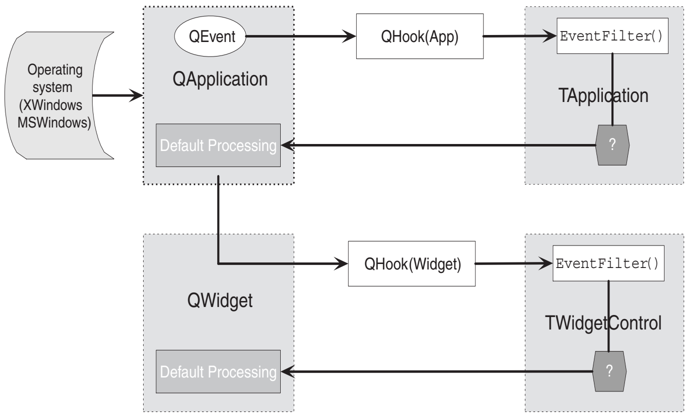
```
virtual void __fastcall TTrackBar::HookEvents(void)
{
  TMethod Method;
  // initialize Method to represent a handler for the QSlider valueChanged signal
  // ValueChangedHook is a method of TTrackBar that responds to the signal.
  QSlider_valueChanged_Event(Method) = @ValueChangedHook;
  // Assign Method to the hook object. Note that you can cast Hooks to the
  // type of hook object associated with the underlying widget.
  QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks), Method);
  // Repeat the process for the sliderMoved event:
  QSlider_sliderMoved_Event(Method) := @ValueChangedHook;
  QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks), Method);
  // Call the inherited method so that inherited signal handlers are hooked up:
  TWidgetControl::HookEvents();
}
```

Responding to system events

When the widget layer receives an event notification from the operating system, it generates a special event object (*QEvent* or one of its descendants) to represent the event. The event object contains read-only information about the event that occurred. The type of the event object indicates the type of event that occurred.

The widget layer notifies your CLX component of system events using a special signal of type event. It passes the *QEvent* object to the signal handler for the event. The processing of the event signal is a bit more complicated than processing other signals because it goes first to the application object. This means an application has two opportunities to respond to a system event: once at the application level (*TApplication*) and once at the level of the individual component (your *TWidgetControl* or *THandleComponent* descendant.) All of these classes (*TApplication*, *TWidgetControl*, and *THandleComponent*) already assign a signal handler for the event signal from the widget layer. That is, all system events are automatically directed to the *EventFilter* method, which plays a role similar to the *WndProc* method on WinCLX controls. The handling of system events is illustrated in Figure 41.2.

Figure 41.2 System event routing



EventFilter handles most of the commonly used system notifications, translating them into the events that are introduced by your component's base classes. Thus, for example, the *EventFilter* method of *TWidgetControl* responds to mouse events (*QMouseEvent*) by generating the *OnMouseDown*, *OnMouseMove*, and *OnMouseUp* events, to keyboard events (*QKeyEvent*) by generating the *OnKeyDown*, *OnKeyPress*, *OnKeyString*, and *OnKeyUp* events, and so on.

Commonly used events

The *EventFilter* method of *TWidgetControl* handles many of the common system notifications by calling on protected methods that are introduced in *TControl* or *TWidgetControl*. Most of these methods are virtual (or dynamic in Delphi), so that you can override them when writing your own components and implement your own responses to the system event. When overriding these methods, you do not need to worry about working with the event object or (in most cases) any of the other objects in the underlying widget layer.

When you want your CLX component to respond to system notifications, it is a good idea to first check whether there is a protected method that already responds to the notification. You can check the documentation for *TControl* or *TWidgetControl* (and any other base classes from which you derive your component) to see if there is a protected method that responds to the event in which you are interested. Table 41.1

lists many of the most commonly used protected methods from *TControl* and *TWidgetControl* that you can use.

Table 41.1 TWidgetControl protected methods for responding to system notifications

Method	Description
BeginAutoDrag	Called when the user clicks the left mouse button if the control has a *DragMode* of *dmAutomatic*.
Click	Called when the user releases the mouse button over the control.
DblClick	Called when the user double-clicks with the mouse over the control.
DoMouseWheel	Called when the user rotates the mouse wheel.
DragOver	Called when the user drags the mouse cursor over the control.
KeyDown	Called when the user presses a key while the control has focus.
KeyPress	Called after *KeyDown* if *KeyDown* does not handle the keystroke.
KeyString	Called when the user enters a keystroke when the system uses a multibyte character system.
KeyUp	Called when the user releases a key while the control has focus.
MouseDown	Called when the user clicks the mouse button over the control.
MouseMove	Called when the user moves the mouse cursor over the control.
MouseUp	Called when the user releases the mouse button over the control.
PaintRequest	Called when the system needs to repaint the control.
WidgetDestroyed	Called when a widget underlying a control is destroyed.

In the override, call the inherited method so that any default processes still take place.responds to signals

Note In addition to the methods that respond to system events, controls include a number of similar methods that originate with *TControl* or *TWidgetControl* to notify the control of various events. Although these do not respond to system events, they perform the same task as many Windows messages that are sent to WinCLX controls. Table 41.1 lists some of these methods.

Table 41.2 TWidgetControl protected methods for responding to events from controls

Method	Description
BoundsChanged	Called when the control is resized.
ColorChanged	Called when the color of the control changes.
CursorChanged	Called when the cursor changes shape. The mouse cursor assumes this shape when it's over this widget.
EnabledChanged	Called when an application changes the enabled state of a window or control.
FontChanged	Called when the collection of font resources changes.
PaletteChanged	Called when the widget's palette changes.
ShowHintChanged	Called when Help hints are displayed or hidden on a control.
StyleChanged	Called when the window or control's GUI styles change.
TabStopChanged	Called when the tab order on the form changes.

Table 41.2 TWidgetControl protected methods for responding to events from controls (continued)

Method	Description
TextChanged	Called when the control's text changes.
VisibleChanged	Called when a control is hidden or shown.

Overriding the EventFilter method

If you want to respond to an event notification and there is no protected method for that event that you can override, you can override the *EventFilter* method itself. In your override, check the type of the *Event* parameter of the *EventFilter* method, and perform your special processing when it represents the type of notification to which you want to respond. You can prevent further processing of the event notification by having your *EventFilter* method return true.

Note See the Qt documentation from TrollTech for details about the different types of *QEvent* objects.

The following code is the *EventFilter* method on *TCustomControl* (or in C++, a translation of the *EventFilter* method). It illustrates how to obtain the event type from the *QEvent* object when overriding *EventFilter*. Note that, although it is not shown here, you can cast the *QEvent* object to an appropriate specialized *QEvent* descendant (such as *QMouseEvent*) once you have identified the event type.

Delphi example

```
function TCustomControl.EventFilter(Sender: QObjectH; Event: QEventH): Boolean;
begin
  Result := inherited EventFilter(Sender, Event);
  case QEvent_type(Event) of
    QEventType_Resize,
    QEventType_FocusIn,
    QEventType_FocusOut:
      UpdateMask;
  end;
end;
```

C++ example

```
virtual bool __fastcall TCustomControl::EventFilter(Qt::QObjectH* Sender, Qt::QEventH*
Event)
{
  bool retval = TWidgetControl::EventFilter(Sender, Event);
  switch (QEvent_type(Event))
  {
    case QEventType_Resize:
    case QEventType_FocusIn:
    case QEventType_FocusOut:
      UpdateMask();
  }
  return retval;
}
```

Generating Qt events

Similar to the way a control can define and send custom Windows messages, you can make your VisualCLX control define and generate Qt system events. The first step is to define a unique ID for the event (similar to the way you must define a message ID when defining a custom Windows message):

```
const
  MyEvent_ID = Integer(QCLXEventType_ClxUser) + 50;
```

In the code where you want to generate the event, use the *QCustomEvent_create* function (declared in the Qt unit (the Qt.hpp file in C++) to create an event object with your new event ID. An optional second parameter lets you supply the event object with a data value that is a pointer to information you want to associate with the event:

Delphi example

```
var
  MyEvent: QCustomEventH;
begin
  MyEvent := QCustomEvent_create(MyEvent_ID, self);
```

C++ example

```
QCustomEventH *MyEvent = QCustomEvent_create(MyEvent_ID, this);
```

Once you have created the event object, you can post it by calling the *QApplication_postEvent* method:

```
QApplication_postEvent(Application.Handle, MyEvent);
```

```
QApplication_postEvent(Application->Handle, MyEvent);
```

For any component to respond to this notification, it need only override its *EventFilter* method, checking for an event type of *MyEvent_ID*. The *EventFilter* method can retrieve the data you supplied to the constructor by calling the *QCustomEvent_data* method that is declared in the Qt unit (the Qt.hpp file in C++).

Chapter

42

Making components available at design time

This chapter describes the steps for making the components you create available in the IDE. Making your components available at design time requires several steps:

- Registering components
- Adding property editors
- Adding property editors
- Adding component editors
- Compiling components into packages

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only steps that are always necessary are registration and compilation.

Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see "Installing component packages" on page 16-7.

Registering components

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once.

To register a component, add a *Register* procedure to the unit (or the .cpp file in C++). Within the *Register* procedure, you register the components and determine where to install them on the Component palette.

Note If you create your component by choosing Component | New Component in the IDE, the code required to register your component is added automatically.

The steps for manually registering a component are:

- Declaring the Register procedure
- Writing the Register procedure

Declaring the Register procedure

Registration involves writing a single procedure in the unit (or the .cpp file in C++), which must have the name *Register*. Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Component palette. If the unit (Delphi) or header and .cpp file (C++) contains several components, you can register them all in one step.

Delphi example

The *Register* procedure must appear in the interface part of the unit, and (unlike the rest of Delphi) its name is case-sensitive, and must be spelled with an uppercase R. The following code shows the outline of a simple unit that creates and registers new components.

```
unit MyBtns;
interface
type
  ⋮                                                  { declare your component types here }
procedure Register;                         { this must appear in the interface section }
implementation
  ⋮                                                 { component implementation goes here }
procedure Register;
begin
  ⋮                                                             { register the components }
end;
end.
```

C++ example

The *Register* procedure must exist within a namespace. The namespace is the name of the file the component is in with all lowercase letters except the first letter.

The following code shows how the *Register* procedure is implemented within a namespace. The namespace is named newcomp, where the file is named newcomp.cpp. The PACKAGE macro expands to a statement that allows classes to be imported and exported.

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
  }
}
```

Writing the Register procedure

Inside the *Register* procedure of a unit containing components, you must register each component you want to add to the Component palette. If the unit contains several components, you can register them at the same time.

To register a component, call the *RegisterComponents* procedure once for each page of the Component palette to which you want to add components. *RegisterComponents* involves three important things:

1 Specifying the components
2 Specifying the palette page
3 Using the RegisterComponents function

Specifying the components

Delphi example

In Delphi, within the Register procedure, pass the component names in an open array, which you can construct inside the call to *RegisterComponents*.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

You can also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);           { two on this page... }
  RegisterComponents('Assorted', [TThird]);                         { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);    { ...and one on the Standard page }
end;
```

C++ example

In C++, within the *Register* procedure, declare an open array of type *TComponentClass* that holds the array of components you are registering. The syntax should look like this:

```
TMetaClass classes[1] = {__classid(TNewComponent)};
```

In this case, the array of classes contains just one component, but you can add all the components you want to register to the array. For example, this code places two components in the array:

```
TMetaClass classes[2] =
                      {__classid(TNewComponent), __classid(TAnotherComponent)};
```

Another way to add a component to the array is to assign the component to the array in a separate statement. These statements add the same two components to the array as the previous example:

```
TMetaClass classes[2];
classes[0] = __classid(TNewComponent);
classes[1] = __classid(TAnotherComponent);
```

Specifying the palette page

The palette page name is a string (AnsiString in C++). If the name you give for the palette page does not already exist, the Forms Designer creates a new page with that name. The IDE stores the names of the standard pages in string-list resources so that international versions of the product can name the pages in their native languages. If you want to install a component on one of the standard pages, you should obtain the string for the page name by calling the *LoadStr* function, passing the constant representing the string resource for that page, such as *srSystem* for the System page.

Using the RegisterComponents function

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a procedure that takes several parameters: the name of a Component palette page, the array of component classes, and in C++ only, the index of the last entry in the array.

Delphi example

In the following Delphi example, set the Page parameter to the name of the page on the Component palette where the components should appear. If the named page already exists, the components are added to that page. If the named page does not exist, Delphi creates a new palette page with that name.

Call RegisterComponents from the implementation of the Register procedure in one of the units that defines the custom components. The units that define the components must then be compiled into a package and the package must be installed before the custom components are added to the Component palette.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);             {add to system page}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);                  { new page}
end;
```

C++ example

The following C++ example, the Register procedure in the newcomp.cpp file registers a component named *TMyComponent* and places it on a Component palette page called Miscellaneous:

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
      TMetaClass classes[1] = {__classid(TMyComponent)};
      RegisterComponents("Miscellaneous", classes, 0);
  }
}
```

Note that the third argument in the *RegisterComponents* call is 0, which is the index of the last entry in the classes array (the size of the array minus 1).

You can also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
namespace Mycomps
{
  void __fastcall PACKAGE Register()
  {
      // declares an array that holds two components
      TMetaClass classes1[2] = {__classid(TFirst), __classid(TSecond)};
      // adds a new palette page with the two components in the classes1 array
      RegisterComponents("Miscellaneous", classes1, 1);
      // declares a second array
      TMetaClass classes2[1];
      // assigns a component to be the first element in the array
      classes2[0]  = __classid(TThird);
      // adds the component in the classes2 array to the Samples page
      RegisterComponents("Samples", classes2, 0);
  }
}
```

In the above C++ example, the two arrays, classes1 and classes2, are declared. In the first *RegisterComponents* call the classes1 array has 2 entries, so the third argument is the index of the second entry, which is 1. In the second *RegisterComponents* call, the classes2 array has one element, so the third argument is 0.

Adding property editors

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing.

Writing a property editor requires these five steps:

1 Deriving a property-editor class
2 Editing the property as text
3 Editing the property as a whole
4 Specifying editor attributes
5 Registering the property editor

Deriving a property-editor class

CLX defines several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor classes described in Table 42.1. The classes in the DesignEditors unit can be used for cross-platform applications. Some of the property editor classes, however, supply specialized dialogs and so are specialized to CLX. These can be found in the CLXEditors units.

Note All that is absolutely necessary for a property editor is that it descend from *TBasePropertyEditor* and that it support the *IProperty* interface. *TPropertyEditor*, however, provides a default implementation of the *IProperty* interface.

The list in Table 42.1 is not complete. The CLXEditors unit also defines some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones that are the most useful for user-defined properties.

Table 42.1 Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from *TOrdinalProperty*.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	*Char*-type and subranges of *Char*, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings (AnsiStrings in C++).
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Classes. Displays the name of the class and allows expansion of the class's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

The following example shows the declaration of a simple property editor named *TMyPropertyEditor*:

D Delphi example

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
    end;
```

C++ example

```
  class PACKAGE TMyPropertyEditor : public TPropertyEditor
  {
    public:
      virtual bool __fastcall AllEqual(void);
      virtual System::AnsiString __fastcall GetValue(void);
      virtual void __fastcall SetValue(const System::AnsiString Value);
     __fastcall virtual ~TMyPropertyEditor(void) { }
     __fastcall TMyPropertyEditor(void) : Dsgnintf::TPropertyEditor() { }
  };
```

Editing the property as text

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. *TPropertyEditor* and its descendants provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called *GetValue* and *SetValue*. Your property editor also inherits methods for assigning and reading different sorts of values from *TPropertyEditor*, as shown in Table 42.2.

Table 42.2 Methods for reading and writing property values

Property type	Get method	Set method
Floating point	GetFloatValue	SetFloatValue
Method pointer (Delphi) or closure (C++) (event)	GetMethodValue	SetMethodValue
Ordinal type	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue

When you override a *GetValue* method, you call one of the Get methods, and when you override *SetValue*, you call one of the Set methods.

Displaying the property value

The property editor's *GetValue* method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, *GetValue* returns "unknown."

To provide a string representation of your property, override the property editor's *GetValue* method.

If the property is not a string value, *GetValue* must convert the value into a string representation.

Setting the property value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value.

To read string values into properties, override the property editor's *SetValue* method.

SetValue should convert the string and validate the value before calling one of the Set methods.

Delphi example

In the following Delphi example, here are the *GetValue* and *SetValue* methods for the *Integer* type of *TIntegerProperty* is an ordinal type, so *GetValue* calls *GetOrdValue* and converts the result to a string. *SetValue* converts the string to an integer, performs some range checking, and calls *SetOrdValue*.

```
function TIntegerProperty.GetValue: string;
begin
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then // unsigned
      Result := IntToStr(Cardinal(GetOrdValue))
    else
      Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
  procedure Error(const Args: array of const);
  begin
    raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
  end;
var
  L: Int64;
begin
  L := StrToInt64(Value);
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then
    begin   // unsigned compare and reporting needed
      if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
      // bump up to Int64 to get past the %d in the format string
        Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
    end
    else if (L < MinValue) or (L > MaxValue) then
      Error([MinValue, MaxValue]);
  SetOrdValue(L);
end;
```

The specifics of the particular Delphi examples here are less important than the principle: *GetValue* converts the value to a string; *SetValue* converts the string and validates the value before calling one of the "Set" methods.

Editing the property as a whole

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor class's *Edit* method.

Edit methods use the same Get and Set methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a Get method and a Set method. Because the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's *Edit* method.

Within your implementation of the *Edit* method, follow these steps:

1. Construct the editor you are using for the property.
2. Read the current value and assign it to the property using a Get method.
3. When the user selects a new value, assign that value to the property using a Set method.
4. Destroy the editor.

