# 2장 오브젝트 파스칼 언어

* 주석
* 프로시저와 함수의 새로운 기능
* 변수
* 상수
* 연산자
* 오브젝트 파스칼의 형
* 사용자 정의형
* 강제 형 변환과 자동 형 변환
* 문자열 자원
* 조건 제어문
* 반복문
* 프로시저와 함수
* 범위
* 유닛
* 패키지
* 객체 지향적 프로그래밍
* 델파이 객체의 활용
* 구조적 예외(exception) 처리
* 런타임형 정보(Runtime Type Information, RTTI)

이번 장에서는 델파이의 시각적 요소는 잠시 접어두고, 델파이의 기본을 이루고 있는 오브젝트 파스칼이라는 언어에 대해서 집중적으로 알아보겠다. 우선, 오브젝트 파스칼 언어의 규칙이나 구조 같은 기본적인 부분부터 설명한 후에 클래스나 예외처리 같은 좀더 진보적인 부분에 대해 설명한다. 이 책은 완전 초보자용은 아니므로 독자가 적어도 C나 C++, 비주얼 베이직 같은 고급 언어에 대해 어느 정도 알고 있다고 가정한 상태에서 오브젝트 파스칼과 다른 고급 언어들의 차이점들을 설명한다. 이번 장을 다 읽고 나면 변수나 형, 연산자, 루프, 조건, 예외, 오브젝트 같은 프로그래밍 개념들이 파스칼에서 어떤 식으로 작동하는지, 또 C++나 비주얼 베이직과는 어떻게 다른지 보다 잘 이해하게 될 것이다.

파스칼에 대해서 어느 정도 알고 있는 독자라도 이번 장을 잘 읽어보기 바란다. 이번 장 이후에는 파스칼의 문법과 규약에 대한 설명이 더 이상 나오지 않기 때문이다.

# 주석

우선, 파스칼 코드의 주석(comment) 처리부터 알아보자. 오브젝트 파스칼은 세 가지 형태의 주석을 지원한다. 첫 번째는 중괄호이고, 두 번째는 괄호/별표, 그리고 세 번째는 C++ 형태의 슬래시 두 개로 구성되는 주석이다. 다음은 각 형식의 주석을 예로 든 것이다.

```
{ 중괄호를 이용한 주석 }
(* 괄호와 별표를 이용한 주석 *)
// C++ 형태의 주석
```

첫 번째와 두 번째의 기능은 본질적으로 동일하다. 컴파일러는 주석이 열린 부분부터 주석이 닫히는 부분까지를 주석으로 간주하고 컴파일에서 제외시킨다. C++ 형태의 주석은 슬래시 두 개가 나온 이후부터 그 행의 끝까지만 주석으로 처리한다.



오브젝트 파스칼에서는 본질적으로 주석을 중첩시킬 수 없다. 다른 형태의 주석을 사용하면 중첩시킬 수는 있지만, 권장할 만한 일은 아니다.

```
{ (* 허용됨 *) }
(* { 허용됨 } *)
(* (*  허용되지 않음 *) *)
{ { 허용되지 않음 } }
```

# 프로시저와 함수의 새로운 기능

프로시저나 함수는 프로그래밍 언어를 조금이라도 배운 사람이라면 기본적으로 알고 있는 개념이기 때문에 자세히 설명하지 않겠다. 새로운, 또는 별로 알려지지 않은 부분에 대해서만 짚고 넘어가자.

## 괄호

Delphi5에서 새로운 기능은 아니지만, 오브젝트 파스칼의 경우에 인자가 없는 프로시저나 함수는 괄호를 생략할 수 있다. 다음의 두 구문은 똑같은 결과로 실행된다.

```
Form1.Show;
Form1.Show();
```

괄호를 생략할 수 있다는 것이 뭐 그리 대단한가 하고 반문하는 독자도 있을텐데, 괄호를 입력하려면 사람의 손가락 중 가장 힘이 약한 새끼 손가락으로 <Shift>키를 눌러야 한다. C++나 자바처럼 괄호를 생략할 수 없다면 프로그램이 커지면 커질수록 새끼 손가락이 점점 더 아파올 것이고, 시간도 더욱 많이 걸릴 것이다.

## 오버로딩(overloading)

델파이에서 나타난 함수 오버로딩(overloading) 기능은 원래는 C++의 기능이었다. 함수 오버로딩이란, 동일한 이름의 함수가 서로 다른 형태의 인자들을 처리할 수 있도록 하는 것을 말한다. 함수 오버로딩을 하려면 다음과 같이 overload 지시자를 이용해서 함수를 선언해야 한다.

```
procedure Hello(I: Integer); overload;
procedure Hello(S: string); overload;
procedure Hello(D: Double); overload;
```

클래스에서의 메소드 오버로딩 규칙은 약간 다르며, 그에 대해서는 "메소드 오버로딩" 부분에서 설명하겠다. 함수 오버로딩은 델파이 1 시절부터 개발자들이 요청한 것이었지만, "주의해서 써야 한다"라는 충고를 하고 싶다. 동일한 이름을 가지는 함수나 프로시저가 여러 개인 경우(오버로딩이 아니더라도, 서로 다른 유닛들에 동일한 이름의 프로시저들이 있는 경우도 마찬가지이다), 코드를 읽거나 이해하기가 힘들고, 디버깅하기도 힘들어진다. 따라서, 오버로딩을 사용하기 전에 오버로딩이 꼭 필요한지 충분히 고려해 봐야 한다. 오버로딩을 사용하지 말라는 뜻이 아니라, 과용하지 말라는 뜻이다.

## 기본값 인자

기본값 인자(default value parameter) 역시 델파이의 새로운 기능이다. 기본값 인자 기능이라는 것은 함수나 프로시저를 선언할 때 인자에 기본값을 지정하면 나중에 다른 루틴에서 그 함수나 프로시저를 호출할 때 기본값이 지정된 인자를 생략할 수 있도록 하는 기능을 말한다. 함수나 프로시저의 어떤 인자를 기본값 인자로 선언하려면 다음 예와 같이 인자의 형 뒤에 등호 표시를 하고 기본값을 써주면 된다.

```
procedure HasDefVal(S: string, I: Integer = 0);
```

HasDefVal 프로시저는 두 가지 방식으로 호출할 수 있다. 첫 번째는 두 개의 인자를 모두 지정해 주는 것이다.

```
HasDefVal('hello', 26);
```

두 번째는 인자 S만 지정하고, I에 대해서는 기본값을 쓰도록 하는 것이다.

```
HasDefVal('hello'); //I는 기본값(0)
```

기본값 인자를 사용할 때에는 다음과 같은 규칙을 지켜야 한다.

- 기본값을 가지는 인자는 인자 목록의 제일 뒤에 나타나야 한다. 기본값을 가지는 인자 뒤에 기본값을 가지지 않는 인자가 와서는 안된다.
- 기본값 인자의 형은 서수(Ordinal), 포인터, 집합(Set)이어야 한다.
- 기본값 인자는 값 전달(val) 또는 상수(const)로 전달되어야 한다. 참조(reference)나 out 또는 형 없는(untyped)) 인자로 전달되어서는 안된다.

기본값 인자의 가장 큰 장점은 이전의 호환성을 깨지 않고도 현존하는 함수나 프로시저의 기능을 확장시킬 수 있다는 점이다. 예를 들어서, 두 정수를 더하는 AddInts() 라는 함수가 있다고 하자.

```
function AddInts(I1, I2: Integer): Integer
begin
  Result := I1 + I2;
end;
```

경쟁 회사에서 이 함수에 인자를 세 개 더하는 대적할 만한 제품을 발표했기 때문에 위의 함수도 세 개의 정수를 더할 수 있게 해야 하는 상황이 발생했다. 그러나 그렇게 하면 AddInts를 호출하는 기존 코드들을 모두 일일이 고쳐야 하는데, 상당히 번거로운 일이다. 다행히 다음처럼 기본값 인자를 사용하면 이전의 호환성을 해치지 않고도 간편하게 AddInts 함수의 기능을 확장할 수 있다.

```
function AddInts(I1, I2: Integer; I3: Integer = 0): Integer;
begin
  Result := I1 + I2 + I3;
end;
```

# 변수

C++에 익숙한 독자라면 코드를 짜다가 새로운 변수가 필요할 때마다 변수를 선언하는 것에 익숙해져 있을 것이다. 그러나 오브젝트 파스칼로 이전할 결심을 했다면 그런 습관을 버려야 한다. 오브젝트 파스칼에서는 변수를 선언하는 부분이 따로 있다. 다음의 C++ 코드를 보자.

```
void foo(void)
{
  int x = 1;
  x++;
  int y = 2;
  float f;
  // ... 기타 작업...
}
```

오브젝트 파스칼의 경우, 다음처럼 var 부분에서 미리 변수들을 선언해야 한다.

```
Procedure Foo;
var
  x, y: Integer;
  f: Double;
begin
  x := 1;
  inc(x);
  y := 2;
  //... 기타 작업...
end;
```



C/C++는 대·소문자를 구별하지만, 오브젝트 파스칼은 비주얼 베이직과 마찬가지로 대·소문자를 구분하지 않는다. 대·소문자는 코드를 읽기 쉽게 하기 위한 것일 뿐이다. 식별자 이름이 몇 개의 단어로 이루어져 있다면 대·소문자를 적절히 섞어서 쓰는 것이 좋다. 예를 들어서, 다음과 같은 이름은 매우 이해하기 힘들다.

```
procedure thisprocedurenamemakesnosense
```

다음과 같이 하면 좀더 이해하기 편하다.

```
procedure ThisProcedureNameIsMoreClear;
```

이 책에 사용된 코딩 스타일에 대해서는 6장, "델파이 5 개발자 가이드 소스 코딩 표준 문서"를 참고하기 바란다.

위와 같이 구조를 사용하는 이유는 무엇이고, 어떤 장점이 있는 것일까? 결론적으로 말하면, C++나 비주얼 베이직처럼 흩어진 구조보다는 오브젝트 파스칼과 같은 구조가 좀더 코드를 읽기 쉽게 하며, 유지보수에도 편하고, 디버깅에도 좋다.

오브젝트 파스칼에서는 다음과 같이 동일한 형을 가지는 여러 변수들을 한꺼번에 선언할 수 있다.

```
VarName1, VarName2 : SomeType;
```

오브젝트 파스칼의 변수 선언 규칙은 변수 이름 다음에 콜론(:)이 오고, 그 다음에 형이 오는 것이다. 변수의 선언과 변수의 초기화는 다른 작업이라는 점을 명심하기 바란다.

델파이 2.0부터 전역 변수들의 초기화를 var 블록 안에서 처리할 수 있게 되었다. 다음은 그 예이다.

```
var
  I: Integer = 10;
  S: string = 'Hello world';
  D: Double = 3.141579;
```



위와 같은 초기화 방식은 전역 변수에만 사용할 수 있다. 프로시저나 함수 내부의 지역 변수에는 사용할 수 없다.



델파이 컴파일러는 모든 전역 데이터들을 자동으로 0에 해당하는 값으로 초기화한다. 애플리케이션이 시작되면 모든 정수 형들은 0이 되며, 부동 소수점 형들은 0.0, 그리고 포인터 형들은 nil이 된다. 문자열은 모두 빈 문자열로 초기화된다. 따라서, 초기화할 값이 0에 해당하는 것이라면 소스 코드에서 직접 초기화할 필요는 없다.

# 상수

파스칼에서는 상수를 const절에서 선언한다. C의 const 키워드를 연상하면 된다. 다음은 C에서의 상수 선언이다.

```
const float ADecimalNumber = 3.14;
const int i = 10;
const char * ErrorString = "Danger, Danger, Danger!";
```

C의 상수와 오브젝트 파스칼의 상수 사이의 주된 차이점은 오브젝트 파스칼 상수의 형을 지시할 필요가 없다는 것이다. 이는 비주얼 베이직과 동일한 측면이다. 델파이 컴파일러는 상수에 배정된 값에 따라서 적절한 메모리 공간을 할당하며, Integer와 같은 수량(Scalar) 상수인 경우는 컴파일러가 값을 관리하지만, 메모리에는 결코 할당되지 않는다.

```
const
  ADecimalNumber = 3.14;
  i = 10;
  ErrorString = 'Danger, Danger, Danger!';
```



상수에 메모리가 할당되는 규칙은 다음과 같다. 정수값들은 가장 작은 형태의 데이터형(10은 ShortInt로, 32000은 SmallInt 등)에 할당된다. 영문자/숫자는 char 또는 $H로 정의된 string형으로 할당된다. 부동 소수점값들은 extended형으로 할당되나, 소수점 이하 자릿수가 4 이하임이 명시되어 있으면 Comp형이 된다. Integer나 char 집합들은 그 자체로 저장된다.

코드에서 상수의 형을 직접 지정할 수도 있다. 정확한 연산을 위해서는 형을 직접 지정해 주어야 할 필요가 있기 때문이다.

```
const
  ADecimalNumber: Double = 3.14;
  I: Integer = 10;
  ErrorString: string = 'Danger, Danger, Danger!';
```

오브젝트 파스칼에서는 엄격한 의미의 상수뿐만 아니라, 컴파일시 함수들을 const나 var 선언에 사용할 수 있다. 컴파일 시점 함수들에는 Ord(), Chr(), Trunc(), Round(), High(), Low(), SizeOf() 등이 있다. 다음은 컴파일시 함수들을 사용하는 예이다.

```
type
  A = array[1..2] of Integer;

const
  w: Word = SizeOf(Byte);

var
  i: Integer = 8;
  j: SmallInt = Ord('a');
  L: Longint = Trunc(3.14159);
  x: ShortInt = Round(2.71828);
  B1: Byte = High(A);
  B2: Byte = Low(A);
  C: char = Chr(46);
```




32비트 델파이의 형 관련 상수들은 16비트 델파이 1.0과 다른 방식으로 작동한다. 델파이 1.0에서 선언된 식별자는 상수가 아니라, typed constant라고 하는 미리 초기화된 변수로 작동한다. 그러나 델파이 2 이후에는 형이 명시된 상수들이 진정한 상수가 되도록 호환성을 가지게 되었다. 델파이는 이전 호환성을 충실히 지원하므로 Project | Options 대화상자의 Compiler 페이지나, $J 컴파일러 지시자를 사용하면 델파이 1.0 방식으로 작동하도록 할 수 있긴 하지만, 그리 권장할 만한 일은 아니다. 오브젝트 파스칼 언어 자체가 typed constant의 사용을 폐기하는 쪽으로 발전했기 때문이다.

상수의 값을 실행 코드 안에서 바꾸려고 하면 델파이 컴파일러는 에러 메시지를 낸다. 상수는 읽기 전용이므로 오브젝트 파스칼은 상수들을 애플리케이션의 코드 페이지에 있는 효율적인 저장 장소에 저장해서 성능을 최적화한다. 코드와 데이터 페이지에 대해서는 3장, “Win32 API”를 참고하기 바란다.



오브젝트 파스칼은 C나 C++와는 달리, 전처리기(preprocessor)를 사용하지 않는다. 오브젝트 파스칼에는 매크로라는 개념이 없으며, C의 #define에 해당하는 것이 없다. 오브젝트 파스칼의 $define 지시자는 조건부 컴파일에 사용하는 것일 뿐, 상수를 선언하는 데 사용할 수 없다는 점을 명심하기 바란다. C나 C++ 코드를 포팅할 때 #define으로 선언된 상수는 모두 const를 사용해서 선언해야 한다.

# 연산자

연산자란, 데이터를 조작하는 데 사용되는 기호들이다. 예를 들어서, 숫자형 데이터의 경우는 더하기, 빼기, 곱하기, 나누기 같은 연산자들이 있다. 또한, 배열의 특정 요소를 가리키는 데 사용하는 연산자들도 있다. 이번에는 오브젝트 파스칼의 연산자들에 대해서, 그리고 C나 비주얼 베이직 연산자들과의 차이점에 대해서 알아보자.

## 할당 연산자

파스칼을 처음 접하는 사람이라면 델파이의 할당 연산자에 익숙해지기가 쉽지 않을 것이다. 변수에 값을 넣을 때에는 :=을 사용한다. C나 비주얼 베이직의 =에 비해, 한 글자가 더 많은 셈이다. 변수에 값을 넣는 것은 할당(assignment)이라고 한다. 다음은 그 예이다.

```
Number1 := 5;
```

위의 코드를 말로 하면 “Number1에 값 5를 넣는다” 또는 “Number1에 값 5를 할당한다”가 된다.

## 비교 연산자

비주얼 베이직 경험자라면 델파이의 비교 연산자가 비주얼 베이직의 것과 동일하다는 점을 깨닫게 될 것이다. 비교 연산자는 대부분의 프로그래밍 언어에서 비슷 비슷하며, 이해하기 어려운 부분도 아니므로 언급할 필요가 있는 부분만 간단히 짚고 넘어가겠다.

두 수식에서 값이 같은지 비교할 때 오브젝트 파스칼은 =을 사용한다. C의 ==과 같은 의미이다. 다음은 C에서 ==을 사용한 예이다.

```
if (x==y)
```

위의 코드를 오브젝트 파스칼로 옮기면 다음과 같다.

```
if x = y
```



오브젝트 파스칼에서 =은 비교 연산자이며, 변수에 값을 넣을 때에는 :=을 사용해야 한다는 점을 잊지 말기 바란다.

델파이의 부등호는 <>이며, C의 !=에 해당한다. 다음은 두 표현식이 다른지를 판단하는 코드이다.

```
if x <> y then DoSomething
```

## 논리 연산자

C의 경우에 논리곱이나 논리합은 &&, ||를 사용하지만, 오브젝트 파스칼은 and와 or를 사용한다. and나 or는 주로 조건문이나 루프에서 많이 사용된다.

```
if (조건식 1) and (조건식 2) then
  DoSomething;

while (조건식 1) or (조건식 2) do
  DoSomething;
```

파스칼의 부정 연산자는 not이다. 참, 거짓을 전환하는 데 사용하며, C의 ! 연산자에 해당한다. 다음은 if문에서 not을 사용하는 예이다.

```
if not (조건식) then (어떤 처리); // 조건이 거짓이면...
```

표 2.1은 파스칼과 C, 비주얼 베이직의 연산자들을 비교한 것이다.

표 2.1 할당, 비교, 논리 연산자

| 연산자 | 파스칼 | C | 비주얼 베이직 |
|---|---|---|---|
| 배정 | := | = | = |
| 등호 | = | == | = |
| 부등호 | <> | != | <> |
| 미만 | < | < | < |
| 초과 | > | > | > |
| 이하 | <= | <= | <= |
| 이상 | >= | >= | >= |
| 논리곱 | and | && | And |
| 논리합 | or | \|\| | Or |
| 부정 | not | ! | Not |

## 사칙 연산자

오브젝트 파스칼의 사칙 연산자들은 C/C++와 비주얼 베이직의 것들과 거의 동일하다. 표 2.2는 오브젝트 파스칼, C, 비주얼 베이직의 사칙 연산자들이다.

표 2.2 사칙 연산자

| 연산자 | 파스칼 | C | 비주얼 베이직 |
|---|---|---|---|
| 더하기 | + | + | + |
| 빼기 | - | - | - |
| 곱하기 | * | * | * |
| 부동 소수점 나누기 | / | / | / |
| 정수 나누기 | div | / | / |
| 나머지 | mod | % | mod |

오브젝트 파스칼은 부동 소수점과 정수 나누기에 개별적인 연산자를 사용한다는 점에서 다른 언어들과 구별된다. div 연산자는 두 정수 표현식을 나눈 값에서 소수점 부분을 자동으로 잘라낸다.



형에 따라 나누기 연산자를 정확하게 사용하는 것이 중요하다. 오브젝트 파스칼 컴파일러는 두 개의 부동 소수점값을 div로 나누거나, 두 개의 정수를 /로 나누는 경우에 에러를 발생시킨다. 다음 코드를 보자.

```
var
  i: Integer;
  r: Real;
begin
  i := 4 / 3;          //컴파일러 에러가 발생한다.
  f := 3.4 div 2.3;    //역시 컴파일러 에러가 발생한다.
end;
```

다른 프로그래밍 언어들은 대부분 정수 나누기와 부동 소수점 나누기를 구분하지 않는다. 대신, 그런 언어들은 항상 부동 소수점 나누기를 수행하며, 필요에 따라 결과값을 정수로 변환한다. 따라서, 성능의 감소가 일어날 소지가 있다. 파스칼의 div는 성능도 빠르고, 코드를 읽기도 쉽다.

## 비트 단위 연산자

비트 단위 연산자란, 주어진 변수의 개별 비트들을 다루는 연산자이다. 비트들을 왼쪽, 오른쪽으로 시프트시키거나, 비트 단위의 논리곱(and), 합(or), 부정(not), 배타적 논리합(xor)을 수행하는 연산자들이다. 왼쪽 이동과 오른쪽 시프트 연산자는 각각 shl과 shr이며, C의 <<와 >>에 해당한다. 나머지 연산자들은 문자 그대로 and, or, not, xor이다. 표 2.3에 비트 단위 연산자들이 나와 있다.

표 2.3 비트 단위 연산자

| 연산자 | 파스칼 | C | 비주얼 베이직 |
|---|---|---|---|
| 논리곱 | and | && | And |
| 부정 | not | ~ | Not |
| 논리합 | or | \|\| | Or |
| 왼쪽 시프트 | shl | << | 없음 |
| 오른쪽 시프트 | shr | >> | 없음 |

## 증가/감소 연산자

변수의 값을 1씩 증가시키거나 감소시킬 때 증가/감소 전용 연산자를 사용하면 컴파일이 최적화된다. 파스칼은 C의 ++나 --처럼 연산자를 사용하는 것이 아니라, Inc()와 Dec()라는 프로시저를 이용한다. 물론, 컴파일 최적화 효과는 동일하다고 할 수 있다(하나의 기계어 명령으로 컴파일된다).