D

Delphi example

The *Color* properties found in most components use the standard color dialog box as a property editor. In Delphi, here is the *Edit* method from *TColorProperty*, which invokes the dialog box and uses the result:

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application);             { construct the editor }
  try
    ColorDialog.Color := GetOrdValue;                         { use the existing value }
    if ColorDialog.Execute then                           { if the user OKs the dialog... }
      SetOrdValue(ColorDialog.Color);                    { ...use the result to set value }
  finally
    ColorDialog.Free;                                            { destroy the editor }
  end;
end;
```

Specifying editor attributes

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's *GetAttributes* method.

GetAttributes is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

Table 42.3 Property-editor attribute flags

Flag	Related method	Meaning if included
paValueList	GetValues	The editor can give a list of enumerated values.
paSubProperties	GetProperties	The property has subproperties that can display.
paDialog	Edit	The editor can display a dialog box for editing the entire property.
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	SetValue	Updates the component after every change instead of waiting for approval of the value.
paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.
paFullWidthName	N/A	The value does not need to be displayed. The Object Inspector uses its full width for the property name instead.
paVolatileSubProperties	GetProperties	The Object Inspector refetches the values of all subproperties any time the property value changes.
paReference	GetComponent Value	The value is a reference to something else. When used in conjunction with paSubProperties the referenced object should be displayed as sub properties to this property.

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. *TColorProperty*'s *GetAttributes* method, therefore, includes several attributes in its return value:

D Delphi example

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

C++ example

```
virtual __fastcall TPropertyAttributes TColorProperty::GetAttributes()
{
  return TPropertyAttributes() << paMultiSelect << paDialog << paValueList << paRevertable;
}
```

Registering the property editor

Once you create a property editor, you need to register it with the IDE. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the *RegisterPropertyEditor* procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit.
  - In Delphi, this is always a call to the built-in function *TypeInfo*, such as `TypeInfo(TMyComponent)`.
  - In C++, specify the type information, such as `__typeinfo(TMyComponent)`
- The type of the component to which this editor applies. If this parameter is nil (Delphi) or NULL (C++), the editor applies to all properties of the given type.
- The name of the property. This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

Here is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

Delphi example

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

C++ example

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    RegisterPropertyEditor(__typeinfo(TComponent), 0L, "", __classid(TComponentProperty));
```

```
        RegisterPropertyEditor(__typeinfo(TComponentName), __classid(TComponent), "Name",
            __classid(TComponentNameProperty));
        RegisterPropertyEditor(__typeinfo(TMenuItem), __classid(TMenu), "",
            __classid(TMenuItemProperty));
        ⋮
    }
}
```

The three statements in this procedure cover the different uses of *RegisterPropertyEditor*:

- The first statement is the most typical. It registers the property editor *TComponentProperty* for all properties of type *TComponent* (or descendants of *TComponent* that do not have their own editors registered). In general, when you register a property editor, you have created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are nil (Delphi) or NULL (C++) and an empty string, respectively.
- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the *Name* property (of type *TComponentName*) of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type *TMenuItem* in components of type *TMenu*.

Property categories

In the IDE, the Object Inspector lets you selectively hide and display properties based on property categories. The properties of new custom components can be fit into this scheme by registering properties in categories. Do this at the same time you register the component by calling *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use *RegisterPropertyInCategory* to register a single property. Use *RegisterPropertiesInCategory* to register multiple properties in a single function call. These functions are defined in the unit DesignIntf.

Note that it is not mandatory that you register properties or that you register all of the properties of a custom component when some are registered. Any property not explicitly associated with a category is included in the *TMiscellaneousCategory* category. Such properties are displayed or hidden in the Object Inspector based on that default categorization.

In addition to these two functions for registering properties, there is an *IsPropertyInCategory* function. This function is useful for creating localization utilities, in which you must determine whether a property is registered in a given property category.

Registering one property at a time

Register one property at a time and associate it with a property category using the *RegisterPropertyInCategory* function. *RegisterPropertyInCategory* comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category.

The first variation lets you identify the property by the property's name. The line below registers a property related to visual display of the component, identifying the property by its name, "AutoSize".

```
RegisterPropertyInCategory('Visual', 'AutoSize');
```

```
RegisterPropertyInCategory("Visual", "AutoSize");
```

The second variation is much like the first, except that it limits the category to only those properties of the given name that appear on components of a given type. The example below registers (into the 'Help and Hints' category) a property named "HelpContext" of a component of the custom class *TMyButton*.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

```
RegisterPropertyInCategory("Help and Hints", __classid(TMyButton), "HelpContext");
```

The third variation identifies the property using its type rather than its name. The example below registers a property based on its type, Integer (Delphi) or a special class called *TArrangement* (C++).

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

```
RegisterPropertyInCategory('Visual', typeid(TArrangement));
```

The final variation uses both the property's type and its name to identify the property. The example below registers a property based on a combination of its type, *TBitmap*, and its name, "Pattern."

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

```
RegisterPropertyInCategory("Visual", typeid(TBitmap), "Pattern");
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Registering multiple properties at once

Register multiple properties at one time and associate them with a property category using the *RegisterPropertiesInCategory* function. *RegisterPropertiesInCategory* comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories.

The first variation lets you identify properties based on property name or type. The list is passed as an array of constants. In the example below, any property that either has the name "Text" or belongs to a class of type *TEdit* is registered in the category 'Localizable.'

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

```
RegisterPropertiesInCategory("Localizable", ARRAYOFCONST("Text", __typeinfo(TEdit)));
```

The second variation lets you limit the registered properties to those that belong to a specific component. The list of properties to register include only names, not types.

For example, the following code registers a number of properties into the ‘Help and Hints’ category for all components:

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint',
'ParentShowHint', 'ShowHint']);
```

```
RegisterPropertyInCategory("Help and Hints", __classid(TComponent),
ARRAYOFCONST("HelpContext", "Hint", "ParentShowHint"));
```

The third variation lets you limit the registered properties to those that have a specific type. As with the second variation, the list of properties to register can include only names:

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

```
RegisterPropertiesInCategory("Localizable", __typeinfo(TStrings), ARRAYOFCONST("Lines",
"Commands"));
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Specifying property categories

When you register properties in a category, you can use any string you want as the name of the category. If you use a string that has not been used before, the Object Inspector generates a new property category class with that name. You can also, however, register properties into one of the categories that are built-in. The built-in property categories are described in Table 42.4.:

Table 42.4 Property categories

Category	**Purpose**
Action	Properties related to runtime actions; the *Enabled* and *Hint* properties of *TEdit* are in this category.
Database	Properties related to database operations; the *DatabaseName* and *SQL* properties of *TQuery* are in this category.
Drag, Drop, and Docking	Properties related to drag-and-drop and docking operations; the *DragCursor* and *DragKind* properties of *TImage* are in this category.
Help and Hints	Properties related to using online Help or hints; the *HelpContext* and *Hint* properties of *TMemo* are in this category.
Layout	Properties related to the visual display of a control at design-time; the *Top* and *Left* properties of *TDBEdit* are in this category.
Legacy	Properties related to obsolete operations; the *Ctl3D* and *ParentCtl3D* properties of *TComboBox* are in this category.
Linkage	Properties related to associating or linking one component to another; the *DataSet* property of *TDataSource* is in this category.
Locale	Properties related to international locales; the *BiDiMode* and *ParentBiDiMode* properties of *TMainMenu* are in this category.
Localizable	Properties that may require modification in localized versions of an application. Many string properties (such as *Caption*) are in this category, as are properties that determine the size and position of controls.

Table 42.4 Property categories (continued)

Category	**Purpose**
Visual	Properties related to the visual display of a control at runtime; the *Align* and *Visible* properties of *TScrollBox* are in this category.
Input	Properties related to the input of data (need not be related to database operations); the *Enabled* and *ReadOnly* properties of *TEdit* are in this category.
Miscellaneous	Properties that do not fit a category or do not need to be categorized (and properties not explicitly registered to a specific category); the AllowAllUp and Name properties of TSpeedButton are in this category.

Using the IsPropertyInCategory function

An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the *IsPropertyInCategory* function are available, allowing for different criteria in determining whether a property is in a category.

The first variation lets you base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return true, the property must belong to a *TCustomEdit* descendant, have the name "Text," and be in the property category `'Localizable.'`

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

```
IsItThere = IsPropertyInCategory(“Localizable”, __classid(TCustomEdit), “Text”);
```

The second variation lets you base the comparison criteria on a combination of the class name of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return true, the property must be a *TCustomEdit* descendant, have the name "Text", and be in the property category `'Localizable'`.

```
IsItThere := IsPropertyInCategory('Localizable', ‘TCustomEdit’, 'Text');
```

```
IsItThere = IsPropertyInCategory(“Localizable”, “TCustomEdit”, “Text”);
```

Adding component editors

Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the Windows clipboard in custom formats.

If you do not give your components a component editor, Kylix uses the default component editor. The default component editor is implemented by the class *TDefaultEditor*. *TDefaultEditor* does not add any new items to a component's context menu. When the component is double-clicked, *TDefaultEditor* searches the properties of the component and generates (or navigates to) the first event handler it finds.

To add items to the context menu, change the behavior when the component is double-clicked, or add new clipboard formats, derive a new class from *TComponentEditor* and register its use with your component. In your overridden methods, you can use the *Component* property of *TComponentEditor* to access the component that is being edited.

Adding a custom component editor consists of the steps:

- Adding items to the context menu
- Changing the double-click behavior
- Adding clipboard formats
- Registering the component editor

Adding items to the context menu

When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu.

Adding items to the context menu requires the steps:

- Specifying menu items
- Implementing commands

Specifying menu items

Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command.

The following code overrides the *GetVerbCount* and *GetVerb* methods to add two commands to the context menu.

D **Delphi example**

```
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := '&DoThis ...';
    1: Result := 'Do&That';
  end;
end;
```

C++ example

```
int __fastcall TMyEditor::GetVerbCount(void)
{
  return 2;
}

System::AnsiString __fastcall TMyEditor::GetVerb(int Index)
{
  switch (Index)
  {
    case 0: return "&DoThis ..."; break;
    case 1: return "Do&That"; break;
  }
}
```

Note Be sure that your *GetVerb* method returns a value for every possible index indicated by *GetVerbCount*.

Implementing commands

When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor.

For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous example.

Delphi example

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
    0: begin
         MyDialog := TMySpecialDialog.Create(Application);       { instantiate the editor }
         if MySpecialDialog.Execute then;                { if the user OKs the dialog... }
           MyComponent.FThisProperty := MySpecialDialog.ReturnValue;   { ...use the value }
         MySpecialDialog.Free;                                       { destroy the editor }
       end;
    1: That;                                                       { call the That method }
  end;
end;
```

C++ example

```
void __fastcall TMyEditor::ExecuteVerb(int Index)
{
  switch (Index)
  {
    case 0:
      TMyDialog *MySpecialDialog = new TMyDialog();
      MySpecialDialog->Execute();
      ((TMyComponent *)Component)->ThisProperty = MySpecialDialog->ReturnValue;
      delete MySpecialDialog;
      break;
    case 1:
      That();  // call the "That" method
      break;
  }
}
```

Changing the double-click behavior

When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example, double-clicking the component executes the *DoThis* command.

While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if

- you are not adding any commands to the context menu.
- you want to display a dialog that combines several commands when the component is double-clicked.

Override the *Edit* method to specify a new behavior when the component is double-clicked. For example, the following *Edit* method brings up a font dialog when the user double-clicks the component:

Delphi example

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
```

C++ example

```
void __fastcall TMyEditor::Edit(void)
{
  TFontDialog *pFontDlg = new TFontDialog(NULL);
  pFontDlg->Execute();
  ((TMyComponent *)Component)->Font = pFontDlg->Font;
  delete pFontDlg;
}
```

Note If you want a double-click on the component to display the Code editor for an event handler, use *TDefaultEditor* as a base class for your component editor instead of

TComponentEditor. Then, instead of overriding the *Edit* method, override the protected *EditProperty*method instead. *EditProperty* scans through the event handlers of the component, and brings up the first one it finds. You can change this to look a particular event instead. For example:

Delphi example

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

C++ example

```
void __fastcall TMyEditor::EditProperty(TPropertyEditor* PropertyEditor,
                                         bool &Continue, bool &FreeEditor)
{
  if (PropertyEditor->ClassNameIs("TMethodProperty") &&
      CompareText(PropertyEditor->GetName, "OnSpecialEvent") == 0)
  {
    TDefaultEditor::EditProperty(PropertyEditor, Continue, FreeEditor);
  }
}
```

Adding clipboard formats

By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Kylix's internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method.

For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the IDE, but can be pasted into other applications.

Delphi example

```
procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
end;
```

C++ example

```
void __fastcall TMyComponentEditor::Copy(void)
{
  WORD AFormat;
  int AData;
  HPALETTE APalette;
  ((TImage *)Component)->Picture->SaveToClipboardFormat(AFormat, AData, APalette);
  TClipboard *pClip = Clipboard(); // don't clear the clipboard!
  pClip->SetAsHandle(AFormat, AData);
}
```

Registering the component editor

Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer.

To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

```
RegisterComponentEditor(__classid( TMyComponent), __classid(TMyEditor));
```

Place the call to *RegisterComponentEditor* in the *Register* procedure (or namespace in C++) where you register your component. For example, if a new component named *TMyComponent* and its component editor *TMyEditor* are both implemented in the same unit (Delphi) or newcomp.cpp (C++), the following code registers the component and its association with the component editor.

Delphi example

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

C++ example

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    TMetaClass classes[1] = {__classid(TMyComponent)};
    RegisterComponents("Miscellaneous", classes, 0);
    RegisterComponentEditor(classes[0], __classid(TMyEditor));
  }
}
```

Compiling components into packages

Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Chapter 16, "Working with packages and components."

To create and compile a package, see "Creating and editing packages" on page 16-8. Put the source-code units for your custom components in the package's Contains list. If your components depend on other packages, include those packages in the Requires list.

To install your components in the IDE, see "Installing component packages" on page 16-7.

Troubleshooting custom components in C++

A common problem when registering and installing custom components is that the component does not show up in the list of components after the package is successfully installed.

The most common causes for component not appearing in the list or on the palette:

- Missing PACKAGE modifier on the *Register* function
- Missing PACKAGE modifier on thme class
- Missing **#pragma** `package(smart_init)` in the C++ source file
- Register function is not found in a namespace with the same name as the source code module name.m
- Register is not being successfully exported. Use tdump on the bpl* (package) to look for the exported function:

  ```
  tdump -ebpl mypack.bpl  mypack.dmp
  ```

  In the exports section of the dump, you should see the *Register* function (within the namespace) being exported.

Chapter

43

Modifying an existing component

The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. The example in this chapter modifies the standard memo component; in Delphi, to create a memo that does not wrap words by default. In C++, to create a memo that has a yellow background. The other chapters in this part describe creating more complex components. The basic process is always the same, but more complex components require more steps in customizing the new class.

Modifying an existing component takes only two steps:

- Creating and registering the component.
- Modifying the component class.

Creating and registering the component

You create every component the same way: create and save a unit (Delphi) or .cpp and .h files (C++), derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 35-7.

D

Delphi example

The following Delphi example is a simple example that modifies the standard memo component to create a memo that does not wrap words by default.

The value of the memo component's *WordWrap* property is initialized to true. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default.

Note To modify published properties or save specific event handlers for an existing component, it is often easier to use a *component template* rather than create a new class.

For this Delphi example, follow the general procedure for creating a component, with these specifics:

1 Save the component's unit as *memos*.
2 Derive a new component type called *TWrapMemo*, descended from *TMemo*.
3 Register *TWrapMemo* on the Samples page of the Component palette.

   The resulting unit should look like this:

```
unit Memos;
interface
uses
  SysUtils, Types, Classes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.
```

   If you compile and install the new component now, it behaves exactly like its ancestor, *TMemo*. In the next section, you will make a simple change to your component.

C++ example

For this C++ example, follow the general procedure for creating a component, with these specifics:

1 Save the component's header file as *yelmemo.h* and the .cpp file as *yelmemo.cpp*.
2 Derive a new component class called *TYellowMemo*, descended from *TMemo*.
3 Register *TYellowMemo* on the Samples page of the Component palette.

   The resulting header file should look like this:

```
#ifndef YelMemoH
#define YelmemoH
//---------------------------------------------------------------------------
#include <classes.hpp>
#include <qcontrols.hpp>
#include <qStdCtrls.hpp>
#include <qforms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TYellowMemo : public TMemo
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

The accompanying .cpp file should look like this:

```
#include <clx.h>
#pragma hdrstop
#include "Yelmemo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TYellowMemo *)
{
  new TYellowMemo(NULL);
}
//---------------------------------------------------------------------------
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
        : TMemo(Owner)
{
}
//---------------------------------------------------------------------------
namespace Yelmemo
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TYellowMemo)};
    RegisterComponents("Common Controls", classes, 0);
  }
}
```

Note This example assumes you are not using the Component wizard to create this component, but that you are creating it manually. If you use the Component wizard, a constructor will automatically be added to *TYellowMemo*.

Modifying the component class

Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class:

- Overriding the constructor.
- Specifying the new default property value.

The constructor actually sets the value of the property. The default tells Kylix what values to store in the form (.xfm) file. Kylix stores only values that differ from the default, so it is important to perform both steps.

Overriding the constructor

When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time.

Note When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see "Overriding methods" on page 36-12.

Delphi example

For this Delphi example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to false. To achieve this, add the constructor override to the forward declaration, then write the new constructor in the **implementation** part of the unit:

```
type
  TWrapMemo = class(TMemo)
  public                                            { constructors are always public }
    constructor Create(AOwner: TComponent); override; { this syntax is always the same }
  end;
⋮
constructor TWrapMemo.Create(AOwner: TComponent);     { this goes after implementation }
begin
  inherited Create(AOwner);                         { ALWAYS do this first! }
  WordWrap := False;                                { set the new desired value }
end;
```

Now you can install the new component on the Component palette and add it to a form. Note that the *WordWrap* property is now initialized to *False*.

If you change an initial property value, you should also designate that value as the default. If you fail to match the value set by the constructor to the specified default value, Delphi cannot store and restore the proper value.

C++ example

For this C++ example, your new component needs to override the constructor inherited from *TMemo* to set the *Color* property to *clYellow*. To achieve this, add the declaration of the constructor override to the class declaration, then write the new constructor in the .cpp part file:

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner); // the constructor declaration
__published:
    __property Color;
};

__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
```

```
  : TMemo(Owner)                          // the constructor implementation first...
                                           // ...calls the constructor for TMemo
{
  Color = clYellow;                      // colors the component yellow
}
```

Note If you used the Component wizard to create the component, all you need to do is add `Color = clYellow;` to the existing constructor.