Inc()나 Dec()는 두 가지 방식으로 사용할 수 있다. 다음과 같이 변수 이름만을 인자로 넣은 경우에는 변수 variable의 값을 1씩 증가 또는 감소시킨다.

```
Inc(variable);
Dec(variable);
```

변수 이름 이외에, 증가/감소치를 직접 인자로 넣을 수도 있다. 다음 코드는 variable의 값을 3씩 증가 또는 감소시킨다.

```
Inc(variable, 3);
Dec(variable, 3);
```

표 2.4는 각 언어의 증가/감소 연산자이다.

표 2.4 증가/감소 연산자

| 연산자 | 파스칼 | C | 비주얼 베이직 |
|---|---|---|---|
| 증가 | Inc() | ++ | 없음 |
| 감소 | Dec() | -- | 없음 |



컴파일 최적화가 활성화되어 있어도 Inc()와 Dec() 프로시저가 variable := variable + 1 형식과 동일한 기계어 코드를 생성하는 경우가 있으므로 애플리케이션의 수행 성능이 큰 문제가 되지 않는다면 변수의 값을 증가 또는 감소시킬 때 그냥 자신에게 편한 방식을 사용하면 된다.

# 오브젝트 파스칼의 형

오브젝트 파스칼의 큰 특징 중 하나가 엄격한 형검사이다. 또는, 이것을 typesafe라고 한다. 어떤 함수나 프로시저에 인자를 넘겨줄 때 그 인자의 형은 함수나 프로시저를 선언할 때 정의한 형과 반드시 일치해야 한다. C 프로그래머들에게 익숙한 '포인터 변환이 의심스럽다'라는 경고는 델파이에서는 결코 나타나지 않는다. 오브젝트 파스칼에서는 함수나 프로시저를 선언할 때 정의한 형 이외의 형을 가지는 값을 결코 받아들이지 않기 때문이다(물론, Pointer형을 사용하면 어떤 형태의 포인터도 받아들일 수 있다). 기본적으로, 파스칼의 엄격한 형 검사는 코드의 안전성에 크게 도움이 된다. 100V용 모니터를 220V에 꽂으면 안된다는 것과 마찬가지 의미이다.

## 형 비교

델파이의 기본 형들은 C나 비주얼 베이직과 크게 다르지 않다. 표 2.5는 오브젝트 파스칼의 기본 형들과 C/C++와 비주얼 베이직의 기본 형들을 비교한 것이다. C++나 비주얼 베이직으로 만든 DLL이나 OBJ(목적 코드) 파일을 델파이에서 사용할 일이 있다면 다음 표를 복사해서 벽에 붙여 놓거나, 책갈피를 해두는 것이 좋겠다.

표 2.5 파스칼과 C/C++, 비주얼 베이직의 32비트 형 비교

| 변수의 표현 형식 | 파스칼 | C/C++ | 비주얼 베이직 |
|---|---|---|---|
| 8비트 부호 있는 정수 | ShortInt | char | 없음 |
| 8비트 부호 없는 정수 | Byte | BYTE, unsigned short | 없음 |
| 16비트 부호 있는 정수 | SmallInt | short | Short |
| 16비트 부호 없는 정수 | Word | unsigned short | 없음 |
| 32비트 부호 있는 정수 | Integer, LongInt | int, long | Integer |
| 32비트 부호 없는 정수 | Cardinal, LongWord | unsigned long | 없음 |
| 64비트 부호 있는 정수 | Int64 | __int64 | 없음 |
| 4바이트 부동 소수점 | Single | float | Single |
| 6바이트 부동 소수점 | Real48 | 없음 | 없음 |
| 8바이트 부동 소수점 | Double | double | Double |
| 10바이트 부동 소수점 | Extended | long double | 없음 |
| 64비트 통화 | currency | 없음 | Currency |
| 16바이트 Variant형 | Variant, OleVariant, TVarData | VARIANT, Variant, OleVariant | (기본) |
| 1바이트 문자 | Char | char | 없음 |
| 2바이트 문자 | WideChar | WCHAR | |
| 고정 길이 문자열 | ShortString | 없음 | 없음 |
| 동적 문자열 | AnsiString | AnsiString | $ |
| 널 종료 문자열 | PChar | char * | 없음 |
| 널 종료 와이드 문자열 | PWideChar | LPCWSTR | 없음 |
| 동적 2바이트 문자열 | WideString | WideString | 없음 |
| 1바이트 부울값 | Boolean, ByteBool | (모든 1바이트형) | 없음 |
| 2바이트 부울값 | WordBool | (모든 2바이트형) | Boolean |
| 4바이트 부울값 | BOOL, LongBool | BOOL | 없음 |

※오브젝트 파스칼형을 지원하기 위한 Borland C++ Builder 클래스임.



델파이 1.0의 16비트 코드를 델파이 4, 5로 포팅할 일이 있다면 Integer와 Cardinal형이 16비트에서 32비트로 바뀌었다는 점을 명심하기 바란다. 좀더 정확히 이야기하자면, 델파이 2와 3에서 Cardinal형은 계산의 정확도를 보장하기 위해 부호 없는 31비트 정수로 처리된다(델파이 2와 3에는 부호 없는 32비트 정수형이 없기 때문에 정수 연산의 결과에 신경을 써야 한다). 델파이 4, 5에서는 Cardinal이 부호 없는 32비트 정수로 처리된다.



델파이 1, 2, 3에서 Real형은 6바이트 부동 소수점값으로, 다른 언어에는 없는 파스칼 고유의 데이터형이다. 그러나 델파이 4, 5에서는 Real이 Double과 같은 의미를 가지며, 기존의 6바이트 부동 소수점값은 Real48이라는 형으로 바뀌었다. 다만, {$REALCOMPATIBILITY ON} 지시자를 써주면 이전처럼 Real이 6바이트 부동 소수점값으로 처리된다.

## 문자형

델파이는 세 종류의 문자형을 제공한다.

- AnsiChar — 크기가 1바이트인 ANSI 문자. 보통, 문자(Character)라고 하면 이것을 의미한다.
- WideChar — 크기가 2바이트인 유니코드 문자
- Char — 현재로서는 AnsiChar와 동일하게 취급되긴 하지만, 델파이 버전에 따라 다른 결과값을 가져올 수 있다.

문자가 반드시 1바이트를 뜻하던 시절은 이미 오래 전에 지나갔다. 따라서, 문자의 길이가 필요한 경우에는 SizeOf() 함수를 사용해야 정확한 결과를 얻을 수 있다.



SizeOf() 함수는 형이나 인스턴스의 크기를 바이트 단위로 그 결과를 리턴시켜 준다.

## 여러 가지 문자열 형

문자열(String)이란, 말 그대로 일련의 문자들을 뜻하는 형이다. 각 언어마다 문자열을 저장하고 표현하는 방식이 다르다. 파스칼 역시 여러 가지 문자열 형들이 있다.

- AnsiString - 오브젝트 파스칼의 기본 문자열 형. AnsiChar 문자들로 이루어지며, 메모리의 제한을 고려하지 않는다면 길이의 제한은 없다. 널 종료 문자열(null-terminated string)과 호환된다.
- ShortString - 델파이 1.0에 대한 호환성을 위한 문자열이다. 최대 길이는 255자로 제한되어 있다.
- WideString - AnsiString과 동일한 기능을 가지나, WideChar 문자로 이루어져 있다.
- PChar - 널 종료 Char 문자열에 대한 포인터. C의 char * 또는 lpstr과 같다.
- PAnsiChar - 널 종료 AnsiChar 문자열에 대한 포인터
- PWideChar - 널 종료 WideChar 문자열에 대한 포인터

다음 예처럼 코드에서 어떤 변수를 string으로 선언하면 기본적으로 컴파일러는 AnsiString으로 처리한다.

```
var
  S: string; // S는 AnsiString이다.
```

컴파일러 지시자 $H를 사용하면 string의 실제 형을 변경할 수 있다. $H-를 써주면 그 이후부터 string으로 선언된 모든 변수들을 ShortString으로 취급하며, $H+를 써주면 AnsiString으로 취급한다. 다음은 그 예이다.

```
var
  {$H-}
  S1: string; //S1은 ShortString
  {$H+}
  S2: string; //S2는 AnsiString
```

단, 문자열의 크기를 정한 경우에는(최대 크기는 255 문자까지 가능) $H에 상관 없이 항상 ShortString이 된다.

```
var
  S: string[63];  //최대 크기가 63 문자인 ShortString
```

### AnsiString형

"긴 문자열"인 AnsiString은 델파이 2.0부터 도입되었다. 델파이 1.0의 사용자들이 늘어나면서 255자의 크기 제한 없이 간편하게 사용할 수 있는 문자열 데이터형을 만들어달라는 요구도 높아졌었는데, AnsiString은 그러한 요구에 대한 응답이다.

AnsiString은 이전 버전과 거의 똑같은 인터페이스를 유지할지라도, 동적으로 할당하고 가비지 컬렉션(Garbage Collection) 처리를 해준다. 이러한 이유 때문에 AnsiString을 수명 관리(Lifetime-managed)형이라고도 한다. 또한, 필요에 따라 문자열의 임시 저장 공간까지 자동으로 할당해 주므로 C/C++에서처럼 문자열의 메모리 할당 때문에 골치를 썩일 필요가 없다. 게다가, AnsiString은 항상 자동적으로 널 종료 처리가 되므로 Win32 API에서 쓰이는 널 종료 문자열과 자연스럽게 호환된다. AnsiString의 실제 모습은 힙 메모리에 있는 문자열 구조체에 대한 포인터이다. 그림 2.1은 AnsiString이 어떻게 메모리에 배치되는지를 나타낸 것이다.

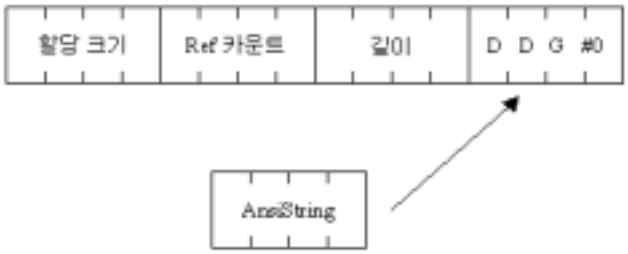



**그림 2.1**
메모리 안의 AnsiString



볼랜드는 아직 긴 문자열형의 완전한 내부 형식에 대해서는 공식적으로 발표하지 않았다. 따라서, 긴 문자열의 내부 형식은 이후 버전의 델파이에서 바뀔 수도 있다. 이 장에 나온 정보는 AnsiString의 작동 방식을 설명하기 위한 것일 뿐이며, AnsiString의 내부 구조에 의존하는 코드는 피하는 것이 좋다.
델파이 1의 코드를 델파이 2로 옮길 때 문자열의 내부 구조에 의존하는(예를 들어서, 문자열의 0번째 문자를 문자열의 길이로 사용한다든지) 코드는 반드시 델파이 2에 맞게 수정해야 한다. 그렇지 않고 평범하게 문자열을 다뤘다면 별 문제 없이 포팅할 수 있을 것이다.

그림 2.1에 나타난 것처럼 AnsiString에는 참조 횟수가 있다. 서로 다른 문자열들이 동일한 메모리 공간을 가리킬 수도 있다는 의미이다. 따라서, 문자열을 복사할 때 문자열의 모든 문자를 복사하는 대신 포인터만 복사하면 되므로 속도가 매우 빨라진다. 둘 이상의 AnsiString이 동일한 문자열(문자들이 있는 메모리 공간)을 가리키는 경우, 델파이의 메모리 관리자는 '쓸 때 복사(copy-on-write)' 기법을 사용한다. 다시 말하면, 문자열이 변경되지 않은 상태에서는 실제로 문자열을 복사하지 않고 있다가 문자열이 변경되면(즉, 동일한 문자열 공간을 공유할 수 없게 되면) 새로운 메모리 공간을 할당해서 변경된 문자열을 복사한다는 것이다. 다음은 그러한 기법을 보여주는 예이다.

```
var
  S1, S2: string;
begin
  // S1에 문자열을 할당한다. S1의 참조 횟수는 1이다.
  S1 := '어쩌구 저쩌구... ';
  S2 := S1;    //S2는 S1을 가리킨다. S1의 참조 횟수는 2가 된다.
  // S2가 변경되면 델파이의 메모리 관리자는 S2만의 메모리 공간을 새로
  // 할당해서 변경된 문자열을 복사한다. 따라서, S1의 참조 횟수는 다시 1이
  // 된다.
  S2 := S2 + '완전히 다름!';
```

## 수명 관리(Lifetime-managed)형

AnsiString 이외에도, 델파이에는 몇 가지 수명 관리(lifetime-managed)형이 있다. 동적 배열이나 WideString, Variant, OleVariant, interface, dispinterface가 수명 관리형이다. 이 형들에 대해서는 이후에 설명하겠다. 중요한 것은 수명 관리형이란 정확하게 무엇이며, 어떻게 작동하는가이다.

수명 관리 데이터형은 "가비지 컬렉션" 형이라고도 하며, 사용중에는 일정한 자원을 차지하고 있다가 자신의 사용 범위에서 벗어나면 자동으로 자원이 해제되는 형이다. 예를 들어서, AnsiString은 사용중에는 문자열이 담긴 메모리 공간을 차지하고 있으며, 범위를 벗어나면 그 메모리 공간이 해제된다.

전역 변수일 경우에는 처리 과정이 상당히 간단하다. 델파이 컴파일러는 애플리케이션의 종료 코드 부분에 각각의 수명 관리 전역 변수의 해제 코드를 삽입한다. 애플리케이션이 시작될 때 모든 전역 데이터는 숫자 0이나 nil, 빈 문자열 등 "아직 사용되지 않았음"에 해당되는 값으로 초기화된다. 따라서, 종료 코드에서는 실제로 사용되지 않았던 자원을 해제하지는 않는다.

지역 변수의 경우는 좀더 복잡한 과정을 거친다. 우선, 컴파일러는 함수나 프로시저의 진입부에 각 변수를 0에 해당하는 값으로 초기화하는 코드를 삽입한다. 그리고 함수 전체를 감싸는 try...finally 예외 처리 블록을 삽입한다. 마지막으로, 컴파일러는 finally 블록에 모든 수명 관리 변수들을 비우는 코드를 집어넣는다(예외 처리에 대해서는 "구조적 예외 처리" 부분에서 설명하겠다).

다음의 코드를 보자.

```
procedure Foo;
var
  S: string;
begin
  // 프로시저의 몸체
  // 여기서 S를 사용한다.
end;
```

이것은 개발자가 직접 작성한 코드이다. 실제로 컴파일되는 코드, 즉 컴파일러가 삽입하는 코드가 추가된 모습은 다음과 같다.

```
procedure Foo;
var
  S: string;
begin
  S := '';
  try
  // 프로시저의 몸체
  // 여기서 S를 사용한다.
  finally
  // S를 해제한다.
  end;
end;
```

### 문자열의 연산

두 문자열을 합칠 때에는 연산자 +를 사용하거나 Concat() 함수를 사용한다. Concat()는 이전 버전 호환성을 위한 것이며, + 연산자를 사용하는 것이 코드를 이해하는 데 더 보기 좋다. 다음은 +와 Concat()를 사용하는 예이다.

```
{ + 사용 예}
var
  S, S2: string;
begin
  S := '2002년 ';
  S2 := '월드컵';
  S := S + S2; { 2002년 월드컵 }
end.

{ Concat() 사용 예}
var
  S, S2: string;
begin
  S := '2002년 ';
  S2 := '월드컵';
  S := Concat(S, S2);  { 2002년 월드컵 }
end.
```



오브젝트 파스칼에서 문자열값은 항상 작은 따옴표('문자열')를 사용해야 한다.



언어 자체와 컴파일러는 다른 것이다. Concat()는 오브젝트 파스칼의 명령어가 아니라, 컴파일러가 제공하는 함수이다(ReadLn()이나 WriteLn()도 마찬가지이다). Concat()는 고정되지 않은 개수의 인자를 가지며, 생략 가능한 인자가 포함되어 있기 때문에 오브젝트 파스칼 언어로는 정의할 수 없다. 오브젝트 파스칼 언어 자체의 기능을 확장하기 위해 언어 자체의 명령어처럼 쓰이는 '도우미' 함수들은 모두 System 유닛에 정의되어 있다. 그런 함수들은 파스칼 언어의 규칙을 만족시키기 위해서 어셈블리 언어로 구현된 것들이다.

문자열 조작을 위한 '도우미' 함수나 프로시저들 이외에도, 문자열을 좀더 편리하게 다루기 위한 여러 함수나 프로시저들이 SysUtils 유닛에 정의되어 있다. 델파이 온라인 도움말에서 "String-handling routines(Pascal-style)" 부분을 참고하기 바란다.
또한, 이 책 부록 CD의 \Code\Utils 디렉토리의 StrUtils.pas에도 유용한 문자열 조작 함수/프로시저들이 있으니 참고하기 바란다.

### 길이와 메모리 할당

AnsiString이 선언된 시점에서는 길이도 없고, 메모리도 할당되지 않는다. 문자열값이나 다른 문자열 변수를 배정하거나, SetLength() 프로시저를 사용한 시점에서 실제 메모리 공간이 할당된다. 다음 코드를 보자.

```
var
  S: string;         // 선언된 시점에서는 길이가 없다.
begin
  S := 'Doh!';       // 문자열값이 들어갈 수 있는 길이의 공간이 할당된다.
  { 또는 }
  S := OtherString; // OtherString의 참조 횟수가 1 증가한다(OtherString이
                    // 실제 문자열 공간을 가리키고 있는 경우).
  { 또는 }
  SetLength(S, 4);  // 적어도 4개의 문자를 넣을 수 있는 공간이 할당된다.
end;
```

AnsiString은 안시 문자(AnsiChar)의 배열로 사용할 수도 있지만, 배열 인덱스가 문자열 길이를 초과해서는 안된다. 예를 들어서, 다음과 같은 코드는 에러를 발생시킨다.

```
var
  S: string;
begin
  S[1] := 'a'; // S에 메모리가 할당되지 않은 상태이므로 작동하지 않는다.
end;
```

그러나 다음 코드는 제대로 작동한다.

```
var
  S: string;
begin
  SetLength(S, 1);
  S[1] := 'a'; // S에 메모리가 할당된 상태이므로 제대로 작동한다.
end;
```

### Win32와의 호환성

앞에서도 말했듯이 AnsiString은 항상 널 종료이므로 널 종료 문자열과 호환된다. 따라서, Win32 API 함수들처럼 PChar형 문자열이 필요한 경우에 AnsiString을 사용할 수 있다. 단, PChar로 형 변환을 해주어야 한다(형 변환에 대해서는 "자동 형 변환과 강제 형 변환" 부분에서 설명하겠다). 다음 코드는 PChar와 버퍼 길이를 인자로 받아들이는 GetWindowsDirectory()라는 Win32 API 함수를 사용하는 예이다.

```
var
  S: string;
begin
  SetLength(S, 256);   //  우선, 문자열의 공간을 할당해야 한다.
  // 함수를 호출하면 S에 디렉토리 이름이 채워진다.
  GetWindowsDirectory(PChar(S), 256);
end;
```

PChar를 사용해야 하는 함수나 프로시저에 AnsiString을 사용한 후에는 문자열의 끝을 알리는 널 문자를 포함한 문자열의 길이를 직접 설정해 주어야 한다. CD에 포함된 StrUtils 유닛에 RealizeLength()라는 함수가 있는데, 실제 코드는 다음과 같다.

```
procedure RealizeLength(var S: string);
begin
  SetLength(S, StrLen(PChar(S)));
end;
```

PChar형을 다시 변환할 필요가 있으면 다음처럼 RealizeLength() 함수를 호출하기만 하면 된다.

```
var
  S: string;
begin
  SetLength(S, 256);   // 우선, 문자열의 공간을 할당해야 한다.
  GetWindowsDirectory(PChar(S), 256);   // 함수를 호출하면
                                        // S에 디렉토리 이름이 채워진다.
  RealizeLength(S);    // S의 길이를 널 문자가 포함된 길이로 갱신한다.
```



string형을 PChar로 형 변환할 때는 주의하기 바란다. 문자열은 자신의 사용 범위를 벗어날 때 자동적으로 가비지 컬렉션이 일어나므로 P의 범위가 Str의 범위보다 넓은 상황에서 P := PChar(Str);처럼 할당하는 경우, Str의 사용 범위가 끝나면 P는 아무 것도 없는 메모리 공간을 가리키게 된다.

### 포팅 관련 문제

다음은 16비트 델파이 1.0 애플리케이션을 포팅(porting, 애플리케이션을 다른 버전의 개발 도구나 다른 운영체제에 맞게 변환하는 것)할 때 AnsiString과 관련된 주의 사항들이다.

- PString(ShortString을 가리키는 포인터)형을 사용한 부분을 string으로 바꾸어야 한다. AnsiString 자체가 문자열에 대한 포인터이기 때문이다.
- 델파이 1과는 달리, 문자열의 0번째(대부분의 프로그래밍 언어에서 배열의 인덱스는 0부터 시작한다. 즉, 0번째 요소는 첫 번째 요소를 뜻한다) 요소가 그 문자열의 길이를 의미하지 않는다는 점을 명심해야 한다. 문자열의 길이를 얻으려면 Length() 함수를, 문자열의 길이를 설정하려면 SetLength() 프로시저를 사용해야 한다.
- PChar 관련 작업에서 더 이상 StrPas()나 StrPCopy()를 사용할 필요가 없다. 앞에서 설명했듯이 AnsiString을 PChar로 직접 형 변환시킬 수 있다. PChar의 내용을 AnsiString으로 복사하려면 다음처럼 직접 할당하면 된다.

```
StringVar := PCharVar;
```

주의

다시 한 번 말하지만, 문자열의 0번째 요소로 문자열의 길이를 설정하면 안된다. 반드시 SetLength() 프로시저를 사용해야 한다. 16비트 델파이 1.0 코드를 32비트로 포팅할 때 반드시 주의해야 할 점이다.