Now you can install the new component on the Component palette and add it to a form. Note that the *Color* property now defaults to *clYellow*.

Specifying the new default property value

When Kylix stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Chapter 42, "Making components available at design time."

Delphi example

In Delphi, to change the default value of a property:

1 Redeclare the property name.

2 Type the directive **default** and the new default value. You don't need to redeclare the entire property, just the name and the default value.

For the word-wrapping memo component, you redeclare the *WordWrap* property in the **published** part of the object declaration, with a default value of *False*:

```
type
  TWrapMemo = class(TMemo)
  ⋮
  published
    property WordWrap default False;
  end;
```

Specifying the default property value does not affect the workings of the component. You must still initialize the value in the component's constructor. Redeclaring the default ensures that Delphi knows when to write *WordWrap* to the form file.

C++ example

In C++, to change the default value of a property:

1 Redeclare the property name.

2 Place an equal sign (=) after the property name.

3 Type the **default** keyword, another equal sign, and the default value all within braces.

Note that you do not need to redeclare the entire property, just the name and the default value.

For the yellow memo component, you redeclare the *Color* property in a **__published** part of the class declaration, with a default value of *clYellow*:

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
};
```

Here is *TYellowMemo* again, but this time it has another published property, *WordWrap*, with a default value of **false**:

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
    __property WordWrap = {default=false};
};
```

Specifying the default property value does not affect the workings of the component at all. You must still explicitly set the default value in the component's constructor. The difference is in the inner workings of the application. For *TYellowMemo*, C++ no longer writes *WordWrap* to the form file if it is **false**, because you have told it that the constructor will set that value automatically. Here is the constructor:

```
__fastcall TYellowMemo::TYellowMemo(TComponent* AOwner) : TMemo(AOwner)
{
  Color = clYellow;
  WordWrap = false;
}
```

Chapter

44

Creating a graphic control

A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need its own window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic control presented in this chapter is *TShape*, the shape component on the Additional page of the Component palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. This chapter calls its shape component *TSampleShape* and shows you all the steps involved in creating the shape component:

- Creating and registering the component
- Publishing inherited properties
- Adding graphic capabilities

Creating and registering the component

You create every component the same way: create and save a unit (Delphi) or .cpp and .h files (C++), derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 35-7.

For this example, follow the general procedure for creating a component, with these specifics:

1 Save the component's unit (Delphi) or .cpp and .h files (C++):
   - In Delphi, name the unit *shapes*.
   - In C++, name the header file *shapes.h* and its .cpp file *shapes.cpp*.
2 Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.
3 Register *TSampleShape* on the Samples page of the Component palette.

Delphi example

In Delphi, the resulting unit should look like this:

```
unit Shapes;

interface

uses SysUtils, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;

type
  TSampleShape = class(TGraphicControl)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;

end.
```

C++ example

In C++, the resulting header file should look like this:

```
//---------------------------------------------------------------------------
#ifndef ShapesH
#define ShapesH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdsctrls.hpp>
#include <QForms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleShape : public TGraphicControl
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

The .cpp file should look like this:

```
//---------------------------------------------------------------------------
#include <clx.h>
#pragma hdrstop
#include "Shapes.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
```

```
//

static inline void ValidCtrCheck(TSampleShape *)
{
  new TSampleShape(NULL);
}
//---------------------------------------------------------------------------
__fastcall TSampleShape::TGraphicControl(TComponent* Owner)
        : TGraphicControl(Owner)
{
}
//---------------------------------------------------------------------------
namespace Shapes
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleShape)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

Publishing inherited properties

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop.

Publishing inherited properties and events is explained in "Publishing inherited properties" on page 37-3 and "Making events visible" on page 38-6. Both processes involve redeclaring just the name of the properties in the published part of the class declaration.

For the shape control, you can publish the three mouse events, the three drag-and-drop events, and the two drag-and-drop properties:

D Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;        { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;        { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;       { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:
__published:
    __property DragCursor ;
    __property DragMode ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnMouseDown ;
    __property OnMouseMove ;
    __property OnMouseUp ;
};
```

The sample shape control now makes mouse and drag-and-drop interactions available to its users.

Adding graphic capabilities

Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. You have two tasks to perform when creating a graphic control:

1 Determining what to draw.
2 Drawing the component image.

In addition, for the shape control example, you will add some properties that enable application developers to customize the appearance of the shape at design time.

Determining what to draw

A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a control.

In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it should draw.

To give your control a property that determines the shape it draws, add a property called *Shape*. This requires

1 Declaring the property type.
2 Declaring the property.
3 Writing the implementation method.

Creating properties is explained in more detail in Chapter 37, "Creating properties."

Declaring the property type

When you declare a property of a user-defined type, you must declare the type first, before the class that includes the property. The most common sort of user-defined type for properties is enumerated.

For the shape control, you need an enumerated type with an element for each kind of shape the control can draw.

Add the following type definition above the shape control class's declaration.

D

Delphi example

```
type
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
  TSampleShape = class(TGraphicControl) { this is already there }
```

C++ example

```
enum TSampleShapeType { sstRectangle, sstSquare, sstRoundRect, sstRoundSquare, sstEllipse,
sstCircle };

class PACKAGE TSampleShape : public TGraphicControl        // this is already there
```

You can now use this type to declare a new property in the class.

Declaring the property

When you declare a property, you usually need to declare a private data member to store the data for the property, then specify methods for reading and writing the property value. Often, you don't need to use a method to read the value, but can just point to the stored data instead.

For the shape control, you will declare a data member that holds the current shape, then declare a property that reads that data member and writes to it through a method call.

Add the following declarations to *TSampleShape*:

D

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType;  { field to hold property value }
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:
    TSampleShapeType FShape;
    void __fastcall SetShape(TSampleShapeType Value);
__published:
    __property TSampleShapeType Shape = {read=FShape, write=SetShape, nodefault};
};
```

Now all that remains is to add the implementation of *SetShape*.

Writing the implementation method

When the **read** or **write** part of a property definition uses a method instead of directly accessing the stored property data, you need to implement the method.

Delphi example

In Delphi, add the implementation of the *SetShape* method to the implementation part of the unit:

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then                           { ignore if this isn't a change }
  begin
    FShape := Value;                                { store the new value }
    Invalidate;                                     { force a repaint with the new shape }
  end;
end;
```

C++ example

Add the implementation of the *SetShape* method to the shapes.cpp file:

```
void __fastcall TSampleShape::SetShape(TSampleShapeType Value)
{
  if (FShape != Value)            // ignore if this isn’t a change
  {
     FShape = Value;              // store the new value
     Invalidate();                // force a repaint with the new shape
  }
}
```

Overriding the constructor and destructor

To change default property values and initialize owned classes for your component, you must override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.

Changing default property values

The default size of a graphic control is fairly small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Chapter 43, "Modifying an existing component."

In this example, the shape control sets its size to a square 65 pixels on each side.

Add the overridden constructor to the declaration of the component class:

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  public                                                { constructors are always public }
    constructor Create(AOwner: TComponent); override      { remember override directive }
  end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
public:
    virtual __fastcall TSampleShape(TComponent *Owner);
};
```

If you started creating the component with the Component wizard, this step will already be done for you.

1 Redeclare the *Height* and *Width* properties with their new default values:

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  published
    property Height default 65;
    property Width default 65;
  end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ⋮
__published:
    __property Height;
    __property Width;
```

2 Write the new constructor in the implementation part of the unit (Delphi) or the .cpp file (C++):

Delphi example

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);  { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
```

C++ example

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
}
```

If you used the Component wizard, all you need to do is add the new default values to the already existing constructor.

Publishing the pen and brush

By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them.

Managing owned classes requires:

1. Declaring the class data members.
2. Declaring the access properties.
3. Initializing owned classes.
4. Setting owned classes' properties.

Declaring the data members

Each class a component owns must have a data member declared for it in the component. The data member ensures that the component always has a pointer to the owned object so that it can destroy the class before destroying itself. In general, a component initializes owned objects in its constructor and destroys them in its destructor.

Data members for owned objects are nearly always declared as private. If applications (or other components) need access to the owned objects, you can declare published or public properties for this purpose.

Add data members for a pen and brush to the shape control:

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  private           { fields are nearly always private }
```

```
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush;  { a field for the brush object }
    ⋮
  end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:              // data members are always private
    TPen *FPen;        // a data member for the pen object
    TBrush *FBrush;    // a data member for the brush object
    ⋮
};
```

Declaring the access properties

You can provide access to the owned objects of a component by declaring properties of the type of the objects. That gives developers a way to access the objects at design time or runtime. Usually, the read part of the property just references the data member, but the write part calls a method that enables the component to react to changes in the owned object.

To the shape control, add properties that provide access to the pen and brush data members. You will also declare methods for reacting to changes to the pen or brush.

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  private                                          { these methods should be private }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published                                    { make these available at design time }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);                         { replace existing brush with parameter }
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);                             { replace existing pen with parameter }
end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
```

```
{
  ⋮
private:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall SetBrush(TBrush *Value);
    void __fastcall SetPen(TPen *Value);
    ⋮
__published:
    __property TBrush* Brush = {read=FBrush, write=SetBrush, nodefault};
    __property TPen* Pen = {read=FPen, write=SetPen, nodefault};
};
```

Then, write the *SetBrush* and *SetPen* methods in the .cpp file:

```
void __fastcall TSampleShape::SetBrush( TBrush* Value)
{
  FBrush->Assign(Value);
}
void __fastcall TSampleShape::SetPen( TPen* Value)
{
  FPen->Assign(Value);
}
```

To directly assign the contents of *Value* to *FBrush*:

```
FBrush := Value;
```

```
FBrush = Value;
```

would overwrite the internal pointer for *FBrush*, lose memory, and create a number of ownership problems.

Initializing owned classes

If you add classes to your component, the component's constructor must initialize them so that the user can interact with the objects at runtime. Similarly, the component's destructor must also destroy the owned objects before destroying the component itself.

Because you have added a pen and a brush to the shape control, you need to initialize them in the shape control's constructor and destroy them in the control's destructor:

1 Construct the pen and brush in the shape control constructor:

Delphi example

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                               { construct the pen }
  FBrush := TBrush.Create;                                         { construct the brush }
end;
```

C++ example

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();                          // construct the pen
  FPen = new TPen();                              // construct the brush
}
```

2 Add the overridden destructor to the declaration of the component class:

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  public                                             { destructors are always public}
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;                     { remember override directive }
  end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ⋮
public:                                           // destructors are always public
    virtual __fastcall TSampleShape(TComponent* Owner);
    __fastcall ~TSampleShape();                   // the destructor
    ⋮
};
```

3 Write the new destructor in the implementation part of the unit (Delphi) or in the .cpp file (C++):

Delphi example

```
destructor TSampleShape.Destroy;
begin
  FPen.Free;                                                  { destroy the pen object }
  FBrush.Free;                                              { destroy the brush object }
  inherited Destroy;                        { always call the inherited destructor, too }
end;
```

C++ example

```
__fastcall TSampleShape::~TSampleShape()
{
  delete FPen;                                    // delete the pen object
  delete FBrush;                                  // delete the brush object
}
```

Setting owned classes' properties

As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it.

Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method. With these changes, the component redraws to reflect changes to either the pen or the brush.

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
  ⋮
implementation
  ⋮
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                      { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                                  { construct the pen }
  FPen.OnChange := StyleChanged;                       { assign method to OnChange event }
  FBrush := TBrush.Create;                                            { construct the brush }
  FBrush.OnChange := StyleChanged;                     { assign method to OnChange event }
end;

procedure TSampleShape.StyleChanged(Sender: TObject);
begin
  Invalidate;                                    { erase and repaint the component }
end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ⋮
public:
    void __fastcall StyleChanged(TObject* Owner);
    ⋮
};
```

In the shapes.cpp file, assign the *StyleChanged* method to the *OnChange* events for the pen and brush classes in the *TSampleShape* constructor:

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();
  FBrush->OnChange = StyleChanged;
  FPen = new TPen();
  FPen->OnChange = StyleChanged;
}
```

Include the implementation of the *StyleChanged* method:

```
void __fastcall TSampleShape::StyleChanged( TObject* Sender)
{
  Invalidate();         // repaints the component
}
```

Drawing the component image

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a method called *Paint* that you override to paint the image you want on your control.

The *Paint* method for the shape control needs to do several things:

- Use the pen and brush selected by the user.
- Use the selected shape.
- Adjust coordinates so that squares and circles use the same width and height.

Overriding the *Paint* method requires two steps:

1. Add *Paint* to the component's declaration.
2. Write the *Paint* method in the implementation part of the unit (Delphi) or in the .cpp file (C++).

For the shape control, add the following declaration to the class declaration:

D

Delphi example

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  protected
    procedure Paint; override;
  ⋮
  end;
```

Then write the method in the implementation part of the unit:

```
procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                                           { copy the component's pen }
    Brush := FBrush;                                     { copy the component's brush }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);                 { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);                              { draw round shapes }
    end;
  end;
end;
```

C++ example

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ⋮
protected:
    virtual void __fastcall Paint();
    ⋮
};
```

Then write the method in the .cpp file:

```
void __fastcall TSampleShape::Paint()
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                           // copy the component's pen
  Canvas->Brush = FBrush;                       // copy the component's brush
  W=Width;                                      // use the component width
  H=Height;                                     // use the component height
  X=Y=0;                                        // save smallest for circles/squares
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)
  {
    case sstRectangle:                           // draw rectangles and squares
    case sstSquare:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundRect:                           // draw rounded rectangles and squares
    case sstRoundSquare:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                              // draw circles and ellipses
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

Paint is called whenever the control needs to update its image. Controls are painted when they first appear or when a window in front of them goes away. In addition, you can force repainting by calling *Invalidate,* as the *StyleChanged* method does.

Refining the shape drawing

The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image.

Here is a refined *Paint* method that adjusts for squares and ellipses:

Delphi example

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
  begin
    Pen := FPen;                                            { copy the component's pen }
    Brush := FBrush;                                      { copy the component's brush }
    W := Width;                                              { use the component width }
    H := Height;                                            { use the component height }
    if W < H then S := W else S := H;                { save smallest for circles/squares }

    case FShape of                                   { adjust height, width and position }
      sstRectangle, sstRoundRect, sstEllipse:
        begin
          X := 0;                                   { origin is top-left for these shapes }
          Y := 0;
        end;
      sstSquare, sstRoundSquare, sstCircle:
        begin
          X := (W - S) div 2;                              { center these horizontally... }
          Y := (H - S) div 2;                                           { ...and vertically }
          W := S;                                   { use shortest dimension for width... }
          H := S;                                                       { ...and for height }
        end;
    end;

    case FShape of
      sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);                    { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);            { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);                                  { draw round shapes }
    end;
  end;
end;
```

C++ example

```
void __fastcall TSampleShape::Paint(void)
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                        // copy the component's pen
  Canvas->Brush = FBrush;                     // copy the component’s brush
  W=Width;                                    // use the component width
  H=Height;                                   // use the component height
  X=Y=0;                                      // save smallest for circles/squares
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)                               // adjust height, width and position
  {
    case sstRectangle:
    case sstRoundRect:
    case sstEllipse:
      Y=X=0;                                  // origin is top-left for these shapes
      break;
    case sstSquare:
    case sstRoundSquare:
    case sstCircle:
      X= (W-S)/2;                             // center these horizontally
      Y= (H-S)/2;                             // and vertically
      break;
    default:
    break;
  }
  switch(FShape)
  {
    case sstSquare:                           // draw rectangles and squares
      W=H=S;                                  // use shortest dimension for width and height
    case sstRectangle:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundSquare:                      // draw rounded rectangles and squares
      W=H=S;
    case sstRoundRect:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                           // draw circles and ellipses
      W=H=S;
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

Chapter

45

Customizing a grid

CLX provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. In this chapter, you will see how to create a small one month calendar from the basic grid component, *TCustomGrid*.

Creating the calendar involves these tasks:

- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Navigating months and years
- Navigating days

Creating and registering the component

You create every component the same way: create and save a unit (Delphi) or .cpp and .h files (C++), derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 35-7.

For this example, follow the general procedure for creating a component, with these specifics:

1 Save the component's unit (Delphi) or .cpp and .h files (C++):

- In Delphi, name the unit *calsamp*.
- In C++, name the header file *calsamp.h* and its .cpp file *calsamp.cpp*.

2 Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.

3 Register *TSampleCalendar* on the Samples page of the Component palette.

The resulting unit (Delphi) or header file (C++) descending from *TCustomGrid* should look like the following:

Delphi example

```
unit CalSamp;

interface

uses
  SysUtils, Classes, QGraphics, QControls, QForms; QDialogs, QGrids;

type
  TSampleCalendar = class(TCustomGrid)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;

end.
```

In Delphi, if you install the calendar component now, you will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note In Delphi, while you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in “Filling in the cells” on page 45-7.

C++ example

```
#ifndef CalSampH
#define CalSampH
//---------------------------------------------------------------------------
#include <Sysutils.hpp>
#include <Classes.hpp>
#include <QControls.hpp>
#include <QGrids.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
```

```
#endif
```

The calsamp.cpp file should look like this:

```
#include <clx.h>
#pragma hdrstop
#include "CalSamp.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TSampleCalendar *ValidCtrCheck()
{
  return new TSampleCalendar(NULL);
}
//---------------------------------------------------------------------------
namespace Calsamp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleCalendar)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

In C++, if you used the Component wizard to create the component, the header file will also declare a new constructor, and the calsamp.cpp file will have the beginnings of a constructor. If the constructor is not there, you can add it later.