## ShortString형

델파이 고수라면 ShortString형이 델파이 1.0의 string형이라는 점을 알고 있을 것이다. ShortString은 흔히 파스칼 문자열 또는 길이-바이트 문자열이라고 부른다. 다시 한 번 말하지만, string이 AnsiString을 의미하도록 할 것인지, ShortString을 의미하도록 할 것인지는 $H 지시자로 변경할 수 있다.

메모리 안에서 문자열은 문자들의 배열로 저장되는데, 0번째 요소에 그 문자열의 길이가 들어 있다. ShortString의 최대 길이는 기본적으로 256바이트이다. 즉, 길이가 들어 있는 0번째 요소를 제외하면 최대 255 문자를 담을 수 있는 것이다(255 문자 + 1 길이 바이트 = 256). AnsiString과 마찬가지로, ShortString 역시 컴파일러가 알아서 문자열의 임시 저장 장소를 할당해 주므로 C에서처럼 메모리 할당 문제로 골치를 썩일 필요가 없다.

그림 2.2는 ShortString이 메모리에 배치되는 모습을 표현한 것이다.




**그림 2.2**
메모리 안의 ShortString

ShortString은 다음과 같이 선언하고 초기화한다.

```
var
  S: ShortString;
begin
  S := 'Bob the Cat';
end.
```

다음과 같이 문자열의 길이를 256 이하로 직접 지정하면 string형은 항상 ShortString으로 처리된다.

```
var
  S: string[45];   { 45 문자를 가질 수 있는 ShortString }
begin
  S := '이 문자열은 45자 이하이어야 한다.'; //ShortString에서 한글 한 글자는
// 길이가 2란 점을 주의해야 한다.
end.
```

위의 코드에서 string은 $H 지시자에 상관 없이 ShortString으로 처리된다. [] 안에 지정할 수 있는 길이의 최대값은 255이다.

지정된 길이 이상의 문자열을 배정한 경우에 에러가 발생하지는 않지만, 길이 이상의 문자들은 잘려나간다. 예를 들어서, string[6]으로 선언한 변수에 '상당히_긴_문자열'이라는 문자열을 넣으려고 하면 '상당히'까지만 들어간다(ShortString에서 한글 한 글자의 길이는 2이다).

배열 인덱스를 이용해서 ShortString의 특정 문자를 얻거나 설정할 때 배열 인덱스가 string 선언에서 지정된 길이를 벗어나면 잘못된 결과를 얻거나 메모리가 깨지게 된다. 예를 들어서, 다음과 같이 변수를 선언했다고 하자.

```
var
  Str: string[8];
```

그런 다음, 다음과 같이 문자열의 10번째 요소를 설정하려고 하면 다른 변수가 사용하는 메모리 영역을 침범하는 결과를 낳을 수 있다.

```
var
  Str: string[8];
  i: Integer;

begin
  i := 10;
  Str[i] := 's'; // 메모리 충돌.
end.
```

실행 시점에서 이러한 오류를 잡아낼 수 있도록 컴파일하려면 Project | Options 대화상자의 Compiler 탭의 Runtime Errors 그룹 중에서 Range Checking을 선택하면 된다.



범위 검사(Range Checking)는 예기치 못한 에러를 처리하는 데 도움이 되긴 하지만, 애플리케이션의 성능이 조금 떨어진다. 일반적으로, 개발과 디버깅 단계에서는 범위 검사를 사용하고, 에러가 없음이 확인된 후에는 범위 검사를 사용하지 않는다.

AnsiString과는 달리, ShortString은 널 종료 문자열과 자동적으로 호환되지 않는다. 따라서, ShortString을 Win32 API 함수에 사용할 때에는 사전 처리 작업이 필요하다. 다음은 CD의 StrUtils.Pas에 있는 ShortStringAsPChar() 함수이다.

```
function ShortStringAsPChar(var S: ShortString): PChar;
{ 이 함수는 PChar형이 필요한 Win32 API 함수에서 문자열을 사용할 수 있도록
문자열 끝에 널 문자를 추가한다. }
{ 문자열의 길이가 254자 이상이면 254자 이후는 잘린다.}
begin
  if Length(S) = High(S) then Dec(S[0]); { 길이가 너무 길면 잘라낸다.}
  S[Ord(Length(S)) + 1] := #0;        { 문자열의 끝에 널값을 넣는다.}
  Result := @S[1];                 {PChar형의 문자열을 돌려준다.}
end;
```



Win32 API 함수나 프로시저는 널 종료 문자열을 사용한다. ShortString형의 문자열을 그대로 API 함수에 사용하면 컴파일이 진행되지 않는다. API를 사용할 때에는 긴 문자열(AnsiString 등)을 사용하는 것이 여러 모로 편리하다.

### WideString형

WideString형도 AnsiString과 마찬가지로, 수명 관리형이다. 두 형 모두 델파이에서 동적 할당과 가비지 컬렉션을 처리하며, 두 형의 내용을 서로 교환할 수 있다. 그러나 WideString은 다음과 같은 세 가지 측면에서 AnsiString과 차이가 난다.

- WideString은 AnsiString과는 달리, 유니코드 문자열을 지원하는 WideChar 문자로 구성된다.
- WideString은 OLE BSTR 문자열과의 호환성을 위해 WIN32 API인 SysAllocStrLen() 함수로 메모리를 할당한다.
- WideString은 참조 횟수를 계산하지 않으므로 한 WideString 변수의 내용을 다른 WideString 변수에 배정하면 그 시점에서 실제로 메모리 할당과 복사가 일어난다. 따라서, 메모리나 속도면에서 AnsiString보다 불리하다.

앞에서도 말했듯이 AnsiString과 WideString의 변환은 컴파일러가 알아서 처리한다. 다음 코드가 그 예이다.

```
var
  W: WideString;
  S: string;
begin
  W := '정보문화사';
  S := W;  // WideString이 AnsiString으로 변환됨.
  S := 'InfoPub';
  W := S;  // AnsiString이 WideString으로 변환됨.
end;
```

WideString은 2바이트 이상의 문자 체계를 가지는 비영어-유럽어권 국가들을 위한 것으로, 사용법은 AnsiString과 다를 것이 없다. 즉, Concat(), Copy(), Insert(), Length(), Pos(), SetLength() 루틴이나 +, =, <> 연산자들을 그대로 사용할 수 있다. 따라서, 다음 코드는 그대로 사용될 수 있다.

```
var
  W1, W2: WideString;
  P: Integer;
begin
  W1 := 'Enfield';
  W2 := 'field';
  if W1 <> W2 then
      P := Pos(W1, W2);
end;
```

AnsiString이나 ShortString에서와 마찬가지로, WideString에서도 배열 인덱스를 이용해서 문자열의 개별 문자를 얻거나 설정할 수 있다.

```
var
  W: WideString;
  C: WideChar;
begin
  W := '내가 아주 어렸을 때';
  C := W[Length(W)];   // C에는 W의 마지막 문자, 즉 '때'가
                       // 들어간다(여기서 한글은 하나의 문자로 취급함).
end;
```

## 널 종료 문자열

이번 장 앞에서 델파이에는 세 종류의 널 종료 문자열(null-terminated string)이 있다고 했다. 바로 PChar, PAnsiChar, PWideChar이다. 이 세 형은 델파이의 세 가지 문자형인 Char와 AnsiChar, WideChar의 널 종료 문자열이다. 설명에 대해 쉽게 이해할 수 있도록 PChar를 기준으로 이야기하겠다. PChar는 델파이 1.0과의 이전 버전 호환성과 널 종료 문자열을 주로 사용하는 Win32 API를 위한 것이다. PChar는 널(null, 실

제 값은 0)로 끝나는 문자열을 가리키는 포인터이다(포인터가 무엇인지 잘 모른다고 해도 계속 읽어나가기 바란다. 포인터에 대해서는 잠시 후에 좀더 자세히 이야기하겠다). AnsiString이나 WideString과는 달리, PChar는 메모리 할당이나 해제할 경우에 오브젝트 파스칼이 자동으로 처리해 주지 않는다. 따라서, 오브젝트 파스칼의 메모리 할당 함수들을 이용해서 PChar가 가리킬 문자열 공간을 직접 할당해 주어야 한다. PChar가 가리키는 문자열 공간의 최대 크기는 이론적으로 4기가 바이트 정도이다(윈도우즈의 한계를 반영한 것이다). PChar 변수의 메모리 구조는 그림 2.3과 같다.




대부분의 경우, PChar를 사용해야 하는 곳에 AnsiString을 사용할 수 있다는 점을 기억하기 바란다. AnsiString의 메모리는 오브젝트 파스칼이 직접 관리하므로 개발자가 일일이 메모리 할당과 해제를 신경써야 하는 PChar보다 훨씬 더 편하며, 메모리가 깨지거나 샐 위험도 훨씬 줄어든다.

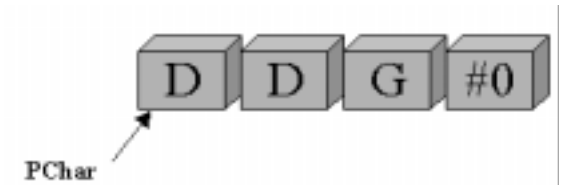



**그림 2.3**
PChar의 메모리 구조

## PChar의 사용법

PChar의 경우, 문자열을 담을 메모리 버퍼를 개발자가 직접 할당하고 해제해야 한다고 했었다. 일반적으로, PChar 메모리 버퍼를 할당할 때에는 StrAlloc() 함수를 사용하지만, 그 이외에도 AllocMem()이나 GetMem(), StrNew(), 심지어는 API 함수 VirtualAlloc()를 사용할 수도 있다. 중요한 것은 할당 함수와 짝이 맞는 해제 함수를 사용해야 한다는 것이다. 표 2.6은 여러 메모리 할당 함수들과 그에 해당하는 해제 함수들을 정리한 것이다.

표 2.6 메모리 할당과 해제 함수들

| 메모리 할당 함수 | 그에 해당되는 해제 함수 |
|---|---|
| AllocMem() | FreeMem() |
| GlobalAlloc() | GlobalFree() |
| GetMem() | FreeMem() |
| New() | Dispose() |
| StrAlloc() | StrDispose() |
| StrNew() | StrDispose() |
| VirtualAlloc() | VirtualFree() |

다음 예제는 PChar와 string을 이용한 여러 가지 메모리 할당 기법들을 보여주는 것이다.

```
var
  P1, P2: PChar;
  S1, S2: string;
begin
  P1 := StrAlloc(64 * SizeOf(Char));// P1은
  // 63자(Char)가 들어갈 수 있는 메모리 공간을 가리킴.
  StrPCopy(P1, '델파이 5');      // 문자열값을 P1에 복사함.
  S1 := '개발자 가이드';         // 문자열 S1에 문자열값을 넣음.
  P2 := StrNew(PChar(S1));      // P1은 S1의 복사본을 가리킴.
  StrCat(P1, P2);               // P1과 P2를 합침.
  S2 := P1;                     // S2는 '델파이 5 개발자 가이드'를 가짐.
  StrDispose(P1);               // P1과 P2의 메모리 버퍼를 해제
  StrDispose(P2);
end.
```

우선, SizeOf(Char)를 사용했다는 점에 주목하기 바란다. 현재 Char는 1바이트이지만, 이후 버전의 델파이에서는 2바이트가 될 수도 있기 때문이다. 1이나 2 같은 값을 직접 사용하는 대신, SizeOf()를 사용하면 이후에 Char의 정의가 변한다고 해도 제대로 작동하는 코드가 된다.

StrCat()은 두 개의 PChar 문자열을 합치는 프로시저이다. PChar의 경우는 긴 문자열이나 ShortString의 + 연산자를 사용할 수가 없다.

StrNew() 함수는 문자열 변수 S1에 담긴 문자열을 PChar 변수 P2에 복사하는 역할을 한다. 이 함수를 쓸 때 주의해야 할 점은, StrNew() 함수를 사용한 후 PChar 변수에는 S1의 문자열을 담을 수 있는 길이만큼의 메모리만 할당된다는 점이다. 다음 코드의 경우, P1에 할당된 메모리 공간보다 더 큰 문자열이 P1에 할당되기 때문에 메모리가 깨지는 결과를 낳는다.

```
var
  P1, P2: PChar;
begin
  P1 := StrNew('Hello ');    // P1에 6자의 문자를 담을 수 있는 메모리가 할당됨.
  P2 := StrNew('World');
  StrCat(P1, P2);            // P1에 6자 이상의 문자열이 들어가므로
                             // 메모리가 충돌한다.
  .
  .
  .
end;
```



다른 문자열들과 마찬가지로, PChar에도 여러 가지 문자열 조작 함수들이 있다. 델파이 온라인 도움말의 "String-handling routines(null-terminated)" 항목을 참고하기 바란다.
또한, 이 책 부록 CD의 \Code\Utils 디렉토리의 StrUtils.pas에도 유용한 널 종료 문자열 조작 함수/프로시저들이 있으니 참고하기 바란다.

## Variant형

델파이 2.0에서 Variant라는 강력한 형이 도입되었다. Variant는 원래 OLE 자동화(OLE Automation)를 지원하기 위해 도입된 것이다. 실제로, 델파이의 Variant형은

OLE의 Variant형을 캡슐화한 것이다. 원래는 OLE를 위한 것이었지만, 다른 부분에서도 여러 모로 유용하게 쓰인다. 현재 Variant형을 실행 시점에서는 동적 형으로, 또 컴파일 시점에서는 정적 형으로 완벽하게 통합할 수 있는 컴파일 언어는 델파이가 유일하다고 할 수 있다.

델파이 3에서는 OleVariant라는 새로운 형이 도입되었다. OleVariant는 기본적으로 Variant와 동일하나, 자동화 호환 형만을 담을 수 있다는 점이 다르다. 일단 Variant에 대해 집중적으로 설명하고, OleVariant에 대해서는 Variant와 다른 부분을 중점적으로 설명하겠다.

## 동적으로 형을 변경할 수 있는 Variant형

Variant의 주요 장점은 컴파일 시점에서 데이터형이 정해질 수 없는 데이터형에 대한 변수를 가지는 것이다. 이것이 의미하는 바는 실행시 참조할 수 있는 형을 바꿀 수 있다는 것을 의미한다. 예를 들어서, 다음 코드는 컴파일이나 실행시 아무런 에러도 발생시키지 않는다.

```
var
  V: Variant;
begin
  V := '델파이 5 만세!';     // Variant에 문자열이 들어감.
  V := 1;                   // Variant에 정수값이 들어감.
  V := 123.32;              // Variant에 부동 소수점값이 들어감.
  V := True;                // Variant에 부울값이 들어감.
  V := CreateOleObject('Word.Basic');// Variant에 OLE 객체가 들어감.
end;
```

Variant는 정수, 부동 소수점, 문자열, 부울, 날짜나 시간, 통화 같은 모든 기본형들과 OLE 자동화 오브젝트를 지원한다. 다만, Variant에 오브젝트 파스칼 오브젝트를 담을 수는 없다는 점을 기억하기 바란다. 또한, Variant 배열의 경우에 배열의 각 Variant 변수들이 서로 다른 형을 가질 수 있다는 점도 기억하기 바란다. 다시 말하면, 문자열, 정수, 실수 등을 하나의 Variant 배열에 담을 수 있다는 뜻이다. 심지어, Variant 배열에 다른 Variant 배열을 담을 수도 있다.

## Variant의 구조

다음은 System 유닛에 정의된 Variant 형의 데이터 구조체이다.

```
type
PVarData = ^TVarData;
  TVarData = packed record
    VType: Word;
    Reserved1, Reserved2, Reserved3: Word;
    case Integer of
      varSmallint: (VSmallint: Smallint);
      varInteger:  (VInteger: Integer);
      varSingle:   (VSingle: Single);
      varDouble:   (VDouble: Double);
      varCurrency: (VCurrency: Currency);
      varDate:     (VDate: Double);
      varOleStr:   (VOleStr: PWideChar);
      varDispatch: (VDispatch: Pointer);
      varError:    (VError: LongWord);
      varBoolean:  (VBoolean: WordBool);
      varUnknown:  (VUnknown: Pointer);
      varByte:     (VByte: Byte);
      varString:   (VString: Pointer);
      varAny:      (VAny: Pointer);
      varArray:    (VArray: PVarArray);
      varByRef:    (VPointer: Pointer);
  end;
```

TVarData 구조체는 16바이트의 메모리를 차지한다. TVarData 레코드의 첫 번째 두 바이트(VType)는 Variant가 가리키는 데이터형을 뜻하는 워드값이다. VType 필드에 사용할 수 있는 상수값들은 다음 목록과 같다. 그 다음의 6바이트는 아직 사용되지 않는다. 나머지 8바이트는 실제 데이터 또는 실제 데이터를 가리키는 포인터이다. 다시 한 번 말하지만, 이 구조체는 OLE Variant의 구조체와 정확히 일치한다.

```
{ Variant형 코드 }
const
       varEmpty    = $0000;
       varNull     = $0001;
       varSmallint = $0002;
       varInteger  = $0003;
       varSingle   = $0004;
       varDouble   = $0005;
       varCurrency = $0006;
       varDate     = $0007;
       varOleStr   = $0008;
       varDispatch = $0009;
       varError    = $000A;
       varBoolean  = $000B;
       varVariant  = $000C;
       varUnknown  = $000D;
       varString   = $0100;
       varAny      = $0101;
       varTypeMask = $0FFF;
       varArray    = $2000;
       varByRef    = $4000;
```




위의 형 코드에서 볼 수 있듯이 Variant는 Pointer나 Class형을 담을 수 없다.

TVarData 레코드는 일종의 variant 레코드일 뿐이다. variant 레코드와 Variant형을 혼동하지 말기 바란다. variant 레코드와 Varinat형은 별개의 개념이며, Variant형 안에 variant 레코드가 포함되어 있는 것이다(TVarData 이외의 레코드들도 포함되어 있다). variant 레코드는 C/C++의 union처럼 동일한 메모리 공간에 여러 종류의 형 중 하나를 담을 수 있는 레코드이다. "레코드"에 대해서는 이후에 좀더 자세히 이야기하겠다. variant 레코드가 의미하는 형은 TVarData의 case문에서 결정된다. 예를 들어서,

VType 필드의 값이 varInteger이면 8바이트로 된 레코드의 variant 부분 중 4바이트에 정수값이 들어간다. 마찬가지로, VType이 varByte이면 8바이트 중 한 바이트만 사용된다.

VType의 값이 varString, 즉 문자열을 뜻할 때에는 8바이트만으로 부족할 것이 당연하다. 따라서, TVarData에 들어가는 값은 문자열 자체가 아니라, 문자열에 대한 포인터이다. variant의 필드들을 직접 접근할 수 있다면 다음처럼 접근하는 것도 가능하다.

```
var
  V: Variant;
begin
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 2;
end;
```

물론, 자연스러운 방법은 아니며, 특히 문자열 같은 "수명 관리"형인 경우에 포인터가 잘못된 메모리 공간을 가리키도록 하면 메모리가 깨지거나 샐 위험도 생긴다. "메모리가 샌다"는 말의 의미는 "가비지 컬렉션"을 설명하면서 다시 이야기하겠다.

## Variant의 수명 관리

Variant형의 메모리 할당이나 해제는 델파이가 자동으로 처리한다. 예를 들어서, 다음과 같이 Variant형에 문자열을 배정하는 코드가 있다고 하자.

```
procedure ShowVariant(S: string);
var
  V: Variant;
begin
  V := S;
  ShowMessage(V);
end;
```

위의 코드를 실행할 때 델파이가 내부적으로 처리하는 과정을 보자. 우선, 델파이는 Variant 변수를 Unassigned(아직 사용되지 않았다는 뜻의 특별한 값)로 초기화한다. 문자열을 배정하는 V := S 문장에서 델파이는 VType 필드를 varString으로 설정하고, S의 문자열 포인터를 VString 필드에 복사한다. Variant 변수가 자신의 범위에서 벗어

날 때(즉, 프로시저가 끝나고 호출한 곳으로 제어권을 넘길 때)는 Variant 변수를 해제하고, 문자열 S의 참조 횟수를 하나 감소시킨다. 다른 수명 관리 데이터형과 마찬가지로, 델파이는 다음과 같은 코드를 삽입한다고 생각하면 된다(실제로, 코드 편집기에 나타나지는 않는다).

```
procedure ShowVariant(S: string);
var
  V: Variant;
begin
  V := Unassigned; // 변수를 빈 값으로 초기화한다.
  try
      V := S;  // VType과 VString 필드가 적절히 설정된다.
      ShowMessage(V);
  finally
      // V에 관련된 모든 자원들을 해제한다.
  end;
end;
```

문자열뿐만 아니라, 다른 형태의 데이터를 배정할 때에도 마찬가지 일이 일어난다. 다음 코드를 보자.

```
procedure ChangeVariant(S: string);
var
  V: Variant;
begin
  V := S;
  V := 34;
end;
```

위의 코드가 실제로 수행하는 일은 다음과 같다(파스칼 문법에 구애되지 않는 가상의 코드가 섞여 있다).

```
procedure ChangeVariant(S: string);
var
  V: Variant;
begin
  // Variant 변수 V가 빈 값을 가지도록 초기화한다.
  try
      V.VType := varSTring; V.VString := S; Inc(S.RefCount);
      // 정수를 넣어야 하므로 문자열을 해제하고 V를 다시 빈 값으로 초기화한다.
      // 이 때 S의 참조 횟수도 감소된다.
      V.VType := varInteger; V.VInteger := 34;
  finally
      // V에 관련된 모든 자원들을 해제시킨다.
end;
```

이 코드를 제대로 이해한다면 TVarData를 직접 다루는 것이 왜 위험한 일인지 이해할 수 있을 것이다. 다음 코드를 보면 더욱 명확해질 것이다.

```
procedure ChangeVariant(S: string);
var
  V: Variant;
begin
  V := S;
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 32;
  V := 34;
end;
```

언뜻 보면 별 문제가 없어 보이지만, V := S에서 증가된 문자열의 참조 횟수가 감소되지 않았으므로 메모리 누수 현상이 일어날 것이다. TVarData 레코드를 직접 조작하지 않아야 하며, 읽기 전용으로만 사용하거나, 아니면 Variant의 내부 작동에 대해 충분한 지식을 갖춘 후 사용하는 것이 좋다.