Publishing inherited properties

The abstract grid component, *TCustomGrid*, provides a large number of **protected** properties. You can choose which of those properties you want to make available to users of the calendar control.

To make inherited protected properties available to users of your components, redeclare the properties in the published part of your component's declaration.

For the calendar control, publish the following properties and events, as shown here:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align;  { publish properties }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;
    property OnClick;  { publish events }
    property OnDblClick;
    property OnDragDrop;
```

```
        property OnDragOver;
        property OnEndDrag;
        property OnKeyDown;
        property OnKeyPress;
        property OnKeyUp;
      end;
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
⋮
__published:
    __property Align ;                  // publish properties
    __property BorderStyle ;
    __property Color ;
    __property Font ;
    __property GridLineWidth ;
    __property ParentColor ;
    __property ParentFont ;
    __property OnClick ;                 // publish events
    __property OnDblClick ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnKeyDown ;
    __property OnKeyPress ;
    __property OnKeyUp ;
};
```

There are a number of other properties you could also publish, but which do not apply to a calendar, such as the *Options* property that would enable the user to choose which grid lines to draw.

If you install the modified calendar component to the Component palette and use it in an application, you will find many more properties and events available in the calendar, all fully functional. You can now start adding new capabilities of your own design.

Changing initial values

A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however.

To change the initial values of the component's properties, override the constructor to set the desired values. In Delphi, the constructor must be virtual.

Remember that you need to add the constructor to the public part of the component's object declaration, then write the new constructor in the implementation part of the component's unit (Delphi) or header file (C++).

Delphi example

In Delphi, the first statement in the new constructor should always be a call to the inherited constructor.

```
type
  TSampleCalendar = class(TCustomGrid
  public
    constructor Create(AOwner: TComponent); override;
  ⋮
  end;
⋮
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                { call inherited constructor }
  ColCount := 7;                                               { always seven days/week }
  RowCount := 7;                                   { always six weeks plus the headings }
  FixedCols := 0;                                                       { no row labels }
  FixedRows := 1;                                               { one row for day names }
  ScrollBars := ssNone;                                              { no need to scroll }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected];  {disable range selection}
end;
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
protected:
    virtual void __fastcall DrawCell(int ACol, int ARow, const Windows::TRect &Rect,
      TGridDrawState AState);
    ⋮
public:
    virtual __fastcall TSampleCalendar(TComponent *Owner);   // the added constructor
    ⋮
};
```

In the calsamp.cpp file, write the constructor code:

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  ColCount = 7;
  RowCount = 7;
  FixedCols = 0;
  FixedRows = 1;
  ScrollBars = ssNone;
  Options = (Options >> goRangeSelect) << goDrawFocusSelected;
}

void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Windows::TRect
  &ARect, TGridDrawState AState)
{
}
```

Note In C++, you will notice that a *DrawCell* method was also added to the class declaration, and in the .cpp file, the *DrawCell* method was started. This is not absolutely necessary now, but if you should attempt to test *TSampleCalendar* before overriding *DrawCell*, you would encounter a pure virtual function error. This is because *TCustomGrid* is an abstract class. Overriding *DrawCell* is discussed later in the "Filling in the cells" section.

The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

Resizing the cells

.Now when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

When a user or application changes the size of a window or control, it is are automatically notified by a call to the protected *BoundsChanged* method. Your CLX component can respond to this notification by altering the size of the cells so they all fit inside the boundaries of the control.

In this case, the calendar control needs to override *BoundsChanged* method so that it calculates the proper cell size to allow all cells to be visible in the new size:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
  ⋮
  end;
⋮
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer;                                   { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                 { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7;            { and cell height }
  inherited; {now call the inherited method }
end;
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
⋮
protected:
    void __fastcall BoundsChanged(void);
};
```

Here is the code for the method in the calsamp.cpp file:

```
void __fastcall TSampleCalendar::BoundsChanged(void)
{
  int GridLines;                                  // temporary local variable
  GridLines = 6 * GridLineWidth;                  // calculated combined size of all lines
  DefaultColWidth = (Width - GridLines) / 7;    // set new default cell width
  DefaultRowHeight = (Height - GridLines) / 7;  // and cell height
  TCustomGrid::BoundsChanged(); // now call the inherited method
}
```

Filling in the cells

A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*.

To fill in the contents of grid cells, override the *DrawCell* method.

The easiest part to fill in is the heading cells in the fixed row. The runtime library contains an array with short day names, so for the calendar, use the appropriate one for each column:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
      override;
  end;
⋮
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);    { use RTL strings }
end;
```

C++ example

Here is the code for the *DrawCell* method:

```
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Types::TRect &ARect,
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0) TheText = ShortDayNames[ACol + 1];
  else
  {
    TheText = "";
    TempDay = DayNum(ACol, ARow);                   // DayNum is defined later
```

```
      if (TempDay != -1) TheText = IntToStr(TempDay);
    }
    Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left
     - Canvas->TextWidth(TheText)) / 2,
    ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
  }
```

Tracking the date

For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. CLX stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use.

You can therefore store the date in encoded form, providing runtime access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time.

Tracking the date in the calendar consists of the processes:

- Storing the internal date
- Accessing the day, month, and year
- Generating the day numbers
- Selecting the current day

Storing the internal date

To store the date for the calendar, you need a private data member to hold the date and a runtime-only property that provides access to that date.

Adding the internal date to the calendar requires three steps:

1 Declare a private data member to hold the date:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  ⋮
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TDateTime FDate;
    ⋮
};
```

2 Initialize the date data member in the constructor:

Delphi example

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);        { this is already here }
  ⋮                                { other initializations here }
  FDate := Date;                   { get current date from RTL }
end;
```

C++ example

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  ⋮
  FDate = FDate.CurrentDate();
}
```

3 Declare a runtime property to allow access to the encoded date:

You'll need a method for setting the date, because setting the date requires updating the onscreen image of the control:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  ⋮
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;               { set new date value }
  Refresh;                      { update the onscreen image }
end;
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
public:
    __property TDateTime CalendarDate = {read=FDate, write=SetCalendarDate, nodefault};
    ⋮
};
```

In C++, declare *SetCalendarDate* in *TSampleCalendar*:

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    void __fastcall SetCalendarDate(TDateTime Value);
    ⋮
};
```

This is the *SetCalendarDate* method:

```
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                      // Set the new date value
  Refresh();                          // Update the onscreen image
}
```

Accessing the day, month, and year

An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties.

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all three properties.

To provide design-time access to the day, month, and year, you do the following:

1 Declare the three properties, assigning each a unique index number:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  ⋮
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
  ⋮
public:
    __property int Day = {read=GetDateElement, write=SetDateElement, index=3,
      nodefault};
    __property int Month = {read=GetDateElement, write=SetDateElement, index=2,
      nodefault};
    __property int Year = {read=GetDateElement, write=SetDateElement, index=1,
      nodefault};
};
```

2 Declare and write the implementation methods, setting different elements for each index value:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;         { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);             { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                       { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                 { get current date elements }
    case Index of                                     { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);                 { encode the modified date }
    Refresh;                                                { update the visible calendar }
  end;
end;
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int __fastcall GetDateElement(int Index);           // note the Index parameter
    void __fastcall SetDateElement(int Index, int Value);
    ⋮
};
```

In C++, the *GetDateElement* and *SetDateElement* methods are:

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &ADay);            // break encoded date into elements
  switch (Index)
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0)                                      // all elements must be positive
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);     // get current date elements
    switch (Index)
    {
      case 1: AYear = Value;     break;
      case 2: AMonth = Value;    break;
      case 3: ADay = Value;      break;
      default: return;
    }
  }
  FDate = TDateTime(AYear, AMonth, ADay);         // encode the modified date
  Refresh();                                      // update the visible calendar
}
```

Now you can set the calendar's day, month, and year at design time using the Object Inspector or at runtime using code. Of course, you have not yet added the code to paint the dates into the cells, but now you have the needed data.

Generating the day numbers

Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit (Delphi) or header file (C++) to get the number of days in the month.

Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go. The calculation is based on the day of the week the month starts on.

Because you will need the month-offset number for each cell you fill in, the best practice is to calculate it once when you change the month or year, then refer to it each time. You can store the value in a class data member, then update that data member each time the date changes.

To fill in the days in the proper cells, you do the following:

1 Add a month-offset data member to the class and a method that updates the data member value:

D

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;                                    { storage for the offset }
  ⋮
  protected
    procedure UpdateCalendar; virtual;                    { property for offset access }
  end;
⋮
procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                           { date of the first day of the month }
begin
  if FDate <> 0 then                          { only calculate offset if date is valid }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                     { get elements of date }
    FirstDate := EncodeDate(AYear, AMonth, 1);                     { date of the first }
    FMonthOffset := 2 - DayOfWeek(FirstDate);       { generate the offset into the grid }
  end;
  Refresh;                                                { always repaint the control }
end;
```

C++

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int FMonthOffset;                         // storage for the offset
    ⋮
protected:
    virtual void __fastcall UpdateCalendar(void);
    ⋮
};

void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;                         // date of first day of the month
  if ((int)FDate != 0)                         // only calculate offset if date is valid
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);  // get elements of date
    FirstDate = TDateTime(AYear, AMonth, 1);   // date of the first
    FMonthOffset = 2 - FirstDate.DayOfWeek();  // generate the offset into the grid
  }
  Refresh();                                   // always repaint the control
}
```

2 Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

Delphi example

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                  { this is already here }
  ⋮                                                     { other initializations here }
  UpdateCalendar;                                               { set proper offset }
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                                            { this was already here }
  UpdateCalendar;                                      { this previously called Refresh }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  ⋮
    FDate := EncodeDate(AYear, AMonth, ADay);                { encode the modified date }
    UpdateCalendar;                                    { this previously called Refresh }
  end;
end;
```

C++ example

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner)
  : TCustomGrid(Owner)
{
  ⋮
  UpdateCalendar();
}

void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                                  // this was already here
  UpdateCalendar();                               // this previously called Refresh
}

void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  ⋮
  FDate = TDateTime(AYear, AMonth, ADay);        // this was already here
  UpdateCalendar();                              // this previously called Refresh
}
```

3 Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

Delphi example

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;          { calculate day for this cell }
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                                                { return -1 if invalid }
```

```
end;
```

C++ example

```
int __fastcall TSampleCalendar::DayNum(int ACol, int ARow)
{
  int result = FMonthOffset + ACol + (ARow - 1) * 7;       // calculate day for this cell
  if ((result < 1)||(result > MonthDays[IsLeapYear(Year)][Month]))
    result = -1;   // return -1 if invalid
  return result;
}
```

Remember to add the declaration of *DayNum* to the component's type declaration.

4 Now that you can calculate where the dates go, you can update *DrawCell* to fill in the dates:

Delphi example

```
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
  TheText: string;
  TempDay: Integer;
begin
  if ARow = 0 then                                          { if this is the header row ...}
    TheText := ShortDayNames[ACol + 1]                             { just use the day name }
  else begin
    TheText := '';                                            { blank cell is the default }
    TempDay := DayNum(ACol, ARow);                             { get number for this cell }
    if TempDay <> -1 then TheText := IntToStr(TempDay);        { use the number if valid }
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;
```

C++ example

```
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const TRect &ARect,
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0)                                 // this is the header row
    TheText = ShortDayNames[ACol + 1];           // just use the day name
  else
  {
    TheText = "";                                // blank cell is the default
    TempDay = DayNum(ACol, ARow);                // get number for this cell
    if (TempDay != -1) TheText = IntToStr(TempDay); // use the number if valid
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left -
    Canvas->TextWidth(TheText)) / 2,
  ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
}
```

Now if you reinstall the calendar component and place one on a form, you will see the proper information for the current month.

Selecting the current day

Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes.

To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:

Delphi example

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    ⋮ { existing statements to set FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { this is already here }
end;
```

C++ example

```
void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;
  if ((int) FDate != 0)
  {
  ⋮                                    // existing statements to set FMonthOffset
    Row = (ADay - FMonthOffset) / 7 + 1;
    Col = (ADay - FMonthOffset) % 7;
  }
  Refresh();                           // this is already here
}
```

Note that you are now reusing the *ADay* variable previously set by decoding the date.

Navigating months and years

Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a "next month" feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component.

The only drawback to encapsulating common manipulations into methods is that methods are only available at runtime. However, such manipulations are generally only cumbersome when performed repeatedly, and that is fairly rare at design time.

For the calendar, add the following four methods for next and previous month and year. Each of these methods uses the *IncMonth* function in a slightly different manner to increment or decrement *CalendarDate*, by increments of a month or a year. After incrementing or decrementing *CalendarDate*, decode the date value to fill the Year, Month, and Day properties with corresponding new values.

Delphi example

```
procedure TCalendar.NextMonth;
begin
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
begin
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;

procedure TCalendar.PrevYear;
begin
  DecodeDate(IncMonth(CalendarDate, -12), Year, Month, Day);
end;
```

C++ example

```
void __fastcall TSampleCalendar::NextMonth()
{
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
}

void __fastcall TSampleCalendar::PrevMonth()
{
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
}

void __fastcall TSampleCalendar::NextYear()
{
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
}


void __fastcall TSampleCalendar::PrevYear()
{
  DecodeDate(IncMonth(CalendarDate, -12), Year, Month, Day);
}
```

Be sure to add the declarations of the new methods to the class declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years.

Navigating days

Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell.

The process of navigating days consists of

- Moving the selection
- Providing an OnChange event
- Excluding blank cells

Moving the selection

The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior.

To handle movements within the calendar, override the *Click* method of the grid.

When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior.

The following example is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click* to *TSampleCalendar*.

Delphi example

In Delphi, include the **override** directive afterward.

```
procedure TSampleCalendar.Click;
var
  TempDay: Integer;
begin
  inherited Click;                              { remember to call the inherited method! }
  TempDay := DayNum(Col, Row);                 { get the day number for the clicked cell }
  if TempDay <> -1 then Day := TempDay;                             { change day if valid }
end;
```

C++ example

```
void __fastcall TSampleCalendar::Click()
{
  int TempDay = DayNum(Col, Row);           // get the day number for the clicked cell
  if (TempDay != -1) Day = TempDay;         // change day if valid
}
```

Providing an OnChange event

Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.

Add an *OnChange* event to *TSampleCalendar*.

1 Declare the event, a data member to store the event, and a dynamicvirtual method to call the event:

Delphi example

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  ⋮
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  ⋮
```

C++ example

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TNotifyEvent FOnChange;
    ⋮
protected:
    virtual void __fastcall Change();
__published:
    __property TNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
    ⋮
}
```

2 Write the *Change* method:

Delphi example

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

C++ example

```
void __fastcall TSampleCalendar::Change()
{
  if(FOnChange != NULL) FOnChange(this);
}
```

3 Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

Delphi example

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                        { this is the only new statement }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ⋮                                         { many statements setting element values }
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;                                                              { this is new }
  end;
end;
```

C++ example

```
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  UpdateCalendar();
  Change();             // this is the only new statement
}

void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  ⋮  // many statements setting element values
  FDate = TDateTime(AYear, AMonth, ADay);
  UpdateCalendar();
  Change();             // this is new
}
```

Applications using the calendar component can now respond to changes in the date of the component by attaching handlers to the *OnChange* event.

Excluding blank cells

As the calendar is written, the user can select a blank cell, but the date does not change. It makes sense, then, to disallow selection of the blank cells.

To control whether a given cell is selectable, override the *SelectCell* method of the grid.

SelectCell is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable.

You can override *SelectCell* to return false if the cell does not contain a valid date:

Delphi example

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False            { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow);      { otherwise, use inherited value }
end;
```

C++ example

```
bool __fastcall TSampleCalendar::SelectCell(int ACol, int ARow)
{
  if (DayNum(ACol,ARow) == -1) return false;          // -1 indicates invalid date
  else return TCustomGrid::SelectCell(ACol, ARow);    // otherwise, use inherited value
}
```

Now if the user clicks a blank cell or tries to move to one with an arrow key, the calendar leaves the current cell selected.

Chapter

46

Making a control data aware

When working with database connections, it is often convenient to have controls that are *data aware*. That is, the application can establish a link between the control and some part of a database. CLX includes data-aware labels, edit boxes, list boxes, combo boxes, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Chapter 20, "Using data controls."

There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable data awareness, or *data editing*, where the user can edit the values in the database by manipulating the control. Note also that the degree of involvement with the database can vary, from the simplest case, a link with a single field, to more complex cases, such as multiple-record controls.

This chapter first illustrates the simplest case, making a read-only control that links to a single field in a dataset. The specific control used will be the *TSampleCalendar* calendar created in Chapter 45, "Customizing a grid."

The chapter then continues with an explanation of how to make the new data browsing control a data editing control.

Creating a data browsing control

Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps:

- Creating and registering the component
- Adding the data link
- Responding to data changes

Creating and registering the component

You create every component the same way: create and save a unit (Delphi) or .cpp and .h files (C++), derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 35-7.

For this example, follow the general procedure for creating a component, with these specifics:

1 Save the component's unit (Delphi) or .cpp and .h files (C++):
   - In Delphi, name the unit *dbcal*.
   - In C++, name the header file *dbcal.h* and its .cpp file *dbcal.cpp*.

2 Derive a new component class called *TDBCalendar*, descended from the component *TSampleCalendar*. Chapter 45, "Customizing a grid," shows you how to create the *TSampleCalendar* component.

3 Register *TDBCalendar* on the Samples page of the Component palette.

Delphi example

The resulting unit descending from *TCustomGrid* in CLX should look like this:

```
unit CalSamp;

interface

uses
  SysUtils, Classes, QGraphics, QControls, QForms; QDialogs, QGrids;

type
  TSampleCalendar = class(TCustomGrid)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;

end.
```

In Delphi, if descending from the CLX version of *TCustomGrid*, only the **uses** clause would differ showing CLX units instead.