## Variant의 형 변환

다른 데이터형을 Variant로 변환할 때에는 다음과 같이 하면 된다.

```
Variant(x)
```

위의 코드는 기본 데이터형(정수, 실수, 통화, 문자열, 부울형 중 하나이어야 한다) 변수 x를 Variant형 변수로 변환시킨다.

Variant 변수를 다른 기본 데이터형으로 변환하는 것도 가능하다. 다음과 같은 코드가 있다고 하자.

```
V := 1.6;
```

여기서 V는 Variant형 변수이다. 다음은 V를 여러 데이터형으로 변환하는 예이다.

```
S := string(V);     // S는 '1.6'이라는 문자열이 된다.
I := Integer(V);    // I에는 1.6이 반올림되어서 2가 들어간다.
B := Boolean(V);
// 부울값에서 0이 아닌 모든 값은 True이므로 B는 True이다.
D := Double(V);// D는 1.6이다.
```

위에서처럼 Variant형의 변환에는 일정한 규칙이 있다. 델파이 온라인 도움말의 Object Pascal Language Guide에 자세히 나와 있으니 참고하기 바란다.

그건 그렇고, 위에서는 string()이나 Integer() 등을 이용해서 명시적으로 형 변환을 했지만, 반드시 그럴 필요는 없다. 다음처럼 그냥 배정하기만 해도 묵시적인 형 변환이 일어나며, 결과는 위와 동일하다.

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

물론, 각 변수는 해당 데이터형으로 미리 선언되어 있어야 한다. 한 가지 중요한 점은, 묵시적인 형 변환은 실행 시점에서 일어나므로 수행 성능이 좀 떨어진다는 것이

다. 따라서, Variant형이 어떤 데이터형으로 변환되는지 설계 시점에서 판단할 수 있다면 명시적으로 형 변환을 시켜주는 것이 수행 성능에 도움이 된다. 특히, 표현식(수식) 안에서 Variant형 변수가 사용될 경우에 성능의 차이는 더욱 커진다.

## Variant와 표현식

표현식에 사용할 수 있는 연산자는 +, =, *, /, div, mod, shl, shr, and, or, xor, not, :=, <>, <, >, <=, >=이다.

Variant 변수를 표현식에서 사용할 때 델파이는 그 변수에 담긴 내용에 따라서 적절한 연산을 수행한다. 예를 들어서, 두 개의 Variant 변수 V1과 V2에 정수값이 들어 있는 경우, V1 + V2는 두 정수의 합을 의미한다. 만일 V1과 V2가 문자열인 경우는 두 문자열이 합쳐진다. 그런데 두 변수에 담긴 값의 데이터형이 다른 경우는 어떻게 될까? 델파이는 특정한 규칙에 따라 연산식을 계산한다. 예를 들어서, V1에는 '4.5'라는 문자열이 들어 있고, V2에는 부동 소수점값이 들어 있는 경우에 델파이는 V1을 부동 소수점값으로 변환한 후 두 값을 더하며, 결과 역시 부동 소수점값이 된다. 다음 코드는 문자열과 정수를 더하는 예이다.

```
var
  V1, V2, V3: Variant;
begin
  V1 := '100'       // string형
  v2 := '50'        // string형
  V3 := 200         // Integer형
  V1 := V1 + V2 + V3;
end;
```

과연 최종적으로 V1에는 어떤 값이 들어갈까? 언뜻 생각하면 350이 될 것 같지만, 그렇지 않다. 표현식의 연산 순위를 잘 생각해 보자. 가장 먼저 일어나는 연산은 V1 + V2인데, 두 변수는 문자열을 담고 있다. 따라서, 연산 결과는 정수 150이 아니라 문자열 '10050'이다. 거기에 정수값 V3을 더하면 문자열 '10050'이 정수 10050으로 변환된 후 V3의 200과 합산되어서 결과는 10250이 된다. 최종적으로, V1에는 10250이 들어가는 것이다.

델파이는 최대한 표현식을 제대로 계산해 내는 방향으로 Variant의 형 변환을 수행한다. 그러나 다음 코드와 같이 계산이 전혀 불가능한 경우에는 "Invalid variant type conversion(잘못된 Variant형 형 변환)" 예외를 발생시킨다.

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  V1 := V1 / V2;   // 예외가 발생한다.
end;
```

앞에서도 이야기한 것과 같이 설계 시점에서 Variant의 형 변환을 결정할 수 있고, 또 Variant형이 표현식에서 사용되는 경우에는 명시적인 형 변환을 해주는 것이 속도나 디버깅에 도움이 된다. 다음 코드를 보자.

```
V4 := V1 * V2 / V3;
```

위의 문장이 실행될 때 실제 연산이 일어나기 전에, 우선 각 변수의 데이터형을 판단해서 적절한 형 변환 규칙을 찾고, 각 변수의 형 변환을 하는 작업이 일어난다. 그러한 사전 작업 때문에 코드의 크기도 커지고, 수행 속도도 떨어지는 것이다. 가장 좋은 방법은 Variant 같은 내부적으로 복잡한 형을 사용하지 않는 것이지만, 꼭 Variant를 사용해야 한다면 다음과 같이 명시적인 형 변환을 해주는 것이 좋다. 그러면 컴파일 시점에서 형 변환이 일어나므로 속도나 코드 크기의 문제를 피할 수 있다.

```
V4 := Integer(V1) * Double(V2) / Integer(V3);
```

물론, 각 변수들의 데이터형을 설계 시점에서 판단할 수 있어야 한다는 점을 잊어서는 안된다(다시 말하면, 위의 문장이 실행되는 시점에서 V1이 반드시 정수라고 확신할 수 있어야 한다).

## 빈 값(Empty)과 널값(Null)

VType의 상수들 중 기본 데이터형에 해당하지 않는 것이 두 개 있다. 첫 번째는 varEmpty로, Variant 변수에 아직 아무런 값도 배정되지 않았다는 뜻이다. Variant 변수가 초기화될 때, 즉 자신의 범위에 진입할 때 varEmpty를 가진다. 두 번째는 varNull인데, 이것은 변수가 비어 있다는 뜻이 아니라, 널값이 들어 있다는 뜻이다. '비어 있음'과 널값은 다르다. 특히, 데이터베이스 테이블의 필드값에서 "비어 있음"과 "널 값이 들어 있음"의 차이는 매우 중요하다. 27장, "데스크탑 데이터베이스 애플리케이션의 작성"에서 데이터베이스 애플리케이션에 Variant를 사용하는 예가 나온다.

그리고 VType의 값이 varEmpty인 Variant 변수를 표현식에서 사용하면 "invalid variant operation(잘못된 Variant형 연산)" 예외가 발생한다는 점도 중요하다. varNull일 경우에는 그런 예외가 발생하지 않는다. 널값을 가진 Variant 변수가 포함된 수식의 결과는 항상 널이 된다.

빈 값이나 널값을 직접 배정할 때에는 System 유닛에 정의된 Variant값인 Unassigned나 Null을 사용하면 된다. 두 값은 VType의 값을 각각 varEmpty와 varNull로 설정한다.




Variant는 매우 융통성 있는 데이터형이기 때문에 상당히 편리해 보이지만, 애플리케이션의 수행 성능이 떨어지며, 코드의 크기가 커진다는 점을 명심하기 바란다. 또한, 코드의 유지보수도 힘들어진다. 물론, Variant형이 꼭 필요한 경우가 있다. 특히, ActiveX나 데이터베이스에 관련된 VCL들은 Variant를 많이 사용한다. 어쨌든, 일반적인 경우라면 Variant보다는 기본 데이터형을 사용하는 것이 좋다. Variant의 융통성이 주는 장점이 수행 속도나 코드 크기의 단점보다 더 나은 경우에만 Variant를 사용하기 바란다. 원래 델파이는 엄격한 형 검사로 유명한 언어이다. 데이터형이 애매하면 버그도 애매해진다는 점을 기억했으면 좋겠다.

## Variant 배열

Variant형은 하나 이상의 데이터를 담는 배열로 사용할 수 있다. 따라서, 다음과 같은 코드는 문법적으로 틀리지 않다.

```
var
  V: Variant;
  I, J: Integer;
begin
  I := V[J];
end;
```

문법적으로 틀리지 않기 때문에 컴파일은 된다. 그러나 실행을 시키면 V에는 아직 Variant 배열이 담겨 있지 않기 때문에 런타임 에러가 발생한다. 오브젝트 파스칼에서 Variant 배열을 만드는 데 사용하는 함수에는 VarArrayCreate()와 VarArrayOf()가 있다.

### VarArrayCreate()

System 유닛에서 VarArrayCreate()는 다음과 같이 정의되어 있다.

```
function VarArrayCreate(const Bounds: array of Integer;
  VarType: Integer): Variant;
```

Bounds에는 만들고자 하는 배열의 상하 한계를 넣고, VarType에는 만들고자 하는 배열의 데이터형을 넣으면 된다(Bounds는 소위 개방형 배열이라고 하는 것인데, 개방형 배열에 대해서는 잠시 후 "인자 전달 방식" 부분에서 이야기하겠다). 다음의 코드는 정수로 구성된 Variant 배열을 만들고 정수값들을 집어넣는 예이다.

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 4], varInteger);
  // 요소가 4개인 배열을 만든다.
  V[1] := 1;
  V[2] := 2;
  V[3] := 3;
  V[4] := 4;
end;
```

만일 VarType에 varVariant를 넣으면 Variant로 된 배열이 만들어진다. 따라서, 하나의 배열에 서로 다른 형의 데이터들을 담을 수 있게 된다. 다차원 배열을 만들 때에는 다음과 같이 배열의 상하 한계 배열을 써주면 된다. 다음은 2차원 배열을 만드는 예이다.

```
V := VarArrayCreate([1, 4, 1, 5], varInteger);
```

### VarArrayOf()

System 유닛에서 VarArrayCreate()는 다음과 같이 정의되어 있다.

```
function VarArrayOf(const Values: array of Variant): Variant;
```

이 함수는 Values 인자(역시 개방형 배열이다)로 넘겨 받은 요소들로 된 1차원 배열을 돌려준다. 다음은 정수, 문자열, 부동 소수점값으로 이루어진 Variant 배열을 만드는 예이다.

```
V := VarArrayOf([1, 'Delphi', 2,2]);
```

### Variant 배열에 유용한 함수와 프로시저들

VarArrayCreate()나 VarArrayOf() 이외에도, Variant 배열을 위한 여러 함수/프로시저들이 있다. 다음은 System 유닛에 정의되어 있는 관련 함수/프로시저들이다.

```
procedure _VarArrayRedim(var A : Variant; HighBound: Integer);
function VarArrayDimCount(const A: Variant): Integer;
function VarArrayLowBound(const A: Variant; Dim: Integer): Integer;
function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;
function VarArrayLock(const A: Variant): Pointer;
procedure VarArrayUnlock(const A: Variant);
function VarArrayRef(const A: Variant): Variant;
function VarIsArray(const A: Variant): Boolean;
```

_VarArrayRedim() 함수는 Variant 배열의 제일 오른쪽 차원의 상한을 변경하는 함수이다. VarArrayDimCount() 함수는 Variant 배열의 차원 수를 돌려준다. VarArrayLowBound()와 VarArrayHighBound()는 각각 배열의 하한과 상한을 돌려준다. VarArrayLock()과 VarArrayUnlock()은 특수한 함수로, 잠시 후에 자세히 이야기하겠다.

VarArrayRef()는 OLE 자동화 서버에 Variant 배열을 넘겨줄 때 생기는 에러를 방지하기 위한 것이다. 다음과 같이 Variant 배열이 담긴 Variant 변수를 자동화 메소드에 넘겨주면 문제가 생긴다.

```
Server.PassVariantArray(VA);
```

배열은 Variant 배열로 절단되는 것이 아니라, Variant 배열을 가리키는 Variant 변수로 전달된다. 배열 자체와 배열을 가리키는 변수는 전혀 다른 것이다. 서버가 배열에 대한 포인터가 아닌 배열 자체를 원하는 경우에는 위와 같은 문법을 사용하면 에러가 발생할 수 있다. VarArrayRef()는 Variant 변수를 자동화 서버가 원하는 형태로 변환시켜서 그러한 문제를 해결한다. VarArrayRef()의 사용법은 다음과 같다.

```
Server.PassVariantArray(VarArrayRef(VA));
```

VarIsArray()는 넘겨 받은 인자가 Variant 배열일 경우에는 True를 돌려주고, 그렇지 않으면 False를 돌려준다.

### 큰 배열의 초기화 - VarArrayLock(), VarArrayUnlock()

Variant 배열은 OLE 자동화에서 중요한 역할을 한다. OLE 자동화 서버에 이진 데이터를 넘겨주려면 Variant 배열을 사용해야만 하기 때문이다(23장, "COM과 ActiveX"에서 설명하겠지만, OLE 자동화에서 포인터는 유효한 데이터형이 아니다). 그러나 Variant 배열을 잘못 사용하면 상당히 비효율적이라는 점을 명심하기 바란다. 다음 코드를 생각해 보자.

```
V := VarArrayCreate([1,1000], VarByte);
```

위의 코드는 10000바이트의 Variant 배열을 생성한다. 만일 Variant가 아닌 10000개짜리 배열의 값들을 Variant 배열에 복사한다고 하자. 그러면 다음과 같이 루프를 사용해서 하나 하나씩 복사할 것이다.

```
begin
  V := VarArrayCreate([1, 1000], VarByte);
  for i := 1 to 10000 do
      V[i] := A[i];
end;
```

Variant 배열이 아닌 일반 배열끼리 복사하는 것과 비교할 때 Variant 배열의 속도는 매우 느리다. 앞에서도 이야기했듯이 Variant에 어떤 값을 배정하면 데이터의 형 검사와 형 변환이 일어나기 때문이다. 위의 코드는 10000번의 형 검사와 형 변환이 일어난다. VarArrayLock()과 VarArrayUnlock()을 사용하면 실행 시점의 형 검사와 형 변환을 하지 않도록 할 수 있다.

VarArrayLock()은 메모리의 배열을 잠그고, 배열 데이터를 가리키는 포인터를 돌려준다. 일단 잠긴 배열은 메모리 안에서 이동하지 않으며, 크기도 변하지 않는다. VarArrayUnlock은 잠긴 배열을 다시 풀어준다. 배열이 풀리면 크기를 변경할 수 있으며, 메모리 안에서 이동할 수도 있다. 배열을 잠근 상태에서 다음 코드와 같이 Move() 프로시저와 배열에 대한 포인터(첨자를 뺀 배열 변수의 이름이 배열에 대한 포인터이다)를 이용하면 좀더 효율적으로 Variant 배열에 값을 복사할 수 있다. 다음 코드는 위의 루프 코드보다 훨씬 더 빠르다.

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  P := VarArrayLock(V);
  try
      Move(A, P^, 10000);
  finally
      VarArrayUnlock(V);
  end;
end;
```

## Variant형에 유용한 함수와 프로시저들

Variant형을 사용할 때 도움이 되는 여러 함수나 프로시저들이 있다. 다음은 System 유닛에 정의된 함수/프로시저들이다.

```
procedure _VarClear(var V : Variant);
procedure _VarCopy(var Dest : Variant; const Source: Variant);
procedure _VarCast(var Dest : Variant; const Source: Variant; VarType:
Integer);
procedure _VarCastOle(var Dest : Variant; const Source: Variant; VarType:
Integer);
procedure VarCopyNoInd(var Dest: Variant; const Source: Variant);
function VarType(const V: Variant): Integer;
function VarAsType(const V: Variant; VarType: Integer): Variant;
function VarIsEmpty(const V: Variant): Boolean;
function VarIsNull(const V: Variant): Boolean;
function VarToStr(const V: Variant): string;
function VarFromDateTime(DateTime: TDateTime): Variant;
function VarToDateTime(const V: Variant): TDateTime;
```

_VarClear() 프로시저는 Variant를 비우고, VType 필드를 varEmpty로 설정한다. _VarCopy()는 Source Variant를 Dest Variant에 복사한다. _VarCast() 프로시저는 Variant를 인자로 지정된 형으로 변환하고, 결과를 다른 Variant에 저장한다.

VarType()은 인자로 지정된 Variant의 데이터형(varXXX)을 돌려준다. VarAsType()은 _VarCast()와 동일한 기능을 한다. VarIsEmpty()는 지정된 Variant의 Vtype이 varEmpty이면 True를 돌려준다. VarIsNull은 지정된 Variant가 널값을 가지고 있으면(varNull) True를 돌려준다. VarToStr()은 지정된 Variant의 내용을 문자열 형태로 돌려준다(널이나 빈 Variant인 경우, 빈 문자열을 돌려준다). VarFromDateTime()은 지정된 TDateTime 값을 담은 Variant를 돌려주며, VarToDateTime()은 Variant에 담긴 날짜/시간 값을 TDateTime형으로 돌려준다.

### OleVariant

OleVariant는 Variant와 거의 동일하다. 유일한 차이점이라면, OleVariant는 자동화 호환 데이터형만을 지원한다는 것이다. 현재 VType에 사용할 수 있는 데이터형들 중 자동화에 호환되지 않는 것은 AnsiString을 뜻하는 varString 하나이다. AnsiString값을

OleVariant에 넣으면 AnsiString은 자동으로 OLE BSTR형으로 바뀌며, varOleStr형으로 저장된다.

### Currency

통화를 표시하기 위한 Currency형은 델파이 2.0에서부터 지원했다. 소수점의 위치가 유동적으로 변하는 부동 소수점과는 달리, Currency형은 정수부가 15자리, 소수부가 4자리로 고정되어 있다. 따라서, 부동 소수점값의 반올림 때문에 생기는 계산 오류를 피할 수 있다. 만일 델파이 1.0에서 돈에 관련된 계산을 Single이나 Real, Double 등으로 수행했다면 델파이 2.0 이상에서는 모두 Currency로 고치는 것이 좋다.

## 사용자 정의형

정수나 문자열, 부동 소수점값만으로는 프로그램이 해결해야 할 데이터 구조를 제대로 표현할 수 없는 경우가 많다. 따라서, 대부분의 프로그래밍 언어는 개발자가 직접 데이터 구조를 정의해서 쓸 수 있도록 하는데, 델파이 역시 마찬가지이다. 델파이에서 사용자 정의형은 주로 레코드나 객체의 형태로 정의된다. 사용자 정의형을 정의할 때에는 Type 키워드를 사용한다.

### 배열

오브젝트 파스칼은 파일을 제외한 모든 형태의 변수들로 배열을 만들 수 있다. 예를 들어서, 여덟 개의 정수로 이루어진 배열은 다음과 같이 선언한다.

```
var
  A: Array[0..7] of Integer;
```

이것을 C로 표현하면 다음과 같다.

```
int A[8];
```

또한, 비주얼 베이직이라면 다음과 같을 것이다.

```
Dim A(8) as Integer
```

배열 선언에 있어서 오브젝트 파스칼은 다른 언어와는 다른 특성을 가지고 있다. 특히, 배열의 하한, 상한을 마음대로 지정할 수 있다. 다음은 첨자가 28부터 시작하는 원소 세 개짜리 배열이다.

```
var
  A: Array[28..30] of Integer;
```

오브젝트 파스칼의 배열은 다른 언어들처럼 0이나 1부터 시작하지 않을 수도 있기 때문에 루프를 돌릴 때 주의해야 한다. 그렇다고 배열의 하한과 상한을 외우고 있을 필요는 없다. 배열의 하한과 상한을 알려주는 Low()와 High()라는 컴파일 시점 함수가 있기 때문이다. 다음은 그 두 함수를 이용해서 루프를 돌리는 예이다.

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do
// 배열의 상한과 하한을 직접 써줄 필요가 없다.
      A[i] := i;
end;
```



문자(Char) 배열은 항상 0으로 시작하는 것이 좋다. 0으로 시작하는 문자 배열은 PChar형의 변수가 필요한 함수에 넘겨줄 수 있기 때문이다. 0으로 시작하는 문자 배열과 PChar의 호환은 일종의 특별 케이스라고 생각하면 된다.

다차원 배열은 다음과 같이 쉼표로 상·하한을 분리해서 써주면 된다.

```
var
  // 2차원 Integer 배열(2x2)
  A: array[1..2, 1..2] of Integer;
```

다차원 배열을 참조할 때는 대괄호 안에 각 첨자를 쉼표로 분리해서 써주면 된다.

```
I := A[1, 2];
```

## 동적 배열

동적 배열은 델파이 4부터 지원한다. 동적 배열(dynamic array)이란, 컴파일 시점에서 배열의 차원이나 원소의 개수를 결정하지 않아도 되는 배열을 뜻한다. 다음처럼 배열을 선언할 때 차원을 써주지 않으면 동적 배열이 된다.

```
var
  // 동적 문자열 배열
  SA: array of string;
```

동적 배열을 사용하려면 우선 SetLength() 프로시저로 배열의 메모리를 할당해야 한다.

```
begin
  // 33개의 원소를 담을 공간을 준비한다.
  SetLength(SA, 33);
```

일단 메모리가 할당된 후에는 일반 배열과 마찬가지 방법으로 사용하면 된다.

```
SA(0) := '델파이 재밌다';
OtherString := SA[0];
```



동적 배열의 첨자는 항상 0부터 시작한다.