If you install the calendar component now, you will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in "Filling in the cells" on page 45-7.

Delphi example

The resulting unit should look like this:

```
unit DBCal;

interface

uses   SysUtils, Classes, QGraphics, QControls, QForms; QDialogs, QGrids, Calendar;

type
  TDBCalendar = class(TSampleCalendar)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDBCalendar]);
end;

end.
```

C++ example

The resulting header file should look like this:

```
#ifndef DBCalH
#define DBCalH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QGrids.hpp>              // include the Grids header
#include “calsamp.h”                  // include the header that declares TSampleCalendar
//---------------------------------------------------------------------------
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

The .cpp file should look like this:

```
#pragma link “Calsamp”                        // link in TSampleCalendar

#include <clx.h>
#pragma hdrstop
#include "DBCal.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TDBCalendar *ValidCtrCheck()
```

```
{
   return new TDBCalendar(NULL);
}
//---------------------------------------------------------------------------
namespace Dbcal
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TDBCalendar)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

Note In C++, if you used the Component wizard to begin the *TDBCalendar* component, your header file will have the constructor already declared, and the .cpp file has the constructor's definition.

You can now make the new calendar a data browser.

Making the control read-only

Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database.

Making the calendar read-only involves:

- Adding the ReadOnly property.
- Allowing needed updates.

Note If you started with the *TCalendar* component from the Samples page instead of *TSampleCalendar*, it already has a *ReadOnly* property, so you can skip these steps.

Adding the ReadOnly property

By adding a *ReadOnly* property, you will provide a way to make the control read-only at design time. When that property is set to true, you can make all cells in the control unselectable.

1 Add the property declaration and a private data member to hold the value:

D

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean;                                  { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override;     { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
```

```
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  FReadOnly := True;                                             { set the default value }
end;
```

C++ example

Add the property declaration and a private data member in the dbcal.h file:

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    bool FReadOnly;                                  // field for internal storage
protected:
public:
    virtual __fastcall TDBCalendar(TComponent* Owner);
__published:
    __property ReadOnly = {read=FReadOnly, write=FReadOnly, default=true};
};
```

Write the constructor in dbcal.cpp:

```
virtual __fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :
  TSampleCalendar(Owner)
{
  FReadOnly = true;                                  // sets the default value
}
```

2 Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in "Excluding blank cells" on page 45-20.

Delphi example

```
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False                         { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
```

C++ example

```
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
  if (FReadOnly) return false;                       // can't select if read only
  return TSampleCalendar::SelectCell(ACol, ARow);     // otherwise, use inherited method
}
```

Remember to add the declaration of *SelectCell* to the class declaration of *TDBCalendar*. In Delphi, append the override directive.

If you now add the calendar to a form, you will find that the component ignores clicks and keystrokes. It also fails to update the selection position when you change the date.

Allowing needed updates

The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but because *SelectCell* disallows changes, the selection remains in place, even though the date changes.

To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to true:

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;                                  { private flag for internal use }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public
    procedure UpdateCalendar; override;                { remember the override directive }
  end;
⋮
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False       { allow select if updating }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;

procedure TDBCalendar.UpdateCalendar;
begin
  FUpdating := True;                                          { set flag to allow updates }
  try
    inherited UpdateCalendar;                                          { update as usual }
  finally
    FUpdating := False;                                          { always clear the flag }
  end;
end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    ⋮
    bool FUpdating;                                // private flag for internal use
protected:
    virtual bool __fastcall SelectCell(long ACol, long ARow);
public:
    ⋮
    virtual void __fastcall UpdateCalendar();
    ⋮
};

bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
    if (!FUpdating && FReadOnly) return false;          // can't select if read only
    return TSampleCalendar::SelectCell(ACol, ARow);     // otherwise, use inherited method
}

void __fastcall TDBCalendar::UpdateCalendar()
{
  FUpdating=true;                                     // set flag to allow updates
  try
  {
     TSampleCalendar::UpdateCalendar();              // update as usual
  }
  catch(...)
  {
    FUpdating = false;
    throw;
  }
  FUpdating = false;                                  // always clear the flag
}
```

The calendar still disallows user changes, but now correctly reflects changes made in the date by changing the date properties. Now that you have a true read-only calendar control, you are ready to add the data browsing ability.

Adding the data link

The connection between a control and a database is handled by a class called a *data link*. The data link class that connects a control with a single data member in a database is *TFieldDataLink.* There are also data links for entire tables.

A data-aware control *owns* its data link class. That is, the control has the responsibility for constructing and destroying the data link. For details on management of owned classes, see Chapter 44, “Creating a graphic control.”

Establishing a data link as an owned class requires these three steps:

1 Declaring the class data member.

2 Declaring the access properties.

3 Initializing the data link.

Declaring the data members

A component needs a data member for each of its owned classes, as explained in , “Declaring the data members,” on page 44-8. In this case, the calendar needs a data member of type *TFieldDataLink* for its data link.

Declare a field for the data link in the calendar:

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FDataLink: TFieldDataLink;
  ⋮
  end;
```

Before you can compile the application, you need to add DB and DBCtrls to the unit's **uses** clause.

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    TFieldDataLink *FDataLink;
    ⋮
};
```

In C++, before you can compile the application, you need to include the db.hpp and dbtables.hpp files in the dbcal.h file:

```
#include <DB.hpp>
#include <DBTables.hpp>
```

Declaring the access properties

Every data-aware control has a *DataSource* property that specifies which data source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned classes in the example in Chapter 44, "Creating a graphic control", which provide access to private data members in the class, these access properties maintained by the owned class. That is, you will create properties that enable the control and its data link to share the same data source and field.

Declare the *DataSource* and *DataField* properties and their implementation methods, then write the methods as "pass-through" methods to the corresponding properties of the data link class:

An example of declaring access properties

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar)
  private                                          { implementation methods are private }
    ...
    function GetDataField: string;                 { returns the name of the data field }
    function GetDataSource: TDataSource;         { returns reference to the data source }
    procedure SetDataField(const Value: string);           { assigns name of data field }
    procedure SetDataSource(Value: TDataSource);              { assigns new data source }
  published                                     { make properties available at design time }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

```
⋮
function TDBCalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
   ⋮
    AnsiString __fastcall GetDataField();             // methods are private
    TDataSource *__fastcall GetDataSource();          // returns name of data field
    void __fastcall SetDataField(AnsiString Value); // returns reference to data
                                                        // source
    void __fastcall SetDataSource(TDataSource *Value);    // assigns name of data field
    ⋮
__published:                                // make properties available at design time
    __property AnsiString DataField = {read=GetDataField, write=SetDataField, nodefault};
    __property TDataSource * DataSource = {read=GetDataSource, write=SetDataSource,
      nodefault};
   ⋮
};

AnsiString __fastcall TDBCalendar::GetDataField()
{
  return FDataLink->FieldName;
}

TDataSource *__fastcall TDBCalendar::GetDataSource()
{
  return FDataLink->DataSource;
}

void __fastcall TDBCalendar::SetDataField(AnsiString Value)
{
  FDataLink->FieldName = Value;
}

void __fastcall TDBCalendar::SetDataSource(TDataSource *Value)
{
  if(Value != NULL)
```

```
      Value->FreeNotification(this);
    FDataLink->DataSource = Value;
  }
```

Now that you have established the links between the calendar and its data link, there is one more important step. You must construct the data link class when the calendar control is constructed, and destroy the data link before destroying the calendar.

Initializing the data link

A data-aware control needs access to its data link throughout its existence, so it must construct the data link object as part of its own constructor, and destroy the data link object before it is itself destroyed.

Delphi example

Override the *Create* and *Destroy* methods of the calendar to construct and destroy the data link object, respectively:

```
type
  TDBCalendar = class(TSampleCalendar)
  public                                  { constructors and destructors are always public }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                  { always call the inherited constructor first }
  FDataLink := TFieldDataLink.Create;                    { construct the datalink object }
  FDataLink.Control := self;              {let the datalink know about the calendar }
  FReadOnly := True;                                              { this is already here }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.Free;                              { always destroy owned objects first... }
  inherited Destroy;                                { ...then call inherited destructor }
end;
```

C++ example

Override the constructor and destructor of the calendar:

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
public:
    virtual __fastcall TDBCalendar(TComponent *Owner);
    __fastcall ~TDBCalendar();
};

__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) : TSampleCalendar(Owner)
{
    FReadOnly = true;
    FDataLink = new TFieldDataLink();
    FDataLink->Control = this;
}
__fastcall TDBCalendar::~TDBCalendar()
{
    FDataLink->Control = NULL;
    FDataLink->OnUpdateData = NULL;
    delete FDataLink;
}
```

Now you have a complete data link, but you have not yet told the control what data it should read from the linked field. The next section explains how to do that.

Responding to data changes

Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field.

Data link classes all have events named *OnDataChange*. When the data source indicates a change in its data, the data link object calls any event handler attached to its *OnDataChange* event.

To update a control in response to data changes, attach a handler to the data link's *OnDataChange* event.

In this case, you will add a method to the calendar, then designate it as the handler for the data link's *OnDataChange*.

Declare and implement the *DataChange* method, then assign it to the data link's *OnDataChange* event in the constructor. In the destructor, detach the *OnDataChange* handler before destroying the object.

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar)
  private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject);         { must have proper parameters for event }
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                   { always call the inherited constructor first }
  FReadOnly := True;                                                 { this is already here }
  FDataLink := TFieldDataLink.Create;                     { construct the datalink object }
  FDataLink.OnDataChange := DataChange;                          { attach handler to event }
end;
destructor TDBCalendar.Destroy;
begin
  FDataLink.OnDataChange := nil;                 { detach handler before destroying object }
  FDataLink.Free;                                  { always destroy owned objects first... }
  inherited Destroy;                                   { ...then call inherited destructor }
```

```
end;

procedure TDBCalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then                         { if there is no field assigned... }
    CalendarDate := 0                                              { ...set to invalid date }
  else CalendarDate := FDataLink.Field.AsDateTime;   { otherwise, set calendar to the date }
end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall DataChange(TObject *Sender);
    ⋮
};

void __fastcall TDBCalendar::DataChange( TObject* Sender)
{
  if (FDataLink->Field == NULL)                          // if no field is assigned ...
  CalendarDate = 0;                                      // ...set to invalid date
  else CalendarDate = FDataLink->Field->AsDateTime;  // otherwise, set to new data
}

__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :  TSampleCalendar(AOwner)
{
  FReadOnly = true;
  FDataLink = new TFieldDataLink();            // construct the datalink object
  FDataLink->Control = this;
  FDataLink->OnDataChange = DataChange;        // attach the handler
}

__fastcall TDBCalendar::~TDBCalendar()
{
  FDataLink->Control = NULL;
  FDataLink->OnUpdateData = NULL;
  FDataLink->OnDataChange = NULL;                // detach the handler before...
  delete FDataLink;                              // ...destroying the datalink object
}
```

You now have a data browsing control.

Creating a data editing control

When you create a data editing control, you create and register the component and add the data link just as you do for a data browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues.

For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset.

The data editing control described here is the same calendar control described in the first part of the chapter. The control is modified so that it can edit as well as view the data in its linked field.

Modifying the existing control to make it a data editing control involves:

- Changing the default value of FReadOnly.
- Handling mouse-down and key-down events.
- Updating the field data link class.
- Modifying the Change method.
- Updating the dataset.

Changing the default value of FReadOnly

Because this is a data editing control, the *ReadOnly* property should be set to false by default. To make the *ReadOnly* property false, change the value of *FReadOnly* in the constructor:

Delphi example

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  ⋮
  FReadOnly := False;  { set the default value }
  ⋮
end;
```

C++ example

```
__fastcall TDBCalendar::TDBCalendar (TComponent* Owner) : TSampleCalendar(Owner)
{
  FReadOnly = false;             // set the default value
  ⋮
}
```

Handling mouse-down and key-down events

When the user of the control begins interacting with it, the control receives either mouse-down events or a key-down event. To enable a control to respond to these events, you must write handlers that respond to these messages.

- Responding to mouse-down events
- Responding to key-down events

Notification is from the operating system in the form of system events. For information on writing components that respond to system and widget events, see "Responding to system notifications using CLX" on page 41-1.

Responding to mouse-down events

A *MouseDown* method is a protected method for a control's *OnMouseDown* event. The control itself calls *MouseDown* in response to a notification from the operating system. When you override the inherited *MouseDown* method, you can include code that provides other responses in addition to calling the *OnMouseDown* event.

To override *MouseDown,* add the *MouseDown* method to the *TDBCalendar* class:

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar);
   ⋮
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
   ⋮
  end;

procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
  begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
  end;
end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
⋮
protected:
    virtual void __fastcall MouseDown(TMouseButton Button,  TShiftState Shift, int X,
      int Y);
    ⋮
};
```

Write the *MouseDown* method in the .cpp file:

```
void __fastcall TDBCalendar::MouseDown(TMouseButton Button,  TShiftState Shift, int  X,
  int Y)
{
  TMouseEvent MyMouseDown;                              // declare event type
  if (!FReadOnly && FDataLink->Edit())                  // if the field can be edited
    TSampleCalendar::MouseDown(Button, Shift, X, Y);   // call the inherited MouseDown
  else
  {
    MyMouseDown = OnMouseDown;                         // assign OnMouseDown event
    if (MyMouseDown != NULL) MyMouseDown(this, Button,  // execute code in the...
        Shift, X, Y);                                    // ...OnMouseDown event handler
    }
  }
```

When *MouseDown* responds to a mouse-down message, the inherited *MouseDown* method is called only if the control's *ReadOnly* property is false and the data link object is in edit mode, which means the field can be edited. If the field cannot be edited, the code the programmer put in the *OnMouseDown* event handler, if one exists, is executed.

Responding to key-down events

A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a Windows key-down message. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.

To override *KeyDown*, follow these steps:

1 Add a *KeyDown* method to the *TDBCalendar* class:

D

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar);
    ⋮
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
    ⋮
  end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
   ⋮
protected:
    virtual void __fastcall KeyDown(unsigned short &Key,  TShiftState Shift);
   ⋮
};
```

1 Implement the *KeyDown* method:

D

Delphi example

```
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [Key_Up, Key_Down, Key_Left, Key_Right, Key_End,
    Key_Home, Key_Prior, Key_Next]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
  begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
  end;
end;
```

C++ example

Write the *KeyDown* method in the .cpp file:

```
void __fastcall TDBCalendar::KeyDown(unsigned short &Key,  TShiftState Shift)
{
  TKeyEvent MyKeyDown;                              // declare event type
  Set<unsigned short,0,8> keySet;
  keySet = keySet << Key_Up << Key_Down << Key_Left      // assign virtual keys to set
    << Key_Right << Key_End << Key_Home << Key_Prior << Key_Next;
  if (!FReadOnly &&                                // if control is not read only...
      (keySet.Contains(Key)) &&                     // ...and key is in the set...
      FDataLink->Edit() )                           // ...and field is in edit mode
  {
    TCustomGrid::KeyDown(Key, Shift);           // call the inherited KeyDown method
  }
  else
  {
    MyKeyDown = OnKeyDown;                          // assign OnKeyDown event
    if (MyKeyDown != NULL) MyKeyDown(this,Key,Shift); // execute code in...
  }                                                    // ...OnKeyDown event handler
}
```

When *KeyDown* responds to a mouse-down event, the inherited *KeyDown* method is called only if the control's *ReadOnly* property is false, the key pressed is one of the cursor control keys, and the data link object is in edit mode, which means the field can be edited. If the field cannot be edited or some other key is pressed, the code the programmer put in the *OnKeyDown* event handler, if one exists, is executed.

Updating the field data link class

There are two types of data changes:

- A change in a field value that must be reflected in the data-aware control.
- A change in the data-aware control that must be reflected in the field value.

The *TDBCalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change.

Similarly, the field data link class also has an *OnUpdateData* event that occurs as the user of the control modifies the contents of the data-aware control. The calendar control has a *UpdateData* method that becomes the event handler for the *OnUpdateData* event. *UpdateData* assigns the changed value in the data-aware control to the field data link.

1 To reflect a change made to the value in the calendar in the field value, add an *UpdateData* method to the private section of the calendar component:

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
    ⋮
  end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall UpdateData(TObject *Sender);
};
```

2 Implement the *UpdateData* method:

Delphi example

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;        { set field link to calendar date }
end;
```

C++ example

Write the *UpdateData* method in the .cpp file:

```
void __fastcall TDBCalendar::UpdateData( TObject* Sender)
{
    FDataLink->Field->AsDateTime = CalendarDate;     // set field link to calendar date
}
```

3 Within the constructor for *TDBCalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

Delphi example

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

C++ example

```
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
  : TSampleCalendar(Owner)
{
  FDataLink = new TFieldDataLink();       // this was already here
  FDataLink->OnDataChange = DataChange;   // this was here too
  FDataLink->OnUpdateData = UpdateData;   // assign UpdateData to the OnUpdateData event
}
```

Modifying the Change method

The *Change* method of the *TDBCalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date.