동적 배열은 수명 관리형이므로 메모리를 직접 해제할 필요는 없다. 자신의 범위를 벗어나면 자동적으로 메모리가 해제된다. 그러나 범위를 벗어나기 전에 동적 배열의 메모리를 명시적으로 해제해야 하는 경우가 종종 발생한다(메모리를 너무 많이 잡아먹고 있다든지). 그럴 때에는 다음과 같이 nil을 넣어주면 된다.

```
SA := nil; // SA를 해제한다.
```

메모리 내부를 들여다 보면, 동적 배열은 일반 배열과는 달리 AnsiString에서 사용하는 참조 기법과 비슷한 기법을 사용한다. 다음 코드를 끝까지 실행한 시점에서 A1[0]의 값이 무엇인지 생각해 보자.

```
var
  A1, A2: array of Integer;
begin
  SetLength(A1, 4);
  A2 := A1;
  A1[0] := 1;
  A2[0] := 26;
```

정답은 1이 아니라 26이다. 왜냐하면, A2 := A1의 문장이 실행되어도 A2에 새로운 메모리가 할당되는 것이 아니라, A1이 차지하고 있는 메모리에 대한 포인터만 복사되기 때문이다. 따라서, A2의 어떤 값을 바꾸면 A1의 값도 바뀐다. 만일 A2에 독자적인 메모리를 할당하고 A1의 값들을 복사하려면 다음과 같이 Copy() 프로시저를 사용해야 한다.

```
A2 := Copy(A1);
```

위의 코드가 실행되면 A2와 A1은 내용은 같지만, 메모리 공간은 다른 배열을 가리키게 된다. 따라서, A2의 어떤 값을 바꾸더라도 A1의 값은 바뀌지 않는다. 다음과 같이 Copy()에 시작 첨자와 개수를 넣어주면 배열의 일정 범위만을 복사할 수도 있다.

```
// 3번부터 2개의 원소를 복사한다.
A2 := Copy(A1, 3, 2);
```

동적 배열 역시 다차원 배열로 사용할 수 있다. 다음은 2차원 동적 배열을 선언하는 것인데, array를 하나 더 써줄 때마다 차원이 하나씩 올라간다고 생각하면 된다.

```
var
  // 2차원 Integer 동적 배열
  IA: array of array of Integer;
```

다차원 동적 배열의 메모리를 할당할 때에는 다음처럼 SetLength()에 각 차원의 크기를 모두 넣어야 한다.

```
begin
  // 5x5 Integer 배열을 만든다.
  SetLength(IA, 5, 5);
```

메모리를 할당한 후에는 일반적인 다차원 배열과 마찬가지로, 대괄호 안에 각 첨자를 쉼표로 분리해서 배열의 각 요소를 사용할 수 있다.

```
IA[0, 3] := 28;
```

## 레코드(record)

오브젝트 파스칼에서 사용자 정의 구조체를 레코드라고 부르며, record라는 키워드로 정의한다. record는 C의 struct나 비주얼 베이직의 Type과 같은 의미이다. 다음은 오브젝트 파스칼과 C, 비주얼에서 사용자 구조체를 정의하는 예이다.

```
{ 파스칼 }
Type
  MyRec = record
      i: Integer;
      d: Double;
  end;
```

```
/* C */
typedef struct {
  int i;
  double d;
} MyRec;

' 비주얼 베이직
Type MyRec
  i As Integer
  d As Double
End Type
```

레코드의 각 필드에는 다음 코드와 같이 마침표(.)를 통해서 접근한다.

```
var
  N: MyRec;
begin
  N.i := 23;
  N.d := 3.4;
end;
```

오브젝트 파스칼은 Variant형(variant) 레코드를 지원한다. Variant형 레코드란, 레코드의 동일한 메모리 공간에 서로 다른 데이터형을 겹쳐쓸 수 있는 것을 말한다. Variant형 레코드를 Variant형과 혼동하면 안된다. Variant형 레코드는 각각의 겹쳐진 데이터 필드를 개별적으로 사용할 수 있다. C/C++를 아는 독자라면 struct의 union과 같은 개념임을 깨달았을 것이다.

다음 코드는 Double과 Integer, char가 동일한 메모리 공간을 차지하는 Variant형 레코드의 예이다.

```
type
  TVarinatRecord = record
       NullStrField: PChar;
       IntField: Integer;

       case Integer of
            0: (D: Double);
            1: (I: Integer);
            2: (C: char);
  end;
```



오브젝트 파스칼의 규칙에 따르면 레코드의 Variant형 부분에는 수명 관리 데이터형(AnsiString 등)를 쓸 수 없다고 되어 있다.

다음은 위의 코드에 해당하는 C의 코드이다.

```
struct TUnionStruct
{
  char * StrField;
  int IntField;
  union
  {
       double D;
       int i;
       char c;
  };
};
```

## 집합(set)

집합은 베이직이나 C, C++에는 없는 파스칼만의 고유한 형이다(볼랜드 C++ 빌더에 파스칼의 집합을 흉내내기 위한 Set이라는 템플릿 클래스가 있긴 하다). 집합은 서수(ordinal, 순서를 매길 수 있는 데이터형)나 문자, 열거값(enumulation)을 표현하는 데 아주 효율적이다. 사용자 정의 집합을 선언할 때에는 다음 코드처럼 type절 안에서 서수형이나 열거값 앞에 set of를 써준다.

```
type
  TCharSet = set of char;        // #0에서 #255를 원소로 하는 집합

  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;   // TEnum의 열거값들을 원소로 하는 집합

  TSubrangeSet = set of 1..10;// 특정 범위의 정수(1 - 10)를 원소로 하는 집합
  TAlphaSet = set of 'A'..'Z';
                        // 'A' - 'Z'에 속하는 문자들을 원소로 하는 집합
```

단, 집합의 원소 개수는 256개 이하이어야 한다. 또한, set of 다음에는 서수형만이 올 수 있다. 따라서, 다음과 같은 집합 선언은 잘못된 것이다.

```
type
  TIntSet = set of Integer;      // 원소의 개수가 너무 많음.
  TstrSet = set of string;       // 서수형이 아님.
```

내부적으로 집합의 원소들은 개별적인 비트들로 저장된다. 따라서, 속도나 메모리의 효율이 좋다. 기본 데이터형을 원소로 하며, 원소의 개수가 32개 미만인 집합은 CPU의 레지스터를 이용하므로 더욱 더 빠르다. 32개 이상인 경우에는(예를 들면, 255개짜리 set of char) 메모리에 저장된다. 따라서, 집합의 진정한 장점을 살리려면 원소의 개수를 32개 이하로 하는 것이 좋다.

## 집합의 사용법

집합에 원소들을 배정할 때에는 대괄호([])를 사용한다. 다음 코드는 집합형의 변수를 설정하고, 원소들을 배정하는 예이다.

```
type
  TCharSet = set of char;    // #0에서 #255까지의 원소를 가지는 집합

  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
  TEnumSet = set of TEnum;   // TEnum의 열거값들을 원소로 하는 집합

var
  CharSet: TCharSet;
  EnumSet: TEnumSet;
  SubrangeSet: set of 1..10; // 특정 범위의 정수(1 - 10)를 원소로 하는 집합
  AlphaSet = set of 'A'..'Z'; // 'A' - 'Z'에 속하는 문자들을 원소로 하는 집합

begin
  CharSet := ['A'..'J', 'a', 'm'];
  EnumSet := [Saturday, Sunday];
  SubrangeSet := [1, 2, 4..6];
  AlphaSet := [];       // 공집합
end;
```

## 집합 연산자

초등학교나 중학교 때 집합의 연산에 대해서 배웠을 것이다. 오브젝트 파스칼도 원소의 포함 관계, 합집합, 차집합, 교집합 같은 개념들을 지원한다.

### 원소 판별(membership)

연산자 in을 사용하면 주어진 값이 특정 집합의 원소인지를 판단할 수 있다. 다음 코드는 문자 'S'가 CharSet 집합의 원소인지를 판단하는 예이다.

```
if 'S' in CharSet then
  // 특정 작업을 수행
```

다음은 Monday가 EnumSet의 원소가 아닌지를 판단하는 예이다.

```
if not (Monday in EnumSet) then
      // 특정 작업을 수행
```

### 합집합과 차집합

+ 연산자나 Include() 프로시저를 사용하면 집합에 원소를 추가할 수 있으며(합집합), - 연산자나 Exclude() 프로시저를 사용하면 집합에서 특정 원소들을 뺄 수 있다(차집합).

```
Include(CharSet, 'a');           // 'a'를 집합에 추가한다.
CharSet := CharSet + ['b'];      // 'b'를 집합에 추가한다.
Exclude(CharSet, 'x');           // 집합에서 'z'를 뺀다.
CharSet := CharSet - ['y', 'z']; // 집합에서 'y'와 'z'를 뺀다.
```



집합에서 하나의 원소를 추가하거나 뺄 때에는 +나 - 연산자 대신, Include()나 Exclude()를 사용하는 것이 좋다. Include()와 Exclude()는 하나의 기계어 코드만으로 구성되는 반면, +나 -는 13 + 6n(여기서 n은 집합의 비트 크기)개의 기계어 코드가 필요하다.

### 교집합

두 집합의 교집합은 * 연산자로 구할 수 있다. Set1 * Set2를 수행하면 Set1과 Set2에 공통적으로 있는 원소들만으로 구성된 새로운 집합이 생겨난다. 학창 시절 수학 시간에 집합 A와 집합 B의 교집합이 집합 A와 같으면 집합 A는 집합 B의 부분 집합이라는 공식을 배웠을 것이다. 다음 코드는 집합 사이의 포함 관계를 판단하는 예이다.

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then
      // ['a', 'b', 'c']는 CharSet의 부분 집합
```

## 객체

객체(object)란, 데이터뿐만 아니라 함수나 프로시저까지 가지고 있는 레코드라고 생각하면 된다. 델파이의 객체 모델에 대해서는 이번 장의 "델파이 객체의 활용" 부분에서 자세히 설명하므로 여기서는 오브젝트 파스칼의 객체에 관련된 기본적인 문법만 짚고 넘어가겠다. 객체는 다음과 같은 형식으로 선언한다.

```
Type
  TChildObject = class(TParentObject);
      SomeVar: Integer;
      procedure SomeProc;
  end;
```

원칙적으로, 델파이의 객체는 C++의 객체와 동일하지는 않지만, 위의 선언을 C++로 대충 옮긴다면 다음과 같다.

```
class TChildObject : public TParentObject
{
  int SomeVar;
  void SomeProc();
};
```

메소드는 일반적인 프로시저나 함수와 동일한 방식으로 정의한다("프로시저와 함수" 부분에서 다시 설명하겠다). 다만, 프로시저의 이름 앞에 객체의 이름과 마침표(.)가 붙는다는 점이 다르다.

```
procedure TChildObject.SomeProc;
begin
  { 프로시저의 코드 }
end;
```

오브젝트 파스칼의 . 기호는 비주얼 베이직의 . 연산자나 C++의 :: 연산자와 동일한 기능을 한다. 그러나 세 언어 모두 클래스를 사용할 수 있지만, 객체 지향적인 개념들을 온전히 지원하는 것은 C++와 오브젝트 파스칼뿐이라는 점을 기억하기 바란다. 그에 대해서는 "객체 지향적 프로그래밍" 부분에서 다시 이야기하겠다.



오브젝트 파스칼의 객체는 C++ 객체와는 다른 방식으로 메모리에 배치되므로 오브젝트 파스칼에서 C++ 객체를 직접 사용하거나, C++에서 오브젝트 파스칼의 객체를 직접 사용할 수는 없다. 물론, 억지로 사용하는 방법이 있긴 한데, 그에 대해서는 13장, "고난도 기법들"에서 이야기하겠다.

그러나 델파이의 쌍둥이 형제라고 할 수 있는 C++ 빌더로는 __declpec(델파이 클래스) 지시자를 적절히 사용함으로써 오브젝트 파스칼의 클래스를 C++에서 직접 생성할 수 있다. 그러한 객체들은 정규적인 C++ 객체들과는 호환되지 않는다.

## 포인터

포인터란, 메모리 주소를 담은 변수이다. 이번 장 앞에서 PChar형에 대해 이야기할 때 포인터를 잠시 언급했었다. 파스칼의 범용 포인터형은 문자 그대로 Pointer이다. Pointer에는 메모리 주소만 저장될 뿐, 그 메모리에 들어 있는 값의 데이터형이 어떤 것인지에 대한 정보는 저장되지 않으므로 종종 Pointer를 "형 없는" 포인터라고 부른다. 그러나 파스칼은 엄격한 형 검사로 유명하므로 대부분의 경우에 Pointer를 형이 있는 포인터로 사용한다.



포인터는 좀 어려운 주제이며, 포인터에 대해서 정통하지 않아도 델파이 애플리케이션을 작성하는 데 큰 무리가 없다. 물론, 포인터에 대해 익숙해지면 이전에는 할 수 없었던 작업도 쉽게 할 수 있게 될 것이다.

형 있는 포인터(typed pointer)는 Type절에서 포인터 연산자(^)로 선언한다. 형 있는 포인터를 사용하면 그 포인터가 가리키고 있는 데이터가 어떤 형인지를 컴파일러가

알기 때문에 포인터 변수로 어떤 연산을 할 수 있는지, 또 어떤 함수의 인자에 들어갈 수 있는지를 컴파일러가 점검할 수 있다. 다음은 전형적인 포인터 선언과 사용 예이다.

```
Type
  PInt = ^Integer;              // PInt는 Integer를 가리키는 포인터
  Foo = record                  // 레코드형
      GobledyGook: string;
      Snarf: Real;
  end;
  PFoo = ^Foo;
var
  P: Pointer;                   // 형 없는 포인터
  P2: PFoo;                     // PFoo의 인스턴스
```



C 프로그래머라면 오브젝트 파스칼의 ^ 연산자가 C의 * 연산자와 비슷하다는 것을 알 것이다. 오브젝트 파스칼의 Pointer형은 C의 void *형에 해당한다.

포인터 변수에 저장되는 것은 메모리 주소뿐이라는 점을 기억하기 바란다. 포인터가 가리키는 메모리 공간을 할당하는 것은 프로그래머의 책임이다. 메모리를 할당하는 것에 대해서는 표 2.6의 함수나 프로시저들을 참고하기 바란다.



포인터가 아무 것도 가리키고 있지 않은 경우(값이 0인 경우에) Nil값을 가진다고 말하며, Nil 포인터 또는 널 포인터라고 부르기도 한다.

어떤 포인터가 가리키고 있는 데이터를 사용하려면 포인터 변수 이름 뒤에 ^ 연산자를 써 주어야 한다. 이것을 "dereferencing"이라고 말한다. 다음 코드는 포인터를 사용하는 예이다.

```
Program PtrTest;
Type
  MyRec = record
      I: Integer;
      S: string;
      R: Real;
  end;
  PMyRec = ^MyRec;

var
  Rec: PMyRec;
begin
  New(Rec);        // Rec에 대한 메모리를 할당한다.
  Rec^.I := 10;// Rec에 데이터를 넣는다. ^ 연산자의 위치를 주목하기 바란다.
  Rec^.S := '포인터가 가리키는 메모리 공간에 들어가는 문자열';
  Rec^.R := 6.384;
  { Rec의 모든 필드가 채워졌음. }
  Dispose(Rec);        // 반드시 메모리를 해제해야 함.
end.
```

### New()를 사용할 수 있는 경우

크기가 정해져 있는 구조체를 가리키는 포인터에 메모리를 할당하는 경우는 New() 함수를 사용한다. 컴파일러는 그 구조체의 크기를 알고 있으므로 GetMem()이나 AllocMem() 같은 메모리 할당 함수와는 달리, New() 함수에는 메모리 크기를 써주지 않아도 된다. 따라서, 메모리 크기를 잘못 정해서 생기는 에러를 피할 수 있다. 그리고 New() 함수로 할당한 메모리는 반드시 Dispose() 함수로 해제해야 한다.

컴파일 시점에서 크기가 정해지지 않는 구조체의 경우에는 GetMem()이나 AllocMem()을 사용해서 메모리를 할당한다. 예를 들어서, PChar나 Pointer 같은 데이터형은 본질적으로 컴파일 시점에서는 메모리 크기를 알 수 없다. GetMem()이나 AllocMem()으로 포인터에 할당한 메모리 공간 이상을 사용하면 접근 위반(Access Violation) 에러가 생기므로 조심해야 한다. 그리고 GetMem()이나 AllocMem()으로 할당한 메모리는 반드시 FreeMem() 함수로 해제해야 한다. AllocMem()은 메모리를 할당한 후 할당된 메모리를 0으로 채우므로 GetMem()보다 조금은 안전하다는 점도 기억하기 바란다.

C에 익숙한 프로그래머들이 오브젝트 파스칼에 대해 불평하는 것들 중 하나는 포인터형에 대한 고지식한 형 검사이다. 예를 들어서, 다음 코드에서 변수 a와 b는 동일한 방식으로 선언되었음에도 불구하고, 데이터형이 호환되지 않는다.

```
var
  a: ^Integer;
  b: ^Integer;
```

반면, C에서는 a와 b가 호환된다.

```
int *a;
int *b;
```

오브젝트 파스칼은 선언된 포인터형을 고유한 개별적인 데이터형으로 인식하기 때문에 위와 같은 결과가 생기는 것이다. 그래서 델파이 프로그래머들은 다음과 같이 매개체로 사용할 포인터형을 쓰곤 한다.

```
type
  PtrInteger = ^Integer;    // 명명된 형을 만든다.

var
  a, b: PtrInteger;         // 이제, a와 b는 호환이 가능하다.
```

## 명명된 형(Type Alias)

오브젝트 파스칼에서는 이미 정의된 데이터형에 새로운 이름을 붙일 수 있다. 예를 들어서, 다음은 Integer형에 MyReallyNiftyInteger라는 이름을 붙이는 코드이다.

```
type
  MyReallyNiftyInteger = Integer;
```

MyReallyNiftyInteger로 선언된 변수는 Integer로 선언된 변수와 호환된다. 이름만 다를 뿐, 기능은 동일하다.

그러나 컴파일러가 명명된 형을 전혀 새롭고 고유한 데이터형으로 받아들이도록 할 수도 있다. 다음과 같이 type 예약어를 붙여주면 된다.

```
type
  MyOtherNeatInteger = type Integer;
```

MyOtherNeatInteger로 선언한 변수에 Integer로 선언한 변수의 값을 넣으면 적절한 형 변환이 일어나므로 컴파일 에러가 발생하지는 않는다. 그러나 Integer로 선언된 var나 out 인자에 MyOtherNeatInteger 변수를 넣을 수는 없다. 일단 다음 코드는 문법적으로 틀리지 않다.

```
var
  MONI: MyOtherNeatInteger;
  I: Integer;
begin
  I := 1;
  MONI := I;
```

그러나 다음 코드는 컴파일되지 않는다.

```
procedure Goon(var Value: Integer);
begin
  // 프로시저의 코드
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);     // 컴파일 에러 : M은 Integer와 var 호환이 되지 않는다.
```

델파이 컴파일러는 위와 같이 데이터형의 호환성을 엄격히 점검할 뿐만 아니라, 명명된 형에 대한 런타임형 정보도 엄격하게 생성해 낸다. 그래서 기본 데이터형에 대한 고유한 속성 편집기를 만들 수도 있는데, 그에 대해서는 22장, "고급 컴포넌트 기법"에서 설명하겠다.

## 강제 형 변환과 자동 형 변환

컴파일 시점 또는 실행 시점에서 동적으로 일어나는 형 변환을 자동 형 변환(type conversion, 묵시적 형 변환)이라고 한다면 강제 형 변환(typecasting, 명시적 형 변환)은 개발자가 코드에서 명시적으로 변수의 데이터형을 변환하는 것을 말한다. 변수 배정문이나 함수 호출의 경우, C는 데이터형의 길이만 맞으면 대충 넘어가지만, 델파이의 컴파일러는 데이터형을 매우 까다롭게 점검한다. 그래서 컴파일러의 기분을 맞춰 주기 위해서 개발자가 직접 변수의 데이터형을 변환해야 하는 일이 많이 생긴다. 다음의 코드에서 char와 byte는 내부적으로 동일한 구조의 데이터형(1바이트)이지만, 컴파일러는 그냥 넘어가지 않는다.

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := c;      // 컴파일러가 에러를 낸다.
end.
```

위의 코드가 제대로 컴파일되려면 c를 byte형으로 변환해야 한다. 강제 형 변환을 하려면 다음 코드와 같이 데이터형 이름을 쓰고, 괄호 안에 변수 이름을 써주면 된다.

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := byte(c);        // 컴파일러가 에러를 낸다.
end.
```



강제 형 변환은 두 형의 데이터 크기가 같은 경우에만 할 수 있다. 예를 들어서, Double형의 변수를 Integer로 변환할 수는 없다. 부동 소수점형을 정수형으로 변환할 때에는 Trunc()나 Round() 함수를 이용해야 한다. 정수형을 부동 소수점형으로 변환하려면 할당문을 써주면 된다(FloatVar := IntVar).

객체의 형 변환은 as 연산자를 이용하는데, 이에 대해서는 이번 장의 "런타임형 정보(Runtime Type Information, RTTI)" 부분에서 설명하겠다.

# 문자열 자원

델파이 3에서부터 resourcestring절을 통해서 오브젝트 파스칼 소스 코드 안에서 문자열 자원(resource)을 직접 정의할 수 있게 되었다(이전에는 리소스 컴파일러를 통해서 실행 파일에 링크했었다). 문자열 자원은 소스 코드에 내장되는 것이 아니라, 애플리케이션의 실행 파일이나 DLL 파일의 특정 영역에 저장되는 문자열값(보통, 사용자에게 제시하는 데 사용된다)을 말한다. 문자열들을 소스 코드와 분리할 수 있으므로 다국어를 지원하는 프로그램에서 애용된다. 문자열 자원은 다음 코드처럼 resourcestring절 안에서 식별자 = 문자열값 형태로 정의한다.

```
resourcestring
  ResString1 = '리소스 문자열 1';
  ResString2 = '리소스 문자열 2';
  ResString3 = '리소스 문자열 3';
```

문법적으로 말하자면, 문자열 자원의 문자열들은 문자열 상수와 동일한 방식으로 사용할 수 있다.

```
resourcestring
  ResString1 = 'hello';
  ResString2 = 'world';

var
  String1: string;

begin
  String := ResString1 + ' ' + ResString2;
  .
  .
  .
end;
```

컴파일 시점에서 델파이 컴파일러는 자원 문자열들을 문자열 자원에 포함시키고, 그 문자열 자원을 애플리케이션의 실행 파일(또는 DLL)에 링크한다. 실행 시점에서는 내부적으로 LoadString() API 함수를 통해서 문자열 자원에 있는 자원 문자열을 불러온 후 사용한다.

# 조건 제어문

이번에는 프로그래밍 언어의 대표적인 제어 구조인 if와 case를 살펴보자. if와 case에 대한 원론적인 이야기보다는 오브젝트 파스칼과 C, 비주얼 베이직 사이의 차이점에 초점을 두겠다.

## if문

if문은 특정한 조건에 따라 코드의 실행을 제어할 때 쓰인다. 다음은 세 언어의 if문을 비교한 것이다.

```
{ 파스칼 }
if x=4 then y := x;

/* C */
if (x==4) y = x;

' 비주얼 베이직
If x = 4 Then y = x
```



하나의 if문에서 둘 이상의 조건식을 검사하는 경우에는 각 조건식을 괄호로 싸서 명확하게 구분해 줘야 한다. 다음이 그 예이다.

```
if (x = 7) and (y =8) then
```

만일 아래와 같이 하면 컴파일러가 불평을 할 것이다.

```
if x = y and y = 8 then
```

begin과 end 키워드는 C나 C++의 {, }와 거의 같은 것이다. 예를 들어서, 다음은 주어진 조건을 만족할 때 하나 이상의 문장들을 수행하는 예이다.

```
if x = 6 then begin
  DoSomething;
  DoSomethingElse;
  DoAnotherThing;
end;
```

여러 개의 조건들을 if..else 구조로 표현할 수 있다.

```
if x = 100 then
  SomeFunction
else if x = 200 then
  SomeOtherFunction
else begin
  SomethingElse;
  Entirely;
end;
```

## case문

오브젝트 파스칼의 case문은 C나 C++의 switch문과 매우 비슷하다. if문이 참, 거짓만을 판단하는 데 비해, case문은 가능한 값이 여러 개인 조건을 처리할 수 있다. 만일 case문이 없다면 if..else if..else if를 여러 번 반복해야 할 것이다.

```
case SomeIntegerVariable of
  101 : DoSomething
  202 : begin
          DoSomething;
          DoSomethingElse;
        end;
  303 : DoAnotherThing;
  else DoTheDefault;
end;
```



case와 of 사이에 오는 선택식은 순서를 매길 수 있는 값, 즉 서수형이어야 한다. 문자열 같은 비서수형을 사용하면 안된다.

다음은 위의 코드에 해당하는 C의 switch문이다.

```
switch (SomeIntegerVariable)
{
  case 101: DoSomething; break;
  case 202: DoSomething;
            DoSomethingElse; break;
  case 303: DoAnotherThing; break;
  default: DoTheDefault;
}
```

# 반복문

반복문(loop)이란, 특정 작업을 반복할 수 있는 제어 구조를 말한다. 파스칼의 반복문 역시 다른 언어와 개념적으로 동일하므로 반복문 자체에 대해서 자세히 설명하지는 않겠다. 그냥 파스칼의 여러 반복문들을 소개하는 정도로 그치겠다.

## for 루프

특정 작업을 정해진 횟수만큼 반복할 때에는 for가 제격이다. 다음은 정수 1부터 10까지의 합을 구하는 예이다.

```
var
  I, X: Integer;
begin
  X := 0;
  for I := 1 to 10 do
      inc(X, I);
end.
```

위의 코드를 C로 옮기면 다음과 같다.

```
void main(void) {
  int x, i:
  x = 0;
  for (i=1; i<=10, i++)
      x += i;
}
```

다음은 비주얼 베이직으로 위의 코드를 구현한 것이다.

```
x = 0
For I = 1 to 10
  X = X + I
Next I
```




델파이 1.0에서와는 달리, 반복문 루프 안에서 루프 변수를 변경할 수는 없다. 델파이 2 이후의 32비트 컴파일러는 델파이 1.0과는 다른 방식으로 루프를 최적화시키기 때문이다.

## while 루프

while문은 특정 조건을 만족할 때까지 어떤 작업을 반복해야 할 때 유용하다. while 루프의 조건은 루프의 코드가 실행되기 전에 검사된다. 즉, 조건에 따라서는 루프의 코드가 한 번도 실행되지 않을 수 있는 것이다. while을 사용하는 대표적인 예는 파일의 끝(EOF)에 도달할 때까지 반복해서 파일의 내용을 읽는 것이다. 다음 코드는 텍스트 파일의 내용을 한 줄씩 읽어서 화면에 표시하는 예이다.

```
Program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile; // 텍스트 파일
  s: string;
begin
  AssignFile(f, 'foo.txt');
  Reset(f);
  while not EOF(f) do begin
      readln(f, S);
      writeln(S);
  end;
  CloseFile(f);
end.
```

파스칼의 while 루프는 C의 while이나 비주얼 베이직의 Do While 루프와 기본적으로 동일하다.

## repeat..until 루프

repeat..until 루프는 while 루프와 마찬가지로, 특정 조건이 만족될 때까지 루프 안의 코드를 반복 실행한다. 그러나 while문과는 달리, repeat..until은 조건을 루프의 마지막에 검사하기 때문에 조건에 상관 없이 적어도 한 번은 루프의 코드가 실행된다. repeat..until은 C의 do..while 루프와 비슷한 개념이다.

다음 코드는 카운터값이 100을 초과할 때까지 카운터의 값을 증가시키는 예이다.

```
var
  x: Integer;
begin
  x := 1;
  repeat
      inc(x);
  until x > 100;
end.
```

## Break() 프로시저

while이나 for, repeat 루프 안에서 Break()를 호출하면 프로그램의 제어권이 즉시 현재 루프를 벗어난다. 루프의 반복 실행을 즉시 취소해야 할 경우에 유용한 프로시저이다. 파스칼의 Break() 프로시저는 C의 Break나 비주얼 베이직의 Exit 명령문과 동일한 것이다. 다음은 루프를 다섯 번만 돌리고 Break()로 루프를 벗어나는 예이다.

```
var
  i: Integer;
begin
  for i := 1 to 100000 do
  begin
      MessageBeep(0);          // '삐' 소리를 낸다.
      if i = 5 then Break;
  end;
end;
```

## Continue() 프로시저

루프 안에서 Continue()를 호출하면 프로그램의 제어권은 루프 안의 나머지 코드를 생략하고 다음 루프 반복으로 넘어간다. 다음 코드를 자세히 분석해 보면, 첫 번째 반복(i=1)일 경우는 writeln(i, '. After continue'); 문장이 수행되지 않음을 알 수 있을 것이다.

```
var
  i: Integer;
begin
  for i := 1 to 3 do
  begin
       writeln(i, '. Before continue');
       if i = 1 then Continue;
       writeln(i, '. After continue');
  end;
end;
```

# 프로시저와 함수

프로그래머라면 프로시저와 함수에 대한 기본적인 지식을 가지고 있을 것이다. 프로시저란, 특정한 작업을 수행하는 코드의 조각 또는 프로그램의 일부분이며, 프로시저가 호출되면 프로그램의 제어권은 프로시저로 넘어가서 프로시저의 코드를 수행한 후 프로그램의 제어권을 호출한 곳으로 돌려준다. 함수 역시 동일한 개념이지만, 함수는 자신을 호출한 곳에 리턴값을 돌려주는 차이가 있다.

C나 C++에 익숙한 독자라면 파스칼의 프로시저를 리턴값이 없는(리턴값이 void인) C/C++의 함수라고 생각하면 된다. 마찬가지로, 파스칼의 함수는 리턴값이 있는 C나 C++의 함수와 동일한 개념이다.

목록 2.1은 프로시저와 함수를 포함한 짤막한 파스칼 프로그램이다.




함수와 프로시저의 예

```
Program FuncProc;

{$APPTYPE CONSOLE}

procedure BiggerThanTen(i: Integer);
{ I가 10보다 크면 화면에 문자열을 표시한다. }
begin
 if I > 0 then
     writeln('크군.');
```

```
end;

function IsPositive(I: Integer): Boolean;
{ I가 0이나 양수이면 True를 돌려주고, 음수이면 False를 돌려준다.}
begin
 if I < 0 then
     Result := False
 else
     Result := True;
 end;

var
 Num: Integer;
begin
 Num := 23;
 BiggerThanTen(Num);
 if IsPositive(Num) then
     writeln(Num, ' : 양수')
 else
     writeln(Num, ' : 음수');
 end.
```

※ 역주 : 당연한 이야기겠지만, 이 프로그램을 실행할 때 만일 한글 바이오스가 떠 있지 않으면 한글이 표시되지 않는다. 이는 이 책에 나오는 모든 콘솔 프로그램(도스 프로그램)에 해당되는 이야기이다.




IsPositive() 함수의 지역 변수 Result를 주목하기 바란다. 모든 오브젝트 파스칼 함수에는 Result라는 지역 변수가 묵시적으로 선언되어 있으며, Result는 함수의 리턴값을 의미한다. C나 C++의 경우는 함수의 실행이 끝나는 시점에서 함수의 리턴값이 결정되지만, 오브젝트 파스칼의 경우는 함수 안의 어느 곳에라도 Result를 설정할 수 있다. Result 이외에도, 함수의 이름을 함수의 리턴값으로 사용할 수 있다. 파스칼의 원래 문법이 그렇다. 델파이 5 역시 이전 버전의 볼랜드 파스칼과의 이전 버전과의 호환성을 위해 함수의 이름을 함수의 리턴값 변수로 사용할 수 있도록 한다. Project | Options | Compiler 대화상자의 Extended Syntax 옵션을 끄거나 {$X-} 지시자를 사용한 경우에는 Result를 사용할 수 없다는 점도 알아두기 바란다.

## 인자 전달 방식

파스칼은 함수나 프로시저에 인자를 넘겨줄 때 값 전달 방식과 참조 전달 방식을 모두 지원한다. 인자로 넘길 수 있는 데이터형은 기본 데이터형이나 사용자 정의형, 또는 개방형 배열(open array, 잠시 후에 설명하겠다)이다. 프로시저나 함수 안에서 변경되지 않는 인자라면 상수를 넘겨줄 수도 있다.

## 값 전달 방식

값 전달 방식은 인자 전달의 기본 방식이다. 프로시저나 함수를 선언할 때 특별한 키워드를 써주지 않는 한 기본적으로 값 전달 방식이 된다. 인자가 값으로 전달된다는 것은 변수(인자)에 담긴 데이터의 복사본이 통째로 전달된다는 뜻이다. 다음 코드를 보자.

```
procedure Foo(s: string);
```

위와 같은 방식으로 선언된 프로시저를 호출하면 문자열 s의 복사본이 만들어지며, Foo()는 그 복사본을 가지고 작업을 하게 된다. 즉, 함수에 넘겨준 원래의 문자열 s 자체는 함수 안에서의 작업에 상관 없이 변하지 않는다.

## 참조 전달 방식

참조 전달 방식은 변수 전달 방식이라고도 한다. 참조를 넘겨준다는 것은 실제 데이터가 있는 메모리 주소를 넘겨준다는 뜻이므로 함수 안에서 인자의 데이터를 변경하면 호출한 쪽에 있는 변수의 데이터 역시 변경된다. 참조 전달 방식으로 함수나 프로시저를 선언하려면 다음과 같이 var 키워드를 사용한다.

```
procedure ChangeMe(var x: longint);
begin
  x : = 2; {호출한 쪽의 x 역시 2가 된다.}
end;
```

var 키워드가 있으면 x의 복사본을 만드는 대신 인자의 주소를 넘겨주므로 호출한 쪽의 인자를 함수 안에서 직접 변경할 수 있다.

var 인자를 사용한 참조 전달 방식은 C++에서 & 연산자를 사용해서 참조를 전달하는 것과 동일한 개념이다. C++의 & 연산자와 마찬가지로, var 키워드를 사용하면 변수의 실제 값 대신 변수의 주소를 넘겨주게 된다.

## 상수 인자

함수 안에서 인자의 실제 값이 절대로 변경되지 않게 하려면 const 키워드를 사용하면 된다. const 키워드를 사용하면 인자의 실제 값이 변경되지 않을 뿐만 아니라, 문자열이나 레코드 같은 데이터형의 수행 속도를 더욱 더 최적화할 수 있다. 다음은 상수 문자열을 인자로 받아들이는 프로시저의 선언이다.

```
procedure Goon(const s: string);
```

## 개방형 배열 인자

개방형 배열 인자란, 일정하지 않은 수의 인자를 사용하기 위한 것이다. 오브젝트 파스칼에서는 동일한 데이터형으로 이루어진 개방형 배열을 넘겨줄 수도 있을 뿐만 아니라, 서로 다른 데이터형으로 이루어진 배열을 넘겨줄 수도 있는데, 데이터형이 다른 경우는 상수 배열이어야 한다. 다음은 정수 개방형 배열을 받아들이는 함수를 선언하는 예이다.

```
function AddEmUp(A: array of Integer): Integer;
```

개방형 배열 함수/프로시저에는 변수들이나 상수들, 상수 표현식들을 넘겨줄 수 있다. 다음은 AddEmUp() 함수에 여러 가지 요소들을 인자로 넘겨주는 예이다.

```
var
  i, Rez: Integer;
const
  j = 23;
begin
  i := 8;
  Rez := AddEmUp([i, 50, j, 89]);
```

개방형 배열을 받아들이는 함수/프로시저 안에서 인자로 넘어온 개방형 배열에 대한 정보를 얻으려면 High(), Low(), SizeOf() 같은 배열 관련 함수들을 이용한다. 다음 코드는 개방형 배열 A로 넘겨 받은 모든 숫자들의 합을 구하는 함수이다.

```
function AddEmUp(A: array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := Low(A) to High(A) do
      inc(Result, A[i]);
end;
```

서로 다른 데이터형으로 이루어진 개방형 배열을 인자로 받으려면 array of const로 선언해야 한다. 다음은 array of const로 프로시저를 선언하는 예이다.

```
procedure WhatHaveIGot(A: array of const);
```

위와 같이 선언했다면 다음과 같은 프로시저 호출이 가능하다.

```
WhatHaveIGot(['Tabasco', 90, 5.6, @WhatHaveIGot, 3.14159. True, 's']);
```

array of const로 선언된 함수나 프로시저에 들어온 모든 인자들은 컴파일러에 의해 TVarRec형으로 변환된다. System 유닛에서 TVarRec는 다음과 같이 정의되어 있다.

```
type
      PVarRec = ^VarRec;
      TVarRec = record
          case Byte of
              vtInteger:    (VInteger: Integer; VType: Byte);

              vtBoolean:    (VBoolean: Boolean);
              vtChar:       (VChar: Char);
              vtExtended:   (VExtended: PExtended);
              vtString:     (VString: PShortString);
```

```
                vtPointer:    (VPointer: Pointer);
                vtPChar:      (VPChar: PChar);
                vtObject:     (VObject: TObject);
                vtClass:      (VClass: TClass);
                vtWideChar:   (VWideChar: WideChar);
                vtPWideChar:  (VPWideChar: PWideChar);
                vtAnsiString: (VAnsiString: Pointer);
                vtCurrency:   (VCurrency: PCurrency);
                vtVariant:    (VVariant: PVariant);
end;
```

VType 필드는 TVarRec에 담긴 데이터의 형을 뜻한다. TVarRec에 담을 수 있는 데이터형은 다음과 같다.

```
const
{ TVarRec.VType의 값들 }
      vtInteger            = 0;
      vtBoolean            = 1;
      vtChar               = 2;
      vtExtended           = 3;
      vtString             = 4;
      vtPointer            = 5;
      vtPChar              = 6;
      vtObject             = 7;
      vtClass              = 8;
      vtWideChar           = 9;
      vtPWideChar          = 10;
      vtAnsiString         = 11;
      vtCurrency           = 12;
      vtVariant            = 13;
      vtInterfase          = 14;
      vtWideString         = 15;
      vtInt64              = 16;
```

array of const로 받은 배열을 함수나 프로시저 안에서 처리하려면 조금 복잡하리란 것은 쉽게 짐작할 수 있다. 이해를 돕기 위해서 앞에서 선언한 WhatHaveIGot()을 실제로 구현해 보겠다. 다음은 자신이 받은 개방형 배열에 포함된 각 인자들의 데이터형과 내용을 표시하는 코드이다.

```
procedure WhatHaveIGot(A: array of const);
var
  i: Integer;
  TypeStr: string;
begin
  for i := Low(A) to High(A) do
  begin
      case A[i].VType of
          vtInteger            : TypeStr := 'Integer';
          vtBoolean            : TypeStr := 'Boolean';
          vtChar               : TypeStr := 'Char';
          vtExtended           : TypeStr := 'Extended';
          vtString             : TypeStr := 'String';
          vtPointer            : TypeStr := 'Pointer';
          vtPChar              : TypeStr := 'PChar';
          vtObject             : TypeStr := 'Object';
          vtClass              : TypeStr := 'Class';
          vtWideChar           : TypeStr := 'WideChar';
          vtPWideChar          : TypeStr := 'PWideChar';
          vtAnsiString         : TypeStr := 'AnsiString';
          vtCurrency           : TypeStr := 'Currency';
          vtVariant            : TypeStr := 'Variant';
          vtInterfase          : TypeStr := 'Interfase';
          vtWideString         : TypeStr := 'WideString';
          vtInt64              : TypeStr := 'Int64';
      end;
      ShowMessage(Format('배열 항목 %d : %s', [i, TypeStr]));
  end;
end;
```

# 범위

범위(scope)란, 어떤 객체나 변수, 함수/프로시저가 프로그램의 어디서부터 어디까지 유효한가를 나타내는 개념이다. 예를 들어서, 전역 상수는 프로그램의 모든 부분에서 유효하며, 어떤 프로시저의 지역 변수는 그 프로시저 안에서만 유효하다. 목록 2.2의 코드를 보자.



목록 2.2 범위를 설명하기 위한 간단한 프로그램

```
program Foo;

{$APPTYPE CONSOLE}

const
 SomeConstant = 100;

var
 SomeGlobal: Integer;
 R: Real;

procedure SomeProc(var R: Real);
var
 LocalReal: Real;
begin
 LocalReal := 10.0;
 R := R - LocalReal;
end;

begin
 SomeGlobal := SomeConstant;
 R := 4.593;
 SomeProc(R);
end.
```

SomeConstant와 SomeGlobal, R의 범위는 전역이다. 즉, 프로그램의 모든 부분에서 컴파일러가 인식할 수 있다. 프로시저 SomeProc() 안에서 선언된 R과 LocalReal의 범위는 지역, 즉 그 프로시저 내부이다. SomeProc()의 외부에서 LocalReal을 읽거나 쓰려

고 하면 컴파일러는 알지 못하는 식별자(unknown identifier)라는 에러를 낸다. SomeProc()의 내부에서 R은 프로시저의 지역 변수 R을 의미하지만, SomeProc()의 외부에서 R은 전역 변수 R을 의미한다.

# 유닛

유닛(unit)이란, 파스칼 프로그램을 구성하는 개별 소스 코드 모듈을 뜻한다. 유닛은 메인 프로그램에서 호출할 수 있는 여러 함수나 프로시저들이 담겨 있는 곳이라고 생각하면 된다. 소스 코드 모듈이 오브젝트 파스칼의 유닛이 되려면 적어도 다음의 내용을 갖추고 있어야 한다.

- unit문. unit문은 자신이 유닛임을 밝히는 역할을 하며, 자신의 이름을 선언하는 역할도 한다. 모든 유닛의 첫 줄에는 unit문이 있어야 한다. 또한, 유닛의 이름은 파일 이름과 동일해야 한다. 예를 들어, 유닛 모듈의 파일 이름이 Foobar라고 한다면 unit문은 다음과 같다.

```
unit FooBar;
```

- interface부. unit문의 다음 줄(주석이나 빈 줄, 컴파일러 지시자 등을 제외한)에는 interface문이 와야 한다. interface문에서 implementation문 사이에 들어가는 모든 것은 메인 프로그램이 다른 유닛들과 공유할 수 있는 정보이다. interface부에서는 메인 프로그램이나 다른 유닛들에서 사용할 데이터형이나 상수, 변수, 프로시저, 함수들을 선언한다. interface 부분에는 선언만 있어야 하며, 함수나 프로시저의 코드 자체가 있어서는 안된다. interface문은 추가적인 정보 없이 단 한 줄로 쓰면 된다.

```
interface
```

- implementation부. interface부 다음에 오는 부분이다. 유닛의 implementation부는 기본적으로 함수나 프로시저의 실제 코드가 들어가며, 이 부분에서 선언되는 데이터형이나 상수, 변수들은 그 유닛의 외부와 공유하지 않는다. interface부에서 선언한 함수나 프로시저들을 실제로 정의하는 곳이 바로 implementation부이다. implementation문 역시 추가적인 정보 없이 단 한 줄로 쓰면 된다.

```
implementation
```

그리고 생략 가능한 부분들이 두 가지 있다.

- initialization부. 이 부분은 유닛 파일의 아래쪽에 위치하는데, 여기에는 유닛의 초기화 코드를 넣는다. 이 부분의 코드는 메인 프로그램이 실행되기 전에 단 한 번만 실행된다.
- finalization부. 이 부분은 initialization 부분과 유닛의 end. 사이에 위치하며, 메인 프로그램이 종료될 때 여러 가지 마무리 작업을 수행하는 코드가 들어가며, 역시 한 번만 실행된다. finalization 부분은 델파이 2.0부터 도입되었다. 델파이 1.0에서는 AddExitProc() 함수를 이용해서 추가한 종료 프로시저에서 프로그램의 마무리 작업을 했었다. 특별한 종료 프로시저가 있는 델파이 1.0의 프로젝트를 포팅해야 한다면 종료 프로시저의 코드를 finalization 부분에 옮겨야 한다.



만일 하나 이상의 유닛들에 각각 다른 initialization/finalization부들이 있다면 각 부분의 실행 순서는 컴파일러가 유닛과 만나는 순서에 따른다(프로그램의 uses 절에 있는 첫 유닛이 1번이고, 그 유닛의 uses 절에 있는 첫 유닛이 2번…). 그러므로 각 initialization/finalization 부분의 실행 순서가 중요하다면 각 유닛의 uses절을 변경할 때 각별히 주의해야 한다. 하나만 바꿔도 순서가 전혀 달라질 수 있기 때문이다.

## uses절

uses절은 C의 #include와 비슷한 개념이다. 즉, 외부 유닛의 변수나 함수/프로시저를 사용하려고 할 때 그 외부 유닛의 이름을 쓰는 곳이다. 예를 들어서, FooProg라는 프로그램이 UnitA와 UnitB에 있는 함수와 데이터형을 사용하려고 한다면 FooProg의 uses절은 다음과 같다.