When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code. These are the steps to follow:

1 Add a new *Change* method to the *TDBCalendar* component:

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure Change; override;
   ⋮
  end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
protected:
    virtual void __fastcall Change();
    ⋮
};
```

2 Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

Delphi example

```
procedure TDBCalendar.Change;
begin
  FDataLink.Modified;                { call the Modified method }
  inherited Change;                  { call the inherited Change method }
end;
```

C++ example

```
void __fastcall TDBCalendar::Change()
{
  if (FDataLink != NULL)
    FDataLink->Modified();              // call the Modified method
  TSampleCalendar::Change();            // call the inherited Change method
}
```

Updating the dataset

So far, a change within the data-aware control has changed values in the field data link class. The final step in creating a data editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the *Tab* key.

TWidgetControl has a protected *DoExit* method that is called when input focus shifts away from the control. This method calls the event handler for the *OnExit* event. You can override this method to update the record in the dataset before generating the *OnExit* event handler.

To update the dataset when the user exits the control, follow these steps:

1 Add an override for the *DoExit* method to the *TDBCalendar* component:

Delphi example

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
   ⋮
  end;
```

C++ example

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    DYNAMIC void __fastcall DoExit(void);
   ⋮
};
```

2 Implement the *DoExit* method so it looks like this:

Delphi example

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                          { tell data link to update database }
  except
    on Exception do SetFocus;                      { if it failed, don't let focus leave }
 end;
  inherited;                             { let the inherited method generate an OnExit event }
end;
```

C++ example

```
void __fastcall TDBCalendar::DoExit(void)
{
  try
  {
    FDataLink.UpdateRecord();          // tell data link to update database
  }
  catch(...)
  {
    SetFocus();                        // if it failed, don't let focus leave
    throw;
  }
  TCustomGrid::DoExit(); // let the inherited method generate an OnExit event
}
```

Chapter

47

Making a dialog box a component

You will find it convenient to make a frequently used dialog box into a component that you add to the Component palette. Your dialog box components will work just like the components that represent the standard common dialog boxes. The goal is to create a simple component that a user can add to a project and set properties for at design time.

Making a dialog box a component requires these steps:

1 Defining the component interface
2 Creating and registering the component
3 Creating the component interface
4 Testing the component

The CLX “wrapper” component associated with the dialog box creates and executes the dialog box at runtime, passing along the data the user specified. The dialog-box component is therefore both reusable and customizable.

In this chapter, you will see how to create a wrapper component around the generic About Box form provided in the Object Repository.

Note Copy the files about.pas (in C++, add about.h or about.cpp) and about.xfm into your working directory. In C++, add about.cpp to your project so that an about.obj file is created when your dialog wrapper component builds.

There are not many special considerations for designing a dialog box that will be wrapped into a component. Nearly any form can operate as a dialog box in this context.

Defining the component interface

Before you can create the component for your dialog box, you need to decide how you want developers to use it. You create an interface between your dialog box and applications that use it.

For example, look at the properties for the common dialog box components. They enable the developer to set the initial state of the dialog box, such as the caption and initial control settings, then read back any needed information after the dialog box closes. There is no direct interaction with the individual controls in the dialog box, just with the properties in the wrapper component.

The interface must therefore contain enough information that the dialog box form can appear in the way the developer specifies and return any information the application needs. You can think of the properties in the wrapper component as being persistent data for a transient dialog box.

In the case of the About box, you do not need to return any information, so the wrapper's properties only have to contain the information needed to display the About box properly. Because there are four separate fields in the About box that the application might affect, you will provide four string-type properties to provide for them.

Creating and registering the component

You create every component the same way: create and save a unit (Delphi) or .cpp and .h files (C++), derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 35-7.

For this example, follow the general procedure for creating a component, with these specifics:

1 Save the component's unit (Delphi) or .cpp and .h files (C++):
   - In Delphi, name the unit *aboutdlg*.
   - In C++, name the header file *aboutdlg.h* and its .cpp file *aboutdlg.cpp*.
2 Derive a new component type called *TAboutBoxDlg*, descended from *TComponent*.
3 Register *TAboutBoxDlg* on the Samples page of the Component palette.

D Delphi example

The resulting unit should look like this:

```
unit AboutDlg;
interface
uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms,
  QDialogs, QStdCtrls;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
```

```
end;
end.
```

C++ example

The resulting .hpp file should look like this:

```
#ifndef AboutDlgH
#define AboutDlgH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Classes.hpp>
//---------------------------------------------------------------------------
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

The .cpp file of the unit should look like this:

```
#include <clx.h>
#pragma hdrstop
#include "AboutDlg.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TAboutBoxDlg *ValidCtrCheck()
{
  return new TAboutBoxDlg(NULL);
}
//---------------------------------------------------------------------------
namespace AboutDlg {
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TAboutBoxDlg)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

In C++, if you used the Component wizard to begin this component, *TAboutDlg* will also have a constructor added.

The new component now has only the capabilities built into *TComponent*. It is the simplest nonvisual component. In the next section, you will create the interface between the component and the dialog box.

Creating the component interface

These are the steps to create the component interface:

1 Including the form unit files.
2 Adding interface properties.
3 Adding the Execute method.

Including the form unit files

For your wrapper component to initialize and display the wrapped dialog box, you must add the form's unit to the uses clause of the wrapper component's unit (Delphi) or the form's files to the project (C++).

Delphi example

In Delphi, append *About* to the uses clause of the *AboutDlg* unit so that the uses clause looks like this:

```
uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms,
  QDialogs, QStdCtrls, About;
```

The form unit always declares an instance of the form class. In the case of the About box, the form class is *TAboutBox*, and the *About* unit includes the following declaration:

```
var
  AboutBox: TAboutBox;
```

In Delphi, by adding *About* to the uses clause, you make *AboutBox* available to the wrapper component.

C++ example

In C++, include about.hpp and link in about.obj in the component's header file:

```
#include "About.h"
#pragma link "About.obj"
```

The form header file always declares an instance of the form class. In the case of the About box, the form class is *TAboutBox*, and the about.h file includes the following:

```
extern TAboutBox *AboutBox;
```

Adding interface properties

Before proceeding, decide on the properties your wrapper needs to enable developers to use your dialog box as a component in their applications. Then, you can add declarations for those properties to the component's class declaration.

Properties in wrapper components are somewhat simpler than the properties you would create if you were writing a regular component. Remember that in this case, you are just creating some persistent data that the wrapper can pass back and forth to the dialog box. By putting that data in the form of properties, you enable developers to set data at design time so that the wrapper can pass it to the dialog box at runtime.

Declaring an interface property requires two additions to the component's class declaration:

- A private data member, which is a variable the wrapper uses to store the value of the property
- The published property declaration itself, which specifies the name of the property and tells it which data member to use for storage

Interface properties of this sort do not need access methods. They use direct access to their stored data. By convention, the data member that stores the property's value has the same name as the property, but with the letter *F* in front. The data member and the property *must* be of the same type.

For example, to declare an integer-type interface property called *Year*, you would declare it as follows:

Delphi example

```
type
  TMyWrapper = class(TComponent)
  private
    FYear: Integer;                                { field to hold the Year-property data }
  published
    property Year: Integer read FYear write FYear;        { property matched with storage }
  end;
```

C++ example

```
class PACKAGE TWrapper : public TComponent
{
private:
    int FYear;                              // data member to hold the Year-property data
protected:
public:
__published:
    __property int Year = {read=FYear, write=FYear};        // property matched with storage
};
```

For this About box, you need four string-type properties—one each for the product name, the version information, the copyright information, and any comments.

Delphi example

```
type
  TAboutBoxDlg = class(TComponent)
  private
    FProductName, FVersion, FCopyright, FComments: string;              { declare fields }
  published
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
  end;
```

C++ example

The aboutdlg.h file should look like this:

```
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
    int FYear;
    String FProductName, FVersion, FCopyright, FComments;
protected:
public:
__published:
    __property int Year = {read=FYear, write=FYear};
    __property String ProductName = {read=FProductName, write=FProductName};
    __property String Version = {read=FVersion, write=FVersion};
    __property String Copyright = {read=FCopyright, write=FCopyright};
    __property String Comments = {read=FComments, write=FComments};
};
```

When you install the component onto the Component palette and place the component on a form, you will be able to set the properties, and those values will automatically stay with the form. The wrapper can then use those values when executing the wrapped dialog box.

Adding the Execute method

The final part of the component interface is a way to open the dialog box and return a result when it closes. As with the common dialog box components, you use a boolean function called *Execute* that returns true if the user clicks OK, or false if the user cancels the dialog box.

The declaration for the *Execute* method always looks like this:

Delphi example

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

C++ example

```
class PACKAGE TMyWrapper : public TComponent
{
  ⋮
public:
    bool __fastcall Execute();
   ⋮
};
```

The minimum implementation for *Execute* needs to construct the dialog box form, show it as a modal dialog box, and return either true or false, depending on the return value from *ShowModal*.

Here is the minimal *Execute* method for a dialog-box form of type *TMyDialogBox*:

Delphi example

```
function TMyWrapper.Execute: Boolean;
begin
  DialogBox := TMyDialogBox.Create(Application);                     { construct the form }
  try
    Result := (DialogBox.ShowModal = IDOK);     { execute; set result based on how closed }
  finally
    DialogBox.Free;                                                   { dispose of the form }
  end;
end;
```

Note the use of a **try..finally** block to ensure that the application disposes of the dialog box object even if an exception occurs. In general, whenever you construct an object this way, you should use a **try..finally** block to protect the block of code and make certain the application frees any resources it allocates.

C++ example

```
bool __fastcall TMyWrapper::Execute()
{
  DialogBox = new TMyDialogBox(Application);    // construct the form
    bool Result;
  try
  {
    Result = (DialogBox->ShowModal()  IDOK);  // execute; set result based on how closed
  }
  catch(...)
  {
    Result = false;                             // if it fails, set Result to false
  }
  DialogBox->Free();                            // dispose of form
}
```

In practice, there will be more code inside the try..finally block. Specifically, before calling *ShowModal*, the wrapper will set some of the dialog box's properties based on the wrapper component's interface properties. After *ShowModal* returns, the wrapper will probably set some of its interface properties based on the outcome of the dialog box execution.

In the case of the About box, you need to use the wrapper component's four interface properties to set the contents of the labels in the About box form. Because the About box does not return any information to the application, there is no need to do anything after calling *ShowModal*. Write the About box wrapper's *Execute* method so that it looks like the following.

Delphi example

Add the following declaration for the *Execute* method within the public part of the *TAboutDlg* class:

```
type
  TAboutDlg = class(TComponent)
public
  function Execute: Boolean;
end;

function TAboutBoxDlg.Execute: Boolean;
begin
  AboutBox := TAboutBox.Create(Application);                       { construct About box }
  try
    if ProductName = '' then                                { if product name's left blank... }
      ProductName := Application.Title;                    { ...use application title instead }
    AboutBox.ProductName.Caption := ProductName;                         { copy product name }
    AboutBox.Version.Caption := Version;                                 { copy version info }
    AboutBox.Copyright.Caption := Copyright;                           { copy copyright info }
    AboutBox.Comments.Caption := Comments;                                   { copy comments }
    AboutBox.Caption := 'About ' + ProductName;                      { set About-box caption }
    with AboutBox do begin
      ProgramIcon.Picture.Graphic := Application.Icon;                           { copy icon }
      Result := (ShowModal = IDOK);                                { execute and set result }
    end;
  finally
    AboutBox.Free;                                                    { dispose of About box }
  end;
end;
```

C++ example

Add the following declaration in the aboutdlg.cpp file:

```
bool __fastcall TAboutBoxDlg::Execute()
{
  AboutBox = new TAboutBox(Application);                // construct the About box
  bool Result;
  try
  {
    if (ProductName == "")                              // if product name's left blank ...
      ProductName = Application->Title;                 // ... use application title instead
    AboutBox->ProductName->Caption = ProductName;       // copy product name
    AboutBox->Version->Caption = Version;               // copy version information
    AboutBox->Copyright->Caption = Copyright;           // copy copyright information
    AboutBox->Comments->Caption = Comments;             // copy comments
    AboutBox->Caption = "About " + ProductName;         // set About-box caption
    Result = (AboutBox->ShowModal() == IDOK);           // execute and set result
  }
  catch(...)
  {
    Result = false;                     // if it fails, set Result to false
    ⋮
  }
  AboutBox->Free();                     // dispose of About box
  return Result == IDOK;                // compare Result to IDOK and return Boolean value
}
```

To the aboutdlg.h header, add the declaration for the *Execute* method to the public part of the *TAboutDlg* class:

```
class PACKAGE TAboutDlg : public TComponent
{
public:
     virtual bool __fastcall Execute();
};
```

Testing the component

Once you have installed the dialog box component, you can use it as you would any of the common dialog boxes, by placing one on a form and executing it. A quick way to test the About box is to add a command button to a form and execute the dialog box when the user clicks the button.

For example, if you created an About dialog box, made it a component, and added it to the Component palette, you can test it with the following steps:

1 Create a new project.

2 Place an About box component on the main form.

3 Place a command button on the form.

4 Double-click the command button to create an empty click-event handler.

5 In the click-event handler, type the following line of code:

```
AboutBoxDlg1.Execute;
```

```
AboutBoxDlg1->Execute();
```

6 Run the application.

When the main form appears, click the command button. The About box appears with the default project icon and the name Project1. Choose OK to close the dialog box.

You can further test the component by setting the various properties of the About box component and again running the application.

Chapter

48

Extending the IDE

You can extend and customize the IDE with the Open Tools API (often shortened to just Tools API). The Tools API is a suite of over 100 interfaces that interact with and control the IDE, including the source editor's internal buffers, keyboard macros and bindings, the debugger and the process being debugged, code completion, the message view, and the To-Do list.

Using the Tools API is a matter of writing classes that implement certain interfaces, and calling on services provided by the IDE. Your Tools API code must be compiled and loaded into the IDE at design-time as a design-time package. Thus, writing a Tools API extension is somewhat like writing a property or component editor. Before tackling the Open Tools API, make sure you are familiar with the basics of working with packages (Chapter 16, "Working with packages and components") and registering components (Chapter 42, "Making components available at design time").

It is also a good idea to read Chapter 14, "C++ language support for CLX," and especially the section "Inheritance and interfaces" on page 14-3 for information about Delphi-style interfaces.

This chapter covers the following topics:

- Overview of the Tools API
- Writing a wizard class
- Obtaining Tools API services
- Working with files and editors
- Creating forms and projects
- Notifying a wizard of IDE events

Overview of the Tools API

All of the Tools API declarations reside in a single unit called ToolsAPI. The corresponding C++ header file is ToolsAPI.hpp, and the namespace is Toolsapi. To use the Tools API, you typically use the designide package, which means you must build your Tools API add-in as a design-time package. For information about package and library issues, see “Installing the wizard package” on page 48-6.

To use the Tools API, you create classes that implement one or more of the interfaces defined in the ToolsAPI unit. The main interface for writing a Tools API extension is *IOTAWizard*, so most IDE add-ins are called wizards. Add-in wizards are, for the most part, interoperable. You can write and compile a wizard in the Delphi IDE, then use it in the C++ IDE, and vice versa.

Recall from Chapter 14 that Delphi interfaces are represented in C++ as a pure virtual class. To implement the interface, you must inherit the pure virtual class, override its member functions (and those of its ancestors), and implement the *QueryInterface* function to recognize the interface GUID.

A wizard makes use of services that the IDE exposes through the Tools API. Each service is an interface that presents a set of related functions. The implementation of the interface is hidden within the IDE. The Tools API publishes only the interface, which you can use to write your wizards without concerning yourself with the implementation of the interfaces. The various services offer access to the source editor, form designer, debugger, and so on. The section “Obtaining Tools API services” on page 48-7 examines this topic in depth.

The service and other interfaces fall into two basic categories. You can tell them apart by the prefix used for the type name:

The NTA (Native Tools API) grants direct access to actual IDE objects. When using these interfaces, the wizard must use Borland packages, which also means the wizard is tied to a specific version of the IDE.The OTA (Open Tools API) accesses the IDE only through interfaces. In theory, you could write a wizard in any language that supports COM-style interfaces, Delphi language calling conventions (__fastcall in C++), and Delphi language types such as AnsiString. OTA interfaces do not grant full access to the IDE, but almost all the functionality of the Tools API is available through OTA interfaces.The Tools API has two kinds of interfaces: those that you, the programmer, must implement and those that the IDE implements. Most of the interfaces are in the latter category: the interfaces define the capability of the IDE but hide the actual implementation. The kinds of interfaces that you must implement fall into three categories: wizards, notifiers, and creators:

- As mentioned earlier, a wizard class implements the *IOTAWizard* interface and possibly derived interfaces.
- A notifier is another kind of interface in the Tools API. The IDE uses notifiers to call back to your wizard when something interesting happens. You write a class that implements the notifier interface, register the notifier with the Tools API, and the IDE calls back to your notifier object when the user opens a file, edits source code, modifies a form, starts a debugging session, and so on. Notifiers are covered in “Notifying a wizard of IDE events” on page 48-18.
- A creator is another kind of interface that you must implement. The Tools API uses creators to create new units, projects, or other files, or to open existing files. The section “Creating forms and projects” on page 48-11 discusses this subject in more depth.

Other important interfaces are modules and editors. A module interface represents an open unit, which has one or more files. An editor interface represents an open file. Different kinds of editor interfaces give you access to different aspects of the IDE: the source editor for source files, the form designer for form files, and project resources for a resource file. The section "Working with files and editors" on page 48-9 covers these topics in more depth.

The following sections take you through the steps of writing a wizard. Refer to the online help files for the complete details of each interface.

Writing a wizard class

There are four kinds of wizards, where the wizard kind depends on the interfaces that the wizard class implements. Table 48.1 describes the four kinds of wizards.

Table 48.1 The four kinds of wizards

Interface	Description
IOTAFormWizard	Typically creates a new unit, form, or other file
IOTAMenuWizard	Automatically added to Help menu
IOTAProjectWizard	Typically creates a new application or other project
IOTAWizard	Miscellaneous wizard that doesn't fit into other categories

The four kinds of wizards differ only in how the user invokes the wizard:

- A menu wizard is added to the IDE's Help menu. When the user picks the menu item, the IDE calls the wizard's *Execute* function. Menu wizards are typically used only for prototypes and debugging.
- Form and project wizards are called repository wizards because they reside in the Object Repository. The user invokes these wizards from the New Items dialog box. The user can also see the wizards in the object repository (by choosing the Tools | Repository menu item). The user can check the New Form checkbox for a form wizard, which tells the IDE to invoke the form wizard when the user chooses the File | New Form menu item. The user can also check the Main Form checkbox. This tells the IDE to use the form wizard as the default form for a new application. The user can check the New Project checkbox for a project wizard. When the user chooses File | New Application, the IDE invokes the selected project wizard.
- The fourth kind of wizard is for situations that don't fit into the other categories. A plain wizard does not do anything automatically or by itself. Instead, you must define how the wizard is invoked.