```
program FooProg;

uses UnitA, UnitB;
```

한 유닛에서 uses절은 두 번, 즉 한 번은 interface부 안에, 그리고 또 한 번은 implementation부 안에 나올 수 있다.

다음은 uses절이 두 개인 유닛의 예이다.

```
Unit FooBar;
interface

uses BarFoo;

  { public 선언부 }

implementation

uses BarFly;

  { private 선언부 }

initialization
  { 유닛의 초기화 코드 }
finalization
  { 유닛의 마무리 코드 }
end.
```

## 유닛의 순환 참조

UnitA의 uses절에 UnitB가 있는데, UnitB의 uses절에도 UnitA가 있는 경우가 있다. 이러한 것을 유닛의 순환 참조라고 한다. 이러한 순환 참조가 생긴다는 것은 프로그램의 설계에 문제가 있다는 뜻이므로 순환 참조가 생기지 않도록 유닛들의 관계를 잘 설정해야 한다. 순환 참조를 해결하는 최적의 방법은 UnitA와 UnitB가 공동으로 사용할 코드를 새로운 유닛에 옮기는 것이다. 그러나 인생에서도 피할 수 없는 일이 있듯이 순환 참조를 도저히 피할 수 없는 경우가 생긴다. 그런 경우에는 uses절들 중 하나를 한 유닛의 implementation부에 옮기고, 다른 uses절은 interface부에 남겨 두면 문제를 해결할 수도 있다.

# 패키지

델파이 패키지(package)는 애플리케이션에 필요한 코드들이 독립된 파일로 분리되어 있는 일종의 라이브러리로, 하나의 패키지를 여러 애플리케이션들이 공유할 수 있다. 델파이 1이나 2에서는 패키지란 개념이 없었다.

델파이 패키지는 DPL(Delphi Package Library)이라는 확장자를 가진다. 델파이 패키지는 DLL처럼 실행 시점에서 애플리케이션에 링크될 수 있다. 실행 파일(EXE)이나 DLL에 들어갈 코드를 DPL 파일에 분리시켜 놓기 때문에 실행 파일이나 DLL의 크기가 매우 줄어든다. 델파이 패키지의 종류는 다음과 같이 네 가지이다.

1. 런타임 패키지. 이 형태의 패키지에는 애플리케이션이 실행 시점에서 사용할 유닛들이 들어 있다. 따라서, 애플리케이션이 사용할 런타임 패키지가 설치되어 있지 않으면 애플리케이션은 실행되지 않는다. 델파이의 VCL50.DPL이 런타임 패키지의 좋은 예이다.

2. 설계 시점 패키지. 이 형태의 패키지에는 컴포넌트나 속성/컴포넌트 편집기, 전문가(expert) 등 애플리케이션의 설계에 필요한 요소들이 들어 있다. 델파이 개발 환경의 컴포넌트 라이브러리에 설계 시점 패키지를 추가하려면 Component | Install Package 메뉴를 이용한다. 델파이의 DCL*.DPL 패키지들이 바로 설계 시점 패키지이다. 이 형태의 패키지에 대해서는 21장, “델파이 사용자 컴포넌트 만들기”에서 자세히 이야기하겠다.

3. 런타임/설계 시점 패키지. 이 형태의 패키지는 위의 두 형태를 모두 만족하는 패키지이다. 이런 형태의 패키지는 애플리케이션의 개발과 배포를 좀더 쉽게 하기 위한 것이지만, 편한 만큼 비효율적이다. 델파이를 사용하지 않는 최종 사용자에게 배포되는 경우에도 패키지에 델파이 개발 환경 지원 코드가 포함되어 있기 때문이다.

4. 런타임도 아니고, 설계 시점도 아닌 패키지. 이런 패키지는 다른 패키지에서 사용할 코드들이 들어 있는 형태이다. 즉, 애플리케이션이 직접 사용하지는 않는 패키지이다.

## 델파이 패키지 사용법

델파이 애플리케이션이 패키지를 이용하도록 하는 것은 어렵지 않다. Project | Options | Package 대화상자에서 Build with Runtime Packages 체크박스를 체크하기만 하면 된다. 그 체크박스가 체크된 이후 애플리케이션을 빌드(컴파일 + 링크)하면 만들어진 실행 파일의 크기가 매우 작다는 것을 알 수 있을 것이다. 런타임 패키지의 코드가 EXE나 DLL에 포함되지 않았기 때문이다. 런타임 패키지는 애플리케이션의 실행 시점에서 애플리케이션과 링크된다(따라서, 애플리케이션을 배포할 때 런타임 패키지도 배포해야 한다).

## 패키지 문법

패키지는 File | New의 New 탭에서 Package 아이콘으로 패키지 편집기(Package Editor)를 띄워서 만든다. 패키지 편집기는 델파이 패키지 소스 파일(DPK)을 만들어 낸다. DPK 파일을 컴파일하면 패키지가 만들어진다. DPK 파일은 다음과 같은 형식으로 되어 있다.

```
package PackageName

requires Package1, Package2, ...;

contains Unit1, Unit2, ...;

end.
```

requires절에는 현재 패키지를 컴파일/실행하는 데 필요한 다른 패키지들의 이름이 들어간다. 패키지를 컴파일/실행하는 데 필요한 패키지들이란, 보통 contains절에 나열된 유닛들이 사용하는(그 유닛의 uses절에 포함된) 유닛들을 담고 있는 패키지들이다. contains절에 나열된 유닛들은 현재 패키지를 구성하는 유닛들이다. contains에 나열된 유닛들이 requires절에 나열된 다른 패키지의 contains절에 나열되어 있어서는 안된다. 또한, contains절에 나열된 유닛들이 사용하는 모든 유닛들은 묵시적으로 현재 패키지에 포함된다(requires절에 포함된 패키지의 유닛들은 제외).

# 객체 지향적 프로그래밍

객체 지향적 프로그래밍(object-oriented programming, OOP)은 컴퓨터 공학 최고의 화두이다. OOP를 일종의 종교적 절대 진리로 여기는 사람들도 많이 있다. 말하자면, OOP 광신자들인 셈이다. 실용적인 컴퓨팅보다는 OOP의 진정한 구현을 목표로 하는 운영체제나 언어들도 여러 개가 나타났다가 사라졌었다. 이 책은 OOP를 다루는 책이 아니므로 여기서는 OOP 자체에 초점을 맞추기보다는 오브젝트 파스칼의 객체 지향적 성격에 초점을 맞추는 것이 좋겠다.

OOP는 분리된 객체들을 사용하는 프로그래밍 패러다임이다. 객체란, 데이터와 코드를 모두 담고 있는 개체(entity)이다. OOP 패러다임의 원래 목표가 코드를 쓰기 쉽도록 하는 것은 아니지만, 결과적으로 보면 OOP를 구현한 언어는 코드의 유지보수가 매우 쉽다. 하나의 객체가 데이터와 코드를 모두 담고 있다는 측면 때문에 디버깅이나 소스의 재사용이 쉬워지는 것이다. 전통적인 개념에서 OOP 언어의 3대 요소는 다음과 같다.

- 은폐화(encapsulation) : 정보의 입·출력(인터페이스) 이외의 구체적인 자료 구조나 코드 구현을 외부에 공개하지 않는다는 개념이다. 은폐화의 장점은 프로그램 구조가 모듈화된다는 점과 하나의 코드 조각이 다른 코드와 명확하게 분리된다는 점이다.
- 상속성(inheritance) : 부모 객체가 가지는 속성이나 행동을 이어받는 새로운 객체를 만들 수 있는 능력을 말한다. 델파이에서 상속성이 구현된 예는 VCL의 계통적 구조이다. 범용적인 객체에서 출발해서 좀더 구체적인 작업을 하는 객체들이 생겨난다. 그것이 바로 상속성이다.

  상속성의 장점은 코드의 중복을 피할 수 있다는 것이다. 그림 2.4는 상속 관계를 그림으로 표현한 것이다. 과일은 모든 과일들의 조상이며, 과일에는 참외가 포함되어 있다. 참외는 모든 종류의 참외들의 조상이며, 개구리 참외가 포함되어 있다.

- 다형성(polymorphism) : 사전적 의미로, 다형성은 "여러 형태"라는 뜻이다. 다형성이란, 동일한 이름의 메소드가 상속 관계나 인자의 종류에 따라서 다른 행동을 할 수 있는 능력을 말한다. 메소드 오버라이딩(overriding)이나 메소드 오버로딩(overloading) 등이 다형성의 예이다(오버라이딩과 오버로딩은 틀린 의미이다).

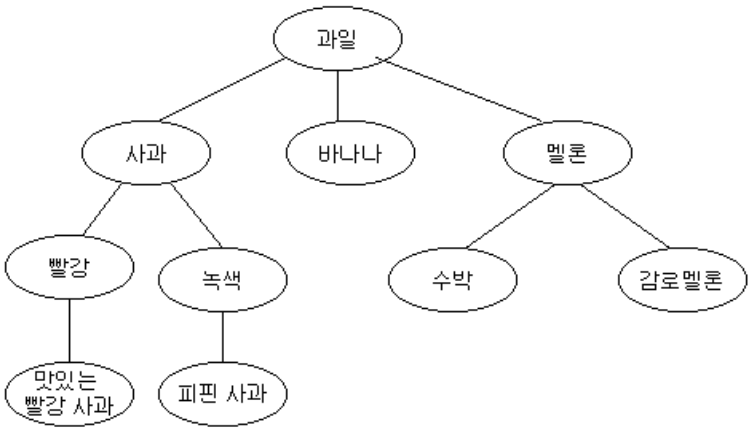



**그림 2.4**
상속성의 예

## 오브젝트 파스칼의 다중 상속

오브젝트 파스칼은 C++와는 달리, 다중 상속을 지원하지 않는다. 다중 상속이란, 말 그대로 둘 이상의 부모 객체들로부터 하나의 자식 객체가 생겨나는 것이다.
예를 들어서, 그림 2.4와 같은 과일 객체 이외에 얼음이라는 객체가 있다고 하면 과일과 얼음 객체를 다중 상속하는 과일 빙수 객체를 만들 수 있다. 다중 상속은 유용한 개념이기는 하지만, 코드의 복잡성을 증가시키는 단점이 있다.
오브젝트 파스칼은 우회적인 방식으로 다중 상속을 지원한다. 첫 번째 방법은 다른 클래스를 "포함(상속이 아님)하는" 클래스를 만드는 것이다. 델파이의 VCL이 그런 방법을 사용한다. 과일 빙수를 만들려면 빙수 객체를 과일 객체의 멤버로 만들면 되는 것이다. 두 번째 방법은 interfaces를 사용하는 것이다(interfaces에 대해서는 이번 장의 "인터페이스"부분을 보기 바란다). interfaces를 사용하면 결과적으로 빙수와 과일 객체의 인터페이스를 모두 지원하는 하나의 객체를 만들 수 있다.

다음은 객체를 이야기할 때 필수적으로 나오는 세 가지 용어들이다.

- 필드 : 필드 정의 또는 인스턴스 변수라고도 한다. 필드란, 객체에 포함되어 있는 데이터 변수를 말한다. 객체의 필드는 파스칼 레코드의 필드와 같은 개념이다. C++에서 데이터 멤버라고 하는 것이 바로 필드이다.
- 메소드 : 객체에 속한 함수나 프로시저를 메소드라고 한다. C++의 멤버 함수가 바로 메소드이다.
- 속성 : 객체에 포함되어 있는 데이터나 코드를 조작하기 위한 것이 속성이다. 일반적으로, 객체 외부에서는 객체의 필드를 직접 조작할 수 없으며, 속성을 통해서만 필드를 조작하거나 객체의 행동을 변경시킬 수 있다.



객체의 필드를 직접 노출하는 것은 OOP의 정신에 위배되는 일이다. 객체의 내부적 구현이 바뀌었을 때 객체의 필드를 직접 조작하는 외부 프로그램 역시 바뀌어야 한다면 OOP의 재사용성이나 유지보수의 편리함이 사라지는 결과를 낳는다. 따라서, 외부 프로그램과 객체의 필드 사이에서 교통 정리를 하는 속성을 사용해서 외부 프로그램으로부터 객체의 내부 구현을 최대한 은폐시켜야 한다. 속성에 대해서는 이번 장의 "속성" 부분에서 더 자세히 설명하겠다.

## 객체 기반과 객체 지향의 차이

어떤 개발 도구는 객체를 사용할 수는 있지만, 객체를 직접 만들 수는 없다. 비주얼 베이직 4와 OCX 컨트롤이 대표적인 예이다. 비주얼 베이직의 경우, 애플리케이션에서 OCX 컨트롤을 사용할 수는 있지만, OCX 컨트롤을 직접 만들 수는 없으며, 하나의 OCX 컨트롤로부터 다른 OCX 컨트롤을 상속받을 수도 없다. 그러한 개발 환경을 객체 기반(object-based) 환경이라고 한다.

델파이는 완전한 객체 지향적 환경이다. 즉, 델파이로는 전혀 새로운 객체를 만들 수도 있고, 기존 객체로부터 상속받을 수도 있다. 비주얼 객체이든, 시각적이 아닌 객체이든, 심지어는 설계 시점의 폼까지도 새로 만들거나 상속받을 수 있다.

# 델파이 객체의 활용

앞에서도 이야기했듯이 객체(프로그래밍의 시점에서는 클래스)란, 데이터와 코드를 모두 가지고 있는 개체를 말한다. 델파이의 오브젝트 파스칼은 상속, 은폐, 다형성 같은 객체 지향적 개념들을 충실히 지원한다.

## 객체의 선언과 인스턴스 생성

객체를 사용하려면 class 키워드로 객체를 선언해야 한다. 앞에서도 이야기했지만, 객체는 유닛이나 프로그램의 type절에서 선언한다.

```
type
  TFooObject = class;
```

type절에서는 객체를 클래스형으로 선언한 것이다. 다음 단계는 var절에서 그 클래스형의 객체 변수, 즉 인스턴스를 선언하는 것이다.

```
var
  FooObject: TFooObject;
```

객체 변수에는 아직 변수의 형에 대한 정보만 있을 뿐이다. 실제 데이터와 코드가 들어갈 메모리 공간을 할당하는 작업을 "인스턴스의 생성"이라고 하며, 인스턴스의 생성은 그 객체의 생성자(constructor)가 담당한다. 생성자가 하는 일은 이후에 객체를 사용할 수 있도록 객체의 각 필드를 초기화하거나, 필요한 메모리를 할당하는 것이다. 오브젝트 파스칼의 객체는 기본적으로 Create()라고 하는 생성자를 가지며, 클래스의 정의에 따라 하나 이상의 생성자를 가질 수도 있다. 객체의 형에 따라 Create()의 인자가 다르다. 이번 장에서는 인자가 없는 Create()에 초점을 맞추겠다.

C++와는 달리, 오브젝트 파스칼의 객체 생성자는 자동적으로 호출되지 않는다. 즉, 프로그래머가 직접 호출해 줘야 한다. 생성자의 호출은 다음과 같은 문법을 따른다.

```
FooObject := TFooObject.Create;
```

오브젝트 파스칼의 특이한 문법이라고 할 수 있다. 보통, 메소드를 호출할 때에는 이미 생성된 인스턴스의 메소드를 호출하는 것이 일반적이긴 하지만, 위의 코드는 인스턴스가 아니라 클래스형인 TFooObject의 메소드를 호출하고 있다. 그러나

TFooObject의 코드는 이미 메모리에 정적으로 고정되어 있는 것임을 안다면 이해가 갈 것이다.

Create의 호출 결과로 FooObject는 TFooObject 코드의 복사본을 할당받으며, 데이터 필드가 있다면 그 데이터 필드를 담을 수 있는 메모리 공간도 할당받는다.



생성자를 이용해서 객체 인스턴스를 생성하는 경우, 컴파일러는 객체의 모든 필드들을 초기화한다. 모든 숫자 필드들은 0이 되며, 포인터 필드는 nil, 그리고 문자열 필드는 빈 문자열이 된다.

## 객체의 파괴

객체를 사용한 후에는 Free() 메소드를 호출해서 객체의 인스턴스를 해제해야 한다. Free() 메소드는 우선 객체 인스턴스가 Nil인지 검사한 다음, Nil이 아니면 객체의 소멸자(destructor) 메소드인 Destroy()를 호출한다. 소멸자는 생성자의 반대 역할을 하는 것으로, 할당된 메모리들을 모두 해제하며, 그 외의 여러 가지 마무리 작업을 적절히 수행해서 객체 인스턴스의 흔적을 깨끗이 지워버린다. 객체의 파괴는 다음과 같다.

```
FooObject.Free;
```

Create()는 프로그래머가 직접 호출해야 하지만, Destroy()는 그렇지 않다. Free()의 내부에서 Destroy()를 호출하므로 프로그래머가 Destroy()를 직접 호출하는 것은 좋지 않다.



C++에서 정적으로 선언된 객체의 경우, 객체가 자신의 범위를 떠날 때 그 객체의 소멸자가 자동적으로 호출되지만, 동적으로 할당된 객체인 경우는 코드에서 직접 소멸자를 호출해야 한다. 오브젝트 파스칼에서도 기본적으로 같은 규칙이 적용되지만, 오브젝트 파스칼에서는 모든 객체들이 묵시적으로 동적이라는 점을 기억해야 한다. 따라서, 코드에서 생성한 객체는 코드에서 해제시켜야 한다. 단, 다른 객체가 소유한 객체는 직접 해제하지 않아도 된다. 소유한 객체가 해제되면 그에 포함된 객체도 자동으로 해제된다(이에 대해서는 20장, "비주얼 컴포넌트 라이브러리의 핵심 요소들"에서 설명하겠다).

그런데 메모리를 할당하고 해제하는 Create()나 Destroy() 메소드는 어디서 나온 것일까? 직접 작성해야 하다면 델파이를 포기할 독자도 많을 것이다. 다행히 앞에서 말한 메소드들은 모두 오브젝트 파스칼의 기반 객체인 TObject 객체의 메소드이다. 오브젝트 파스칼의 모든 객체들은 TObject의 후손들이다. 다음 선언을 보자.

```
Type TFoo = Class;
```

위의 코드는 사실 다음 코드를 간략하게 쓴 것이다.

```
Type TFoo = Class(TObject);
```

## 메소드

메소드(method)는 객체에 속한(엄밀히 말해서는 객체가 공개한) 함수나 프로시저를 말한다. 메소드는 객체의 성질을 뜻하는 데이터 필드가 아닌 객체의 행동을 뜻하는 코드이다. 앞에서 말한 생성자나 소멸자 역시 객체의 메소드이다. 객체의 메소드는 개발자의 필요에 따라 얼마든지 정의할 수 있다.

메소드 작성은 두 단계를 거친다. 우선, 객체의 type절에서 메소드를 선언해야 하며, type절 외부에서 메소드의 코드를 정의해야 한다. 다음은 메소드를 선언하고 정의하는 예이다.

```
type
  TBoogieNights = class
      Dance: Boolean;
      procedure DoTheHustle;
  end;

procedure TBoogieNights.DoTheHustle;
begin
  Dance := True;
end;
```

메소드의 코드를 정의할 때 메소드의 이름 앞에 클래스의 이름도 추가했음을 주목하기 바란다. 그래야 컴파일러가 어떤 클래스에 속하는 메소드인지 알 수 있기 때문이다. 그리고 메소드 안에서는 Dance 필드를 직접 조작할 수 있다는 점도 주목하기 바란다.

## 메소드의 형

메소드의 종류에는 static, virtual, dynamic, message가 있다. 다음의 예제 객체를 보자.

```
TFoo = class
  procedure IAmAStatic;
  procedure IAmAVirtual; virtual;
  procedure IAmADynamic; dynamic;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

### 정적(static) 메소드

IAmAStatic이 정적 메소드이다. 정적 메소드는 기본 메소드형이며, 일반적인 프로시저나 함수 호출과 같은 방식으로 작동한다. 정적 메소드의 주소 정보는 컴파일 시점에서 결정되므로 정적 메소드를 호출하는 코드의 주소 정보 역시 정적으로 결정된다. 따라서, 정적 메소드는 가장 빠르게 수행되지만, 주소 정보가 컴파일 시점에서 결정되므로 다형성의 메소드 재정의를 지원하지 않는다.

### 가상(virtual) 메소드

IAmAVirtual이 가상 메소드이다. 가상 메소드는 정적 메소드와 동일한 방식으로 링크되지만, 가상 메소드는 오버로딩이 가능하므로 가상 메소드를 호출하는 코드의 주소 정보는 컴파일 시점에서 결정되지 않는다. 컴파일러는 가상 메소드 테이블(Virtual Method Table, VMT)을 만들고, 가상 테이블의 위치 정보를 메소드 호출 코드에 연결한다. 실행 시점에서 가상 메소드를 호출하면 가상 메소드 테이블을 참조해서 실제 메소드의 주소 정보를 결정한다. 한 객체의 VMT에는 자신의 가상 메소드뿐만 아니라, 자신이 상속한 모든 조상 객체들의 가상 메소드 정보도 들어 있다. 따라서, 가상 메소드는 동적 메소드보다 더 많은 메모리를 소비하지만, 동적 메소드보다는 빠르게 수행된다.