The Tools API does not enforce any restrictions on wizards, such as requiring a project wizard to create a project. You can just as easily write a project wizard to create a form and a form wizard to create a project (if that's something you really want to do).

Implementing the wizard interfaces

Every wizard class must implement at least *IOTAWizard*, which requires implementing its ancestors, too: *IOTANotifier* and *IInterface*. Form and project wizards must implement all their ancestor interfaces, namely, *IOTARepositoryWizard*, *IOTAWizard*, *IOTANotifier*, and *IInterface*.

In C++, *IInterface* is derived from *IUnknown*, therefore, C++ wizards must also implement *IUnknown.*

Your implementation of *IInterface* (and *IUnknown*, for C++) must follow the normal rules for Delphi interfaces. That is, *QueryInterface* performs type casts, and *_AddRef* and *_Release* (Delphi) or *AddRef* and *Release* (C++) manage reference counting. You might want to use a common base class to simplify writing wizard and notifier classes.

In Delphi, the ToolsAPI unit defines a class, *TNotifierObject*, which implements the *IOTANotifier* interface with empty method bodies. You can use *TNotifierObject* as a common base class for your wizards, and avoid having to constantly reimplement the basic interfaces, *IOTANotifier*, and *IInterface*.

C++ example

The ToolsAPI namespace does not provide a class analogous to Delphi's *TNotifierObject* for C++. You can write a class similar to Delphi's *TNotifierObject* in C++, as shown below.

```
 class PACKAGE NotifierObject : public IOTANotifier {
 public:
  __fastcall NotifierObject() : ref_count(0) {}
  virtual __fastcall ~NotifierObject();
  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  long ref_count;
};
```

The implementation of the *IInterface* interface is straightforward:

```
ULONG __stdcall NotifierObject::AddRef()
{
  return InterlockedIncrement(&ref_count);
}

ULONG __stdcall NotifierObject::Release()
{
  ULONG result = InterlockedDecrement(&ref_count);
  if (ref_count == 0)
```

```
    delete this;
  return result;
}
HRESULT __stdcall NotifierObject::QueryInterface(const GUID& iid, void** obj)
{
  if (iid == __uuidof(IInterface)) {
    *obj = static_cast<IInterface*>(this);
    static_cast<IInterface*>(*obj)->AddRef();
    return S_OK;
  }
  if (iid == __uuidof(IOTANotifier)) {
    *obj = static_cast<IOTANotifier*>(this);
    static_cast<IOTANotifier*>(*obj)->AddRef();
    return S_OK;
  }
  return E_NOINTERFACE;
}
```

Although wizards inherit from *IOTANotifier*, and must therefore implement all of its functions, the IDE does not usually make use of those functions, so your implementations can be empty (as they are in Delphi's *TNotifierObject*):

```
void __fastcall NotifierObject::AfterSave()  {}
void __fastcall NotifierObject::BeforeSave() {}
void __fastcall NotifierObject::Destroyed()  {}
void __fastcall NotifierObject::Modified()   {}
```

To use *NotifierObject* as a base class you must use multiple inheritance. Your wizard class must inherit from *NotifierObject* and from the wizard interfaces that you need to implement, such as *IOTAWizard*. Because *IOTAWizard* inherits from *IOTANotifier* and *IInterface*, there is an ambiguity in the derived class: functions such as *AddRef*() are declared in every branch of the ancestral inheritance graph. To resolve this problem, pick one base class as the primary base class and delegate all ambiguous functions to that one class. For example, the class declaration might look as follows:

```
class PACKAGE MyWizard : public NotifierObject, public IOTAMenuWizard {
  typedef NotifierObject inherited;
public:
  // IOTAWizard
  virtual AnsiString __fastcall GetIDString();
  virtual AnsiString __fastcall GetName();
  virtual TWizardState __fastcall GetState();
  virtual void __fastcall Execute();

  // IOTAMenuWizard
  virtual AnsiString __fastcall GetMenuText();

  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
};
```

The class implementation might include the following:

```
ULONG __stdcall MyWizard::AddRef()  { return inherited::AddRef(); }
ULONG __stdcall MyWizard::Release() { return inherited::Release(); }
HRESULT __stdcall MyWizard::QueryInterface(const GUID& iid, void** obj)
{
  if (iid == __uuidof(IOTAMenuWizard)) {
    *obj = static_cast<IOTAMenuWizard*>(this);
    static_cast<IOTAMenuWizard*>(*obj)->AddRef();
    return S_OK;
  }
  if (iid == __uuidof(IOTAWizard)) {
    *obj = static_cast<IOTAWizard*>(this);
    static_cast<IOTAWizard*>(*obj)->AddRef();
    return S_OK;
  }
  return inherited::QueryInterface(iid, obj);
}
```

Because *AfterSave*, *BeforeSave*, and so on, have empty function bodies in the base class, you can leave them as empty function bodies in the derived class, and avoid the unnecessary call to *inherited::AfterSave*().

Once you have finished writing the wizard class, the next step is to install the wizard.

Installing the wizard package

As with any other design-time package, a wizard package must have a *Register* function. (See Chapter 42, "Making components available at design time" for details about the *Register* function.) In the *Register* function, you can register any number of wizards by calling *RegisterPackageWizard*, and passing a wizard object as the sole argument, as shown below:

```
procedure Register;
begin
  RegisterPackageWizard(MyWizard.Create);
  RegisterPackageWizard(MyOtherWizard.Create);
end;
```

```
namespace Example {
  void __fastcall PACKAGE Register()
  {
    RegisterPackageWizard(new MyWizard());
    RegisterPackageWizard(new MyOtherWizard());
  }
}
```

You can also register property editors, components, and so on, as part of the same package.

Remember that a design-time package is part of the IDE, which means all names must be unique throughout the entire application and all other design-time packages.

During development, install the wizard package the way you would any other design-time package: click the Install button in the package manager. The IDE will compile and link the package and attempt to load it. The IDE displays a dialog box telling you whether it successfully loaded the package.

Obtaining Tools API services

To do anything useful, a wizard needs access to the IDE: its editors, windows, menus, and so on. This is the role of the service interfaces. The Tools API includes many services, such as action services to perform file actions, editor services to access the source code editor, debugger services to access the debugger, and so on. Table 48.2 summarizes all the service interfaces.

Table 48.2 Tools API service interfaces

Interface	Description
INTAServices	Provides access to native IDE objects.
IOTAActionServices	Performs basic file actions: open, close, save, and reload a file.
IOTACodeCompletionServices	Provides access to code completion, allowing a wizard to install a custom code completion manager.
IOTADebuggerServices	Provides access to debugger.
IOTAEditorServices	Provides access to source code editor and its internal buffers.
IOTAKeyBindingServices	Permits a wizard to register custom keyboard bindings.
IOTAKeyboardServices	Provides access to keyboard macros and bindings.
IOTAKeyboardDiagnostics	Toggle debugging of keystrokes.
IOTAMessageServices	Provides access to message view.
IOTAModuleServices	Provides access to open files.
IOTAPackageServices	Queries the names of all installed packages and their components.
IOTAServices	Miscellaneous services.
IOTAToDoServices	Provides access to the To-Do list, allowing a wizard to install a custom To-Do manager.
IOTAToolsFilter	Registers tools filter notifiers.
IOTAWizardServices	Registers and unregisters wizards.

D To use a service interface, cast the *BorlandIDEServices* variable to the desired service using the global *Supports* function, which is defined in the SysUtils unit.

```
procedure set_keystroke_debugging(debugging: Boolean);
var
  diag: IOTAKeyboardDiagnostics
begin
  if Supports(BorlandIDEServices, IOTAKeyboardDiagnostics, diag) then
    diag.KeyTracing := debugging;
end;
```

In C++, the Supports function is a member of *IInterface*; therefore, you can call it directly through the *BorlandIDEServices* variable.

```
void set_keystroke_debugging(bool debugging)
{
  _di_IOTAKeyboardDiagnostics diag;
  if (BorlandIDEServices->Supports(diag))
    diag->KeyTracing = debugging;
}
```

By using the *DelphiInterface* template (e.g. *_di_IOTAKeyboardDiagnostics*), the Tools API automatically manages the object's lifetime, and you don't need to do anything special in your wizard's destructor.

If your wizard needs to use a specific service often, you can cache the service interface as a data member of your wizard class.

Debugging a wizard

When writing wizards that use the native tools API, you can write code that causes the IDE to crash. It is also possible that you write a wizard that installs but does not act the way you want it to. One of the challenges of working with design-time code is debugging. It's an easy problem to solve, however. Because the wizard is installed in the IDE itself, you simply need to set the package's Host Application to the IDE executable (bin/startdelphi or bin/startbcb) from the Run | Parameters… menu item.

When you want (or need) to debug the package, don't install it. Instead, choose Run | Run from the menu bar. This starts up a new instance of the IDE. In the new instance, install the already-compiled package by choosing Components | Install Package… from the menu bar. Back in the original instance of the IDE, you should now see the telltale blue dots that tell you where you can set breakpoints in the wizard source code. (If not, double-check your compiler options to be sure you enabled debugging; make sure you loaded the right package; and double-check the process modules to make extra sure that you loaded the .bpl file you wanted to load.)

You cannot debug into CLX, or RTL code this way, but you have full debug capabilities for the wizard itself, which might be enough to tell what is going wrong.

Interface version numbers

If you look closely at the declarations of some of the interfaces, such as *IOTAMessageServices*, you will see that they inherit from other interfaces with similar names, such as *IOTAMessageServices50*, which inherits from *IOTAMessageServices40*. This use of version numbers helps insulate your code from changes between releases of the IDE.

The Tools API follows the basic principle that an interface and its unique identifier (GUID) never change. If a new release adds features to an interface, the Tools API declares a new interface that inherits from the old one. The GUID remains the same, attached to the old, unchanged interface. The new interface gets a brand new GUID. Old wizards that use the old GUIDs continue to work.

The Tools API also changes interface names to try to preserve source-code compatibility. To see how this works, it is important to distinguish between the two kinds of interfaces in the Tools API: Borland-implemented and user-implemented. If the IDE implements the interface, the name stays with the most recent version of the interface. The new functionality does not affect existing code. The old interfaces have the old version number appended.

For a user-implemented interface, however, new member functions in the base interface require new functions in your code. Therefore, the name tends to stick with the old interface, and the new interface has a version number tacked onto the end.

For example, consider the message services. A previous release of the IDE introduced a new feature: message groups. Therefore, the basic message services interface required new member functions. These functions were declared in a new interface class, which retained the name *IOTAMessageServices*. The old message services interface was renamed to *IOTAMessageServices50* (for version 5). The GUID of the old *IOTAMessageServices* is the same as the GUID of the new *IOTAMessageServices50* because the member functions are the same.

Consider *IOTAIDENotifier* as an example of a user-implemented interface. Two new overloaded functions were added when a new version of the IDE was released: *AfterCompile* and *BeforeCompile*. Existing code that used *IOTAIDENotifier* did not need to change, but new code that required the new functionality had to be modified to override the new functions inherited from *IOTAIDENotifier50*. The new version of the Tools API did not add any more functions, so the current version to use is *IOTAIDENotifier50*.

The rule of thumb is to use the most-derived class when writing new code. Leave the source code alone if you are merely recompiling an existing wizard under a new release of the IDE.

Working with files and editors

It is important to understand how the Tools API works with files. The main interface is *IOTAModule*. A module represents a set of logically related open files. For example, a single module represents a single unit. The module, in turn, has one or more editors, where each editor represents one file, such as the .pas unit source (Delphi), the .cpp implementation and .h header files (C++), or the form (.xfm) file. The editor interfaces reflect the internal state of the IDE's editors, so a wizard can see the modified code and forms that the user sees, even if the user has not saved any changes.

Using module interfaces

To obtain a module interface, start with the module services (*IOTAModuleServices*). You can query the module services for all open modules, look up a module from a file name or form name, or open a file to obtain its module interface.

There are different kinds of modules for different kinds of files, such as projects, resources, and type libraries. Cast a module interface to a specific kind of module interface to learn whether the module is of that type. For example, one way to obtain the current project group interface is as follows:

Delphi example

```
{ Return the current project group, or nil if there is no project group. }
function CurrentProjectGroup: IOTAProjectGroup;
var
  I: Integer;
  Svc: IOTAModuleServices;
  Module: IOTAModule;
begin
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  for I := 0 to Svc.ModuleCount - 1 do
  begin
    Module := Svc.Modules[I];
    if Supports(Module, IOTAProjectGroup, Result) then
      Exit;
  end;
  Result := nil;
end;
```

C++ example

```
// Return the current project group, or 0 if there is no project group.
_di_IOTAProjectGroup __fastcall CurrentProjectGroup()
{
  _di_IOTAModuleServices svc;
  BorlandIDEServices->Supports(svc);
  for (int i = 0; i < svc->ModuleCount; ++i)
  {
    _di_IOTAModule module = svc->Modules[i];
    _di_IOTAProjectGroup group;
    if (module->Supports(group))
      return group;
  }
  return 0;
}
```

Using editor interfaces

Every module has at least one editor interface. Some modules have several editors, such as a source file editor, and form description (.xfm) file. All editors implement the *IOTAEditor* interface; cast the editor to a specific type to learn what kind of editor it is. For example, to obtain the form editor interface for a unit, you can do the following:

Delphi example

```
{ Return the form editor for a module, or nil if the unit has no form. }
function GetFormEditor(Module: IOTAModule): IOTAFormEditor;
var
  I: Integer;
  Editor: IOTAEditor;
begin
  for I := 0 to Module.ModuleFileCount - 1 do
  begin
    Editor := Module.ModuleFileEditors[I];
    if Supports(Editor, IOTAFormEditor, Result) then
      Exit;
  end;
  Result := nil;
end;
```

C++ example

```
// Return the form editor for a module, or 0 if the unit has no form.
_di_IOTAFormEditor __fastcall GetFormEditor(_di_IOTAModule module)
{
  for (int i = 0; i < module->ModuleFileCount; ++i)
  {
    _di_IOTAEditor editor = module->ModuleFileEditors[i];
    _di_IOTAFormEditor formEditor;
    if (editor->Supports(formEditor))
      return formEditor;
  }
  return 0;
}
```

The editor interfaces give you access to the editor's internal state. You can examine the source code or components that the user is editing, make changes to the source code, components, or properties, change the selection in the source and form editors, and carry out almost any editor action that the end user can perform.

Using a form editor interface, a wizard can access all the components on the form. Each component (including the root form or data module) has an associate *IOTAComponent* interface. A wizard can examine or change most of the component's properties.

Creating forms and projects

The IDE comes with a number of form and project wizards already installed, and you can write your own. The Object Repository lets you create static templates that can be used in a project, but a wizard offers much more power because it is dynamic. The wizard can prompt the user and create different kinds of files depending on the user's responses. This section describes how to write a form or project wizard.

Creating modules

A form or project wizard typically creates one or more new files. Instead of real files, however, it is best to create unnamed, unsaved modules. When the user saves them, the IDE prompts the user for a file name. A wizard uses a creator object to create such modules.

A creator class implements a creator interface, which inherits from *IOTACreator*. The wizard passes a creator object to the module service's *CreateModule* method, and the IDE calls back to the creator object for the parameters it needs to create the module.

For example, a form wizard that creates a new form typically implements *GetExisting* to return false and *GetUnnamed* to return true. This creates a module that has no name (so the user must pick a name before the file can be saved) and is not backed by an existing file (so the user must save the file even if the user does not make any changes). Other methods of the creator tell the IDE what kind of file is being created (e.g., project, unit, or form), provide the contents of the file, or return the form name, ancestor name, and other important information. Additional callbacks let a wizard add modules to a newly created project, or add components to a newly created form.

To create a new file, which is often required in a form or project wizard, you usually need to provide the contents of the new file. To do so, write a new class that implements the *IOTAFile* interface. If your wizard can make do with the default file contents, you can return nil (Delphi) or NULL (C++) from any function that returns *IOTAFile*.

For example, suppose your organization has a standard comment block that must appear at the top of each source file. You could do this with a static template in the Object Repository, but the comment block would need to be updated manually to reflect the author and creation date. Instead, you can use a creator to dynamically fill in the comment block when the file is created.

The first step is to write a wizard that creates new units and forms. Most of the creator's functions return zero, empty strings, or other default values, which tells the Tools API to use its default behavior for creating a new unit or form. Override *GetCreatorType* to inform the Tools API what kind of module to create: a unit or a form. To create a unit, return sUnit. To create a form, return sForm. To simplify the code, use a single class that takes the creator type as an argument to the constructor. Save the creator type in a data member, so that *GetCreatorType* can return its value. Implement *NewImplSource* and *NewIntfSource* to return the desired file contents.

D

Delphi example

```
TCreator = class(TInterfacedObject, IOTAModuleCreator)
public
  constructor Create(const CreatorType: string);

  { IOTAModuleCreator }
  function GetAncestorName: string;
  function GetImplFileName: string;
  function GetIntfFileName: string;
  function GetFormName: string;
  function GetMainForm: Boolean;
```

```
    function GetShowForm: Boolean;
    function GetShowSource: Boolean;
    function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
    function NewImplSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
    function NewIntfSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
    procedure FormCreated(const FormEditor: IOTAFormEditor);

    { IOTACreator }
    function GetCreatorType: string;
    function GetExisting: Boolean;
    function GetFileSystem: string;
    function GetOwner: IOTAModule;
    function GetUnnamed: Boolean;

  private
    FCreatorType: string;
  end;
```

C++ example

```
class PACKAGE TCreator : public IOTAModuleCreator {
public:
  __fastcall TCreator(const AnsiString creator_type)
  : ref_count(0), creator_type(creator_type) {}
  virtual __fastcall ~TCreator();

// IOTAModuleCreator
  virtual AnsiString __fastcall GetAncestorName();
  virtual AnsiString __fastcall GetImplFileName();
  virtual AnsiString __fastcall GetIntfFileName();
  virtual AnsiString __fastcall GetFormName();
  virtual bool __fastcall GetMainForm();
  virtual bool __fastcall GetShowForm();
  virtual bool __fastcall GetShowSource();
  virtual _di_IOTAFile __fastcall NewFormFile(
    const AnsiString FormIdent, const AnsiString AncestorIdent);
  virtual _di_IOTAFile __fastcall NewImplSource(
    const AnsiString ModuleIdent, const AnsiString FormIdent,
    const AnsiString AncestorIdent);
  virtual _di_IOTAFile __fastcall NewIntfSource(
    const AnsiString ModuleIdent, const AnsiString FormIdent,
    const AnsiString AncestorIdent);
  virtual void __fastcall FormCreated(
    const _di_IOTAFormEditor FormEditor);

// IOTACreator
  virtual AnsiString __fastcall GetCreatorType();
  virtual bool __fastcall GetExisting();
  virtual AnsiString __fastcall GetFileSystem();
  virtual _di_IOTAModule __fastcall GetOwner();
  virtual bool __fastcall GetUnnamed();

protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  long ref_count;
  const AnsiString creator_type;
};
```

Most of the members of *TCreator* return nil (Delphi) or zero (C++), or empty strings. The boolean methods return true, except *GetExisting*, which returns false. The most interesting method is *GetOwner*, which returns a pointer to the current project module, or nil or 0 if there is no project. There is no simple way to discover the current project or the current project group. Instead, *GetOwner* must iterate over all open modules. If a project group is found, it must be the only project group open, so *GetOwner* returns its current project. Otherwise, the function returns the first project module it finds, or nil or 0 if no projects are open.

Delphi example

```
function TCreator.GetOwner: IOTAModule;
var
  I: Integer;
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Project: IOTAProject;
  Group: IOTAProjectGroup;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Result := nil;
  for I := 0 to Svc.ModuleCount - 1 do
  begin
    Module := Svc.Modules[I];
    if Supports(Module, IOTAProject, Project) then
    begin
      { Remember the first project module}
      if Result = nil then
        Result := Project;
    end
    else if Supports(Module, IOTAProjectGroup, Group) then
    begin
      { Found the project group, so return its active project}
      Result := Group.ActiveProject;
      Exit;
    end;
  end;
end;
```

C++ example

```
_di_IOTAModule __fastcall TCreator::GetOwner()
{
  // Return the current project.
  _di_IOTAProject result = 0;

  _di_IOTAModuleServices svc = interface_cast<IOTAModuleServices>(BorlandIDEServices);
  for (int i = 0; i < svc->ModuleCount; ++i)
  begin
    _di_IOTAModule module = svc->Modules[i];
    _di_IOTAProject project;
    _di_IOTAProjectGroup group;
    if (module->Supports(project)) {
      // Remember the first project module.
      if (result == 0)
        result = project;
    } else if (module->Supports(group)) {
      // Found the project group, so return its active project.
      result = group->ActiveProject;
      break;
    }
  }
  return result;
}
```

The creator returns nil or 0 from *NewFormSource*, to generate a default form file. The interesting methods are *NewImplSource* and *NewIntfSource*, which create an *IOTAFile* instance that returns the file contents.

The *TFile* class implements the *IOTAFile* interface. It returns –1 as the file age (which means the file does not exist), and returns the file contents as a string. To keep the *TFile* class simple, the creator generates the string, and the *TFile* class simply passes it on.

Delphi example

```
TFile = class(TInterfacedObject, IOTAFile)
public
  constructor Create(const Source: string);
  function GetSource: string;
  function GetAge: TDateTime;
private
  FSource: string;
end;

constructor TFile.Create(const Source: string);
begin
  FSource := Source;
end;

  function TFile.GetSource: string;
begin
  Result := FSource;
end;

  function TFile.GetAge: TDateTime;
begin
  Result := TDateTime(-1);
end;
```

C++ example

```
class TFile : public IOTAFile {
public:
  __fastcall TFile(const AnsiString source);
  virtual __fastcall ~TFile();
  AnsiString __fastcall GetSource();
  System::TDateTime __fastcall GetAge();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  long ref_count;
  AnsiString source;
};

__fastcall TFile::File(const AnsiString source)
: ref_count(0), source(source)
{}

AnsiString __fastcall TFile::GetSource()
{
  return source;
}

System::TDateTime __fastcall TFile::GetAge()
{
  return -1;
}
```

You can store the text for the file contents in a resource to make it easier to modify, but for the sake of simplicity, this example hardcodes the source code in the wizard. The example below generates the source code, assuming there is a form. You can easily add the simpler case of a plain unit. Test *FormIdent*, and if it is empty, create a plain unit; otherwise create a form unit. The basic skeleton for the code is the same as the IDE's default (with the addition of the comments at the top, of course), but you can modify it any way you desire.

Delphi example

```
function TCreator.NewImplSource(
                    const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
var
  FormSource: string;
begin
  FormSource :=
  '{ ----------------------------------------------------------------- ' +   #13#10 +
  '%s - description'+   #13#10 +
  'Copyright © %y Your company, inc.'+   #13#10 +
  'Created on %d'+   #13#10 +
  'By %u'+   #13#10 +
  ' ----------------------------------------------------------------- }' +   #13#10 +
  #13#10;

  return TFile.Create(Format(FormSource, ModuleIdent, FormIdent,
                         AncestorIdent));
}
```

C++ example

```
_di_IOTAFile __fastcall TCreator::NewImplSource(
  const AnsiString ModuleIdent,
  const AnsiString FormIdent,
  const AnsiString AncestorIdent)
{
  const AnsiString form_source =
"/*-----------------------------------------------------------------\n"
" %m - description\n"
" Copyright © %y Your company, inc.\n"
" Created on %d\n"
" By %u\n"
"  ---------------------------------------------------------------*/\n"
"\n"
"#include <clx.h>\n"
"#pragma hdrstop\n"
"\n"
"#include \"%m.h\"\n"
"//-----------------------------------------------------------------\n"
"#pragma package(smart_init)\n"
"#pragma resource \"*.dfm\"\n"
"T%f *%f;\n"
"//-----------------------------------------------------------------\n"
"__fastcall T%m::T%m(TComponent* Owner)\n"
"        : T%a(Owner)\n"
"{\n"
"}\n"
"//----------------------------------------------------------------\n";

  return new File(expand(form_source, ModuleIdent, FormIdent,
                         AncestorIdent));
}
```

Notice that the source code contains strings of the form %m and %y. These are similar in spirit to printf or Format controls, but are expanded by the wizard's expand function. Specifically, %m expands to the module or unit identifier, %f to the form name, and %a to the ancestor name. Notice how the form name is used in the form's type name by inserting a capital T. Some additional format specifiers make it easier to generate the comment block: %d for the date, %u for the user, and %y for the year. (Writing the expand function is unrelated to the Tools API and is left as an exercise for the reader.)

NewIntfSource is similar to *NewImplSource*, but it generates the interface (.h) file.

The final step is to create two form wizards: one uses sUnit as the creator type, and the other uses sForm.Some wizards need to enable or disable the menu items, depending on what else is happening in the IDE. For example, a wizard that checks a project into a source code control system should disable its Check In menu item if no files are open in the IDE. You can add this capability to your wizard by using notifiers, the subject of the next section.

Notifying a wizard of IDE events

An important aspect of writing a well-behaved wizard is to have the wizard respond to IDE events. In particular, any wizard that keeps track of module interfaces must know when the user closes the module, so the wizard can release the interface. To do this, the wizard needs a notifier, which means you must write a notifier class.

All notifier classes implement one or more notifier interfaces. The notifier interfaces define callback methods; the wizard registers a notifier object with the Tools API, and the IDE calls back to the notifier when something important happens.

Every notifier interface inherits from *IOTANotifier*, although not all of its methods are used for a particular notifier. Table 48.3 lists all the notifier interfaces, and gives a brief description of each one.

Table 48.3 Notifier interfaces

Interface	Description
IOTANotifier	Abstract base class for all notifiers
IOTABreakpointNotifier	Triggering or changing a breakpoint in the debugger
IOTADebuggerNotifier	Running a program in the debugger, or adding or deleting breakpoints
IOTAEditLineNotifier	Tracking movements of lines in the source editor
IOTAEditorNotifier	Modifying or saving a source file, or switching files in the editor
IOTAFormNotifier	Saving a form, or modifying the form or any components on the form (or data module)
IOTAIDENotifier	Loading projects, installing packages, and other global IDE events
IOTAMessageNotifier	Adding or removing tabs (message groups) in the message view
IOTAModuleNotifier	Changing, saving, or renaming a module
IOTAProcessModNotifier	Loading a process module in the debugger?
IOTAProcessNotifier	Creating or destroying threads and processes in the debugger
IOTAThreadNotifier	Changing a thread's state in the debugger
IOTAToolsFilterNotifier	Invoking a tools filter

To see how to use notifiers, consider the previous example. Using module creators, the example creates a wizard that adds a comment to each source file. The comment includes the unit's initial name, but the user almost always saves the file under a different name. In that case, it would be a courtesy to the user if the wizard updated the comment to match the file's true name.

To do this, you need a module notifier. The wizard saves the module interface that *CreateModule* returns, and uses it to register a module notifier. The module notifier receives notification when the user modifies the file or saves the file, but these events are not important for this wizard, so the *AfterSave* and related functions all have empty bodies. The important function is *ModuleRenamed*, which the IDE calls when the user saves the file under a new name. The declaration for the module notifier class is shown below:

Delphi example

```
TModuleIdentifier = class(TNotifierObject, IOTAModuleNotifier)
public
  constructor Create(const Module: IOTAModule);
  destructor Destroy; override;
  function CheckOverwrite: Boolean;
  procedure ModuleRenamed(const NewName: string);
  procedure Destroyed;
private
  FModule: IOTAModule;
  FName: string;
  FIndex: Integer;
end;
```

C++ example

```
class TModuleNotifier : public NotifierObject, public IOTAModuleNotifier
{
  typedef NotifierObject inherited;
public:
  __fastcall TModuleNotifier(const _di_IOTAModule module);
  __fastcall ~TModuleNotifier();

// IOTAModuleNotifier
  virtual bool __fastcall CheckOverwrite();
  virtual void __fastcall ModuleRenamed(const AnsiString NewName);

// IOTANotifier
  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  _di_IOTAModule module;
  AnsiString name;        // Remember the module's old name.
  int index;              // Notifier index.
};
```

One way to write a notifier is to have it register itself automatically in its constructor. The destructor unregisters the notifier. In the case of a module notifier, the IDE calls the *Destroyed* method when the user closes the file. In that case, the notifier must unregister itself and release its reference to the module interface. The IDE releases its reference to the notifier, which reduces its reference count to zero and frees the object. Therefore, you need to write the destructor defensively: the notifier might already be unregistered.

Delphi example

```
constructor TModuleNotifier.Create( const Module: IOTAModule);
```

```
begin
  FIndex := -1;
  FModule := Module;
  { Register this notifier. }
  FIndex := Module.AddNotifier(self);
  { Remember the module's old name. }
  FName := ChangeFileExt(ExtractFileName(Module.FileName), '');
end;

destructor TModuleNotifier.Destroy;
begin
  { Unregister the notifier if that hasn't happened already. }
  if Findex >= 0 then
    FModule.RemoveNotifier(FIndex);
end;

procedure TModuleNotifier.Destroyed;
begin
   { The module interface is being destroyed, so clean up the notifier. }
  if Findex >= 0 then
  begin
    { Unregister the notifier. }
    FModule.RemoveNotifier(FIndex);
    FIndex := -1;
  end;
  FModule := nil;
end;
```

C++ example

```
__fastcall TModuleNotifier::ModuleNotifier(const _di_IOTAModule module)
: index(-1), module(module)
{
  // Register this notifier.
  index = module->AddNotifier(this);
  // Remember the module's old name.
  name = ChangeFileExt(ExtractFileName(module->FileName), "");
}

__fastcall TModuleNotifier::~ModuleNotifier()
{
  // Unregister the notifier if that hasn't happened already.
  if (index >= 0)
    module->RemoveNotifier(index);
}

void __fastcall TModuleNotifier::Destroyed()
{
  // The module interface is being destroyed, so clean up the notifier.
  if (index >= 0)
  {
    // Unregister the notifier.
    module->RemoveNotifier(index);
    index = -1;
  }
  module = 0;
}
```

The IDE calls back to the notifier's *ModuleRenamed* function when the user renames the file. The function takes the new name as a parameter, which the wizard uses to update the comment in the file. To edit the source buffer, the wizard uses an edit position interface. The wizard finds the right position, double checks that it found the right text, and replaces that text with the new name.

Delphi example

```
procedure TModuleNotifier.ModuleRenamed(const NewName: string);
var
  ModuleName: string;
  I: Integer;
  Editor: IOTAEditor;
  Buffer: IOTAEditBuffer;
  Pos: IOTAEditPosition;
  Check: string;
begin
  { Get the module name from the new file name. }
  ModuleName := ChangeFileExt(ExtractFileName(NewName), '');
  for I := 0 to FModule.GetModuleFileCount - 1 do
  begin
   { Update every source editor buffer. }
    Editor := FModule.GetModuleFileEditor(I);
    if Supports(Editor, IOTAEditBuffer, Buffer) then
    begin
      Pos := Buffer.GetEditPosition;
      { The module name is on line 2 of the comment.
        Skip leading white space and copy the old module name,
        to double check we have the right spot. }
      Pos.Move(2, 1);
      Pos.MoveCursor(mmSkipWhite or mmSkipRight);
      Check := Pos.RipText('', rfIncludeNumericChars or rfIncludeAlphaChars);
      if Check = FName then
      begin
        Pos.Delete(Length(Check));    // Delete the old name.
        Pos.InsertText(ModuleName);   // Insert the new name.
        FName := ModuleName;          // Remember the new name.
      end;
    end;
  end;
end;
```

C++ example

```
void __fastcall TModuleNotifier::ModuleRenamed(const AnsiString NewName)
{
  // Get the module name from the new file name.
  AnsiString ModuleName = ChangeFileExt(ExtractFileName(NewName), "");
  for (int i = 0; i < module->GetModuleFileCount(); ++i)
  {
    // Update every source editor buffer.
    _di_IOTAEditor editor = module->GetModuleFileEditor(i);
    _di_IOTAEditBuffer buffer;
    if (editor->Supports(buffer))
    {
      _di_IOTAEditPosition pos = buffer->GetEditPosition();
      // The module name is on line 2 of the comment.
      // Skip leading white space and copy the old module name,
      // to double check we have the right spot.
      pos->Move(2, 1);
      pos->MoveCursor(mmSkipWhite | mmSkipRight);
      AnsiString check = pos->RipText("", rfIncludeNumericChars | rfIncludeAlphaChars);
      if (check == name)
      {
        pos->Delete(check.Length());    // Delete the old name.
        pos->InsertText(ModuleName);    // Insert the new name.
        name = ModuleName;              // Remember the new name.
      }
    }
  }
}
```

What if the user inserts additional comments above the module name? In that case, you need to use an edit line notifier to keep track of the line number where the module name sits. To do this, use the *IOTAEditLineNotifier* and *IOTAEditLineTracker* interfaces, which are described in the Online Help.

You need to be cautious when writing notifiers. You must make sure that no notifier outlives its wizard. For example, if the user were to use the wizard to create a new unit, then unload the wizard, there would still be a notifier attached to the unit. The results would be unpredictable, but most likely, the IDE would crash. Thus, the wizard needs to keep track of all of its notifiers, and must unregister every notifier before the wizard is destroyed. On the other hand, if the user closes the file first, the module notifier receives a *Destroyed* notification, which means the notifier must unregister itself and release all references to the module. The notifier must remove itself from the wizard's master notifier list, too.

Below is the final version of the wizard's *Execute* function. It creates the new module, uses the module interface and creates a module notifier, then saves the module notifier in an interface list (*TInterfaceList*).

D Delphi example

```
procedure DocWizard.Execute;
var
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Notifier: IOTAModuleNotifier;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Module := Svc.CreateModule(TCreator.Create(creator_type));
  Notifier := TModuleNotifier.Create(Module);
  list.Add(Notifier);
```

```
end
```

C++ example

```
void __fastcall DocWizard::Execute()
{
  _di_IOTAModuleServices svc;
  BorlandIDEServices->Supports(svc);
  _di_IOTAModule module = svc->CreateModule(new Creator(creator_type));
  _di_IOTAModuleNotifier notifier = new ModuleNotifier(module);
  list->Add(notifier);
}
```

The wizard’s destructor iterates over the interface list and unregisters every notifier in the list. Simply letting the interface list release the interfaces it holds is not sufficient because the IDE also holds the same interfaces. You must tell the IDE to release the notifier interfaces in order to free the notifier objects. In this case, the destructor tricks the notifiers into thinking their modules have been destroyed. In a more complicated situation, you might find it best to write a separate Unregister function for the notifier class.

Delphi example

```
destructor DocWizard.Destroy; override;
var
  Notifier: IOTAModuleNotifier;
  I: Integer;
begin
  { Unregister all the notifiers in the list. }
  for I := list.Count - 1 downto 0 do
  begin
    Supports(list.Items[I], IOTANotifier, Notifier);
    { Pretend the associated object has been destroyed.
      That convinces the notifier to clean itself up. }
    Notifier.Destroyed;
    list.Delete(I);
  end;
  list.Free;
  FItem.Free;
end;
```

C++ example

```
__fastcall DocWizard::~DocWizard()
{
  // Unregister all the notifiers in the list.
  for (int i = list->Count; --i >= 0; )
  {
    _di_IOTANotifier notifier;
    list->Items[i]->Supports(notifier);
    // Pretend the associated object has been destroyed.
    // That convinces the notifier to clean itself up.
    notifier->Destroyed();
    list->Delete(i);
  }
  delete list;
  delete item;
}
```

The rest of the wizard manages the mundane details of registering the wizard, installing menu items, and the like.

Index