### 동적(dynamic) 메소드

IAmADynamic이 동적 메소드이다. 동적 메소드는 가상 메소드와 같은 개념이지만, VMT와는 다른 방식을 사용한다. 컴파일러는 각각의 동적 메소드마다 고유한 번호를 부여하고, 그 번호와 메소드 주소 정보를 동적 메소드 테이블(Dynamic Method Table)로 만든다. VMT와는 달리, 객체의 DMT에는 객체 자신이 선언한 동적 메소드들에 대한 정보만 들어 있으며, 조상 객체의 동적 메소드를 호출한 경우는 조상 객체의 DMT를 직접 참조한다. 따라서, 가상 메소드보다는 메모리를 적게 소비하지만, 특정 동적 메소드를 찾으려면 조상 객체의 DMT를 뒤져야 하기 때문에 속도가 느리다.

### 메시지(message) 메소드

IAmAMessage는 메시지 처리 메시지이다. message 키워드 다음의 값은 그 메소드가 반응할 메시지 번호이다. 메시지 메소드는 윈도우즈 메시지를 직접 받아야 할 때 사용한다. 따라서, 프로그램이나 유닛의 코드 안에서 메시지 메소드를 직접 호출하는 경우는 거의 없다. 메시지 처리에 대해서는 5장, “메시지의 이해”에서 자세하게 이야기하겠다.

### 메소드 오버라이딩

메소드의 오버라이딩은 OOP의 다형성을 지원하기 위한 것이다. 메소드 오버라이딩이란, 조상 객체에 정의된 같은 이름의 메소드(동적 메소드나 가상 메소드)의 행동을 자손 객체에서 바꾸는 것을 말한다. 메소드를 오버라이딩하려면 조상 객체의 메소드 중 virtual이나 dynamic으로 선언된 부분을 override로 바꾸면 된다. 다음은 IAmAVirtual과 IAmADynamic 메소드를 오버라이딩하는 예이다.

```
TFooChild = class(TFoo)
  procedure IAmAVirtual; override;
  procedure IAmADynamic; override;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

override 지시자는 원래 메소드의 VMT 항목을 새로운 메소드로 대치한다. 만일 IAmAVirtual과 IAmADynamic을 override가 아닌 virtual이나 dynamic으로 다시 선언한다면 조상 객체의 메소드가 아니라, 새로운 가상 메소드와 동적 메소드를 선언하는 것이 된다. 그리고 조상 객체의 정적 메소드를 재정의하려고 하면 새 객체의 정적 메소드는 조상 객체의 정적 메소드를 완전히 대치하게 된다.

### 메소드 오버로딩

일반적인 프로시저나 함수들과 마찬가지로, 메소드 역시 오버로딩(overloading)할 수 있다. 즉, 이름은 같지만 인자는 다른 메소드를 여러 개 만들 수 있는 것이다. overload라는 지시자를 써주면 메소드가 중복될 수 있으며, 한 클래스에서 중복된 메소드들 중 첫 번째 메소드에는 overload를 써주지 않아도 된다. 다음은 메소드 오버로딩의 예이다.

```
type
  TSomeClass = class
       procedure AMethod(I: Integer); overload;
       procedure AMethod(S: string); overload;
       procedure AMethod(D: Double); overload;
end;
```

### 메소드 이름의 재도입

객체를 만들다 보면 조상 객체의 메소드를 오버라이딩하지 않고 그대로 가져와야 할 경우도 생긴다. 그렇다고 자식 객체의 메소드에 조상 객체의 메소드 이름과 동일한 메소드를 선언하면 그 메소드는 조상 객체의 메소드와는 관련이 없는 자식 객체의 독자적인 메소드로 인식된다(컴파일러도 그런 경고 메시지를 낸다). 따라서, 그 메소드가 조상 객체의 메소드를 그대로 사용한다는 것을 명시적으로 밝혀야 하는데, 다음 코드에서처럼 reintroduce 지시자를 사용하면 된다.

```
type
  TSomeBase = class
      procedure Cooper;
  end;

  TSomeClass = class
      procedure Cooper; reintroduce;
  end;
```

### Self

묵시적 변수인 Self는 객체의 모든 메소드 안에서 유효하며, 호출한 메소드가 속한 객체 인스턴스 자체를 가리키는 포인터이다. 컴파일러는 모든 메소드에 Self를 넘겨주지만, 코드에는 나타나지 않는다.

## 속성

속성(property)이란, 클래스에 담긴 데이터를 수정하거나 코드를 수행할 수 있는 특별한 종류의 필드라고 생각하면 된다. 컴포넌트의 경우, 속성은 객체 조사기(Object Inspector) 창의 목록에 나타나는 것들이다. 다음은 객체의 속성을 선언하고 정의하는 예이다.

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
public
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
      SomeValue := AValue;
end;
```

선언에서 볼 수 있듯이 TMyObject는 SomeValue라는 Integer 필드와 SetSomeValue라는 메소드, 그리고 Value라는 속성을 가진 객체이다. SetSomeValue는 SomeValue 필드의 값을 설정하는 역할만을 한다. Value 속성 자체는 아무런 값도 가지지 않는다. Value는 SomeValue 필드의 관리자 역할을 할 뿐이다. Value 속성에 어떤 값을 설정하면 SetSomeValue 메소드를 통해서 SomeValue의 값이 설정되며, Value 속성의 값을 읽으면 SomeValue의 값을 얻게 된다. 이렇게 중간 단계를 거치는 것은 다음과 같은 두 가지 장점 때문이다. 첫 번째는 클래스의 내부 구현을 최종 사용자(즉, 개발자 또는 프로그램)로부터 숨길 수 있다는 점이고, 두 번째는 부모 객체의 필드 조작 메소드(위의 예에서 SetSomeValue)를 자식 객체에서 오버라이딩할 수 있다는 점이다. 복잡한 객체 모델을 만들다 보면 이와 같은 장점이 얼마나 유용한지 알 수 있을 것이다.

## 가시 범위 지시자

다른 객체 지향 언어와 마찬가지로, 오브젝트 파스칼에서도 메소드가 사용될 수 있는 범위를 직접 지정할 수 있다. 그럴 때 사용되는 지시자는 protected, private, public, published, automated 등이다. 다음의 예를 보자.

```
TSomeObject = class
private
  APrivateVariable: Integer;
  AnotherPrivateVariable: Boolean;
protected
  procedure AProtecetedProcedure;
  function ProtectMe: Byte;
public
  constructor APublicConstructor;
  destructor ApubliKiller;
published
  property AProperty read APrivateVariable write APrivateVariable;
end;
```

각 지시자 아래에는 원하는 만큼 메소드나 필드를 추가할 수 있다. 각 지시자는 클래스 이름과 들여쓰기를 맞추는 것이 일반적인 오브젝트 파스칼 코딩 스타일이다. 다음은 각 지시자의 의미이다.

- private : 이 지시자로 선언된 메소드나 필드는 객체를 구현하는 유닛의 내부에서만 사용할 수 있다. 외부로부터 은폐해야 할 메소드나 필드는 이 지시자로 선언하면 된다. 이런 메소드나 필드는 주로 다른 메소드의 기능을 지원하는 용도로 쓰인다.
- protected : 이 지시자로 선언한 메소드나 필드는 유닛 내부뿐만 아니라, 현재 객체를 상속한 자식 객체에서도 사용할 수 있다. 최종 사용자로부터는 은폐해야 하지만, 자식 객체에게는 공개할 메소드나 필드는 protected로 선언하면 된다.
- public : 이 지시자로 선언한 메소드나 필드는 프로그램의 어느 곳에서나 사용할 수 있다. 객체의 생성자나 소멸자는 반드시 public이어야 한다.
- published : 이 지시자로 선언한 메소드나 필드에 대해서는 런타임형 정보(Runtime Type Information, RTTI)가 생성된다. 애플리케이션의 다른 코드들은 RTTI를 통해서 published 메소드나 필드를 사용할 수 있다. 델파이 개발 환경의 객체 조사기도 RTTI를 이용해서 속성 목록을 생성한다.
- automated : 이 지시자는 델파이 2와의 하위 호환성을 위해 남아 있는 것이다. 23장에서 좀더 자세히 이야기하겠다.

그럼, 앞에서 만들었던 TMyObject 클래스의 지시자를 수정해서 좀더 완벽한 클래스로 개선시켜 보자.

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
published
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
      SomeValue := AValue;
end;
```

SomeValue는 private이기 때문에 유닛의 외부에서 직접 수정할 수 없으며, 속성 Value는 런타임형 정보를 만들어내게 되었다.

### "friend" 클래스

C++ 언어에는 friend라는 키워드가 있다. friend로 선언된 클래스는 원래 클래스의 private 데이터나 함수를 사용할 수 있다. 오브젝트 파스칼에는 friend에 직접적으로 해당하는 키워드가 없지만, 비슷한 기능을 구현할 수는 있다. 오브젝트 파스칼은 한 유닛에 여러 개의 클래스를 구현할 수 있는데, 같은 유닛에서 선언된 모든 객체들은 서로 친구(friend) 사이가 되며, 다른 객체의 private 필드나 메소드를 사용할 수 있다는 점을 기억하기 바란다.

## 객체의 내부

오브젝트 파스칼의 모든 클래스 인스턴스의 실제 모습은 힙(heap) 메모리에 위치한 클래스 인스턴스 데이터 영역에 대한 32비트 포인터이다. 코드에서 어떤 클래스의 필드나 메소드, 속성에 접근하면 컴파일러는 자동적으로 그 포인터가 가리키는 곳의 코드를 생성해낸다. 따라서, 외관상으로 보기에는 클래스가 정적 변수처럼 보인다. 그러나 C++와는 달리, 오브젝트 파스칼에서는 클래스의 데이터 영역을 애플리케이션의 힙 메모리에서만 할당할 수 있으며, 애플리케이션의 데이터 세그먼트에서 할당할 수는 없다.

## TObject : 모든 객체들의 원조

모든 클래스(객체)들은 TObject로부터 비롯한 것이므로 모든 클래스는 TObject의 기능을 이어받고 있다. 객체가 객체답게 되기 위한 기본적인 기능은 모두 TObject에서 비롯된 것임을 기억하기 바란다. 클래스의 기본적인 생성이나 해제뿐만 아니라, 클래스의 이름이나 데이터형, 또는 특정 클래스로부터 상속된 것인지의 여부 등을 알 수 있는 것도 모두 TObject 덕분이다. 만일 클래스 인스턴스의 생성, 메모리 할당, 해제 등을 매번 직접 작성해야 한다면 델파이가 이처럼 인기를 끌지는 않았을 것이다.

TObject는 System 유닛에 정의되어 있으며, 컴파일러는 항상 TObject를 "인식하고" 있다. 다음은 TObject 클래스의 선언부(type)이다.

```
type
  TObject = class
      constructor Create;
  procedure Free;
      class function InitInstance(Instance: Pointer): TObject;
      procedure CleanupInstance;
      function ClassType: TClass;
      class function ClassName: ShortString;
      class function ClassNameIs(const Name: string): Boolean;
      class function ClassParent: TClass;
      class function ClassInfo: Pointer;
      class function InstanceSize: Longint;
      class function InheritsFrom(AClass: TClass): Boolean;
      class function MethodAddress(const Name: ShortString): Pointer;
      class function MethodName(Address: Pointer): ShortString;
      function FieldAddress(const Name: ShortString): Pointer;
      function GetInterface(const IID: TGUID; out Obj): Boolean;
      class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
      class function GetInterfaceTable: PInterfaceTable;
      function SafeCallException(ExceptObject: TObject;
          ExceptAddr: Pointer): Integer; virtual;
      procedure AfterConstruction; virtual;
      procedure BeforeConstruction; virtual;
      procedure Dispatch(var Message);

      procedure DefaultHandler(var Message); virtual;
      class function NewInstance: TObject; virtual;
      procedure FreeInstance; virtual;
      destructor Destroy; virtual;
 end;
```

위의 각 메소드에 대한 자세한 사항은 델파이 온라인 도움말을 참고하기 바란다.

위의 코드를 보면 class라는 키워드가 나온다. class 키워드가 붙은 메소드는 클래스의 인스턴스를 생성하지 않고도 일반적인 프로시저나 함수처럼 호출할 수 있다. 다른 메소드들은 클래스의 인스턴스 개수만큼 메모리에 존재하지만, class 메소드는 클래스당 하나만 존재한다. C++에 익숙한 독자라면 static 함수와 동일한 개념임을 알

수 있을 것이다. 만일 class 메소드를 작성할 일이 있다면 클래스의 인스턴스 정보에 의존하는 코드를 작성하지 않도록 주의하기 바란다. 인스턴스 정보에 의존하는 코드가 있으면 컴파일러가 에러를 낸다.

## 인터페이스

델파이 3에서부터 지원하기 시작한 인터페이스(interface)는 최근 몇 년간 오브젝트 파스칼 언어에 추가된 것들 중 가장 중요한 것이라 할 수 있다. 간단히 말해, 인터페이스란 객체와 상호 작용을 하기 위한 일련의 함수나 프로시저들이며, COM 기술을 지원하기 위해 도입된 것이다. 인터페이스는 말하자면, 객체와 객체의 사용자 사이의 상호 작용을 중재하는 중간 대리인 같은 존재이다. 그리고 하나의 클래스는 여러 개의 인터페이스를 구현할 수 있다.

C++에 익숙한 독자라면 인터페이스를 순수 가상(pure virtual) 클래스와 비슷한 개념이라고 생각하면 된다. 즉, 원칙적으로 말해서 인터페이스 자체는 정의만 있을 뿐, 구체적인 구현은 다른 클래스에서 이루어진다.

이번 장에서는 인터페이스의 언어적인 측면만을 다루겠다. 애플리케이션에서 인터페이스를 사용하는 방법에 대해서는 23장에서 구체적으로 이야기한다.

### 인터페이스의 정의

모든 델파이 클래스들이 묵시적으로 TObject에서 비롯된 것과 마찬가지로, 모든 인터페이스들은 IUnknown이라는 인터페이스로부터 파생된 것이다. IUnkwon은 System 유닛에 다음과 같이 정의되어 있다.

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

인터페이스의 정의는 클래스의 정의와 매우 비슷함을 알 수 있다. 클래스 정의와 눈에 띄게 다른 부분은 GUID(Globally Unique Identifier) 부분이다. GUID는 전 세계적으로 고유한 값으로, 서로 다른 인터페이스가 동일한 GUID를 가지는 경우는 없다(이론적으로). IUnkown의 정의는 마이크로소프트가 제공한 COM(Component Object Model) 사양으로부터 온 것이다. COM에 대해서도 23장에서 이야기하겠다.

델파이 클래스를 만들 줄 안다면 사용자 정의 인터페이스도 쉽게 만들 수 있다. 다음은 F1()이라는 함수를 가지고 있는 IFoo라는 인터페이스를 정의하는 코드이다.

```
type
  IFoo = interface
      ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
      function F1: Integer;
  end;
```



GUID는 개발자가 직접 지정하는 것이 아니다. 델파이 개발 환경의 경우, 코드 편집 창에서 〈Ctrl+Shift+G〉키를 누르면 새로운 GUID가 삽입된다.

다음은 IFoo를 상속하는 IBar라는 새로운 인터페이스를 정의하는 코드이다.

```
type
  IBar = interface(IFoo)
      ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
      fuction F2: Integer;
  end;
```

### 인터페이스의 구현

다음은 Ifoo와 IBar 인터페이스를 구현하는 TFooBar라는 클래스의 예이다.

```
type
  TFooBAr = class(TInterfacedObject, IFoo, IBar)
      function F1: Integer;
      function F2: Integer;
  end;

function TFooBar.F1: Integer;
begin
  Result := 0;
end;

function TFooBar.F2: Integer;
begin
  Result := 0;
end;
```

TFooBar를 선언할 때 괄호 안에 IFoo와 IBar가 명시되어 있다. 이처럼 둘 이상의 인터페이스를 지정하면 다중 인터페이스를 가지는 클래스가 된다. 인터페이스의 메소드 정보와 클래스의 메소드 정보가 일치하면 컴파일러는 인터페이스의 메소드와 클래스의 메소드를 연결한다. 클래스가 인터페이스를 구현한다고 선언은 했지만, 인터페이스의 메소드를 하나 이상 제대로 구현하지 못하면 컴파일러 에러가 발생한다.

다중 인터페이스 클래스에서 클래스가 구현해야 할 인터페이스 메소드들이 동일한 이름과 인자를 가지는 경우는 다음과 같이 메소드에 알리아스를 붙여줘야 한다.

```
type
  IFoo = interface
      ['{2137BF60-AA33-11D0-A9BF-9A4537A42701'}]
      function F1: Integer;
  end;

type
  IBar = interface
      ['{2137BF61-AA33-11D0-A9BF-9A4537A42701'}]
      fuction F1: Integer;
  end;

type
  TFooBAr = class(TInterfacedObject, IFoo, IBar)
      // 알리아스를 붙인 메소드들
      function IFoo.F1 = FooF1;
      function IBar.F1 = BarF1;
      // 인터페이스 메소드들
      function FooF1: Integer;
      function BarF1: Integer;
  end;

function TFooBar.FooF1: Integer;
begin
  Result := 0;
end;

function TFooBar.BarF1: Integer;
begin
  Result := 0;
end;
```

### implements 지시자

implements는 델파이 4 버전에서 도입된 지시자로, 이것은 인터페이스 메소드의 구현을 다른 클래스나 인터페이스에 위임하기 위한 것이다. 이러한 기법을 위임에 의한 구현(implementation by delegation)이라고 한다. implements는 다음과 같이 클래스의 속성이나 인터페이스형에 사용한다.

```
type
  TSomeClass = class(TInterfacedObject, IFoo)
  // 중간 생략
  function GetFoo: TFoo;
  property Foo: TFoo read GetFoo implements IFoo;
  // 중간 생략
end;
```

위의 코드를 컴파일할 때 컴파일러가 implements 지시자를 만나면 IFoo 인터페이스를 구현하는 메소드를 찾아서 Foo 속성과 연결시킨다. 속성의 형은 반드시 IFoo 메소드를 가지고 있는 클래스이거나 IFoo 인터페이스형, 또는 IFoo를 상속받은 인터페이스형이어야 한다. implements 다음에 여러 개의 인터페이스들을 쉼표로 분리해서 써줄 수도 있다. 그런 경우, 속성의 형은 반드시 그 인터페이스들을 모두 구현하고 있는 메소드가 있는 클래스이어야 한다.

implements 지시자의 장점은 크게 두 가지이다. 우선, implements 지시자를 이용하면 쉽게 집성(aggregation)을 수행할 수 있다. 집성이란, 여러 개의 클래스를 하나의 목적으로 결합하는 것을 뜻하는 COM의 개념이다(23장에서 집성에 대해 이야기하겠다). 두 번째로, 인터페이스의 구현이 실제로 필요해지기 전까지 인터페이스의 구현에 필요한 자원의 소비를 미뤄둘 수 있다. 예를 들어서, 기능을 구현하려면 1MB의 비트맵이 필요한 인터페이스가 있는데, 그 인터페이스가 자주 쓰이지는 않는다고 하자. 쓰이지도 않을 인터페이스가 항상 1MB를 할당한 상태로 구현하는 것은 낭비이다. implements를 이용해서 속성과 인터페이스를 연결시킨 클래스를 만들면 그 속성이 실제로 사용되는 경우에만 인터페이스가 구현되므로 자원의 낭비를 피할 수 있다.

### 인터페이스의 사용법

인터페이스형의 변수를 사용하는 것은 클래스를 사용하는 것과 그리 다르지 않지만, 중요한 규칙들이 몇 가지 있다. 우선, 인터페이스는 수명 관리형이다. 즉, 오브젝트 파스칼은 인터페이스형을 항상 nil로 초기화하며, 참조 횟수를 관리한다. 그리고 인터페이스 변수가 자신의 범위를 벗어나거나 변수에 nil값을 배정하면 자동적으로 메모리가 해제된다. 다음 코드는 인터페이스 변수의 수명 관리를 보여주는 예이다.

```
var
  I: ISomeInterface;
begin
  // I는 nil로 초기화된다.
  I := FunctionReturningAnInterface;
  // I에 인터페이스가 할당되면 I의 참조 횟수가 하나 증가한다.

  I.SomeFunc;  // I의 참조 횟수가 하나 감소한다. 만일 참조
                // 횟수가 0이 되면 자동으로 해제된다.
end;
```

그리고 또 한 가지 규칙은 인터페이스에 그 인터페이스를 구현한 클래스를 직접 배정할 수 있다는 점이다. 예를 들어서, 다음 코드는 앞에서 정의한 TFooBar 클래스를 사용한 것인데, 에러가 나지 않는다.

```
procedure Test(FB: TFooBar)
var
  F: IFoo;
begin

  F := FB; // FB가 IFoo를 구현하고 있으므로 유효한 문장이다.
  .
  .
  .
```

마지막으로, 형 변환 연산자 as를 사용해서 주어진 인터페이스 변수를 다른 인터페이스로 변환할 수 있다. 내부적으로는 QueryInterface가 수행된다(이에 대해서도 23장에서 설명하겠다). 다음이 그 예이다.

```
var
  FB: TFooBar;
  F: IFoo;
  B: IBar;
begin
  FB := TFooBar.Create;
  F := FB; // FB가 IFoo를 구현하고 있으므로 유효한 문장이다.
  B := F as IBar;  // QueryInterface로 해당 인터페이스를 얻는다.

  .
  .
  .
```

만일 인터페이스를 지원할 수 없으면 예외가 발생한다.

# 구조적 예외(exception) 처리

예외란, 애플리케이션 내에서 복구할 수 있는 에러를 말한다. 구조적 예외 처리(structured exception handling, SEH)란, 심각한 에러가 아닌 에러를 애플리케이션 안에서 자연스럽게 처리할 수 있는 기법이다. 델파이 1.0의 경우에 예외를 오브젝트 파스칼 언어로 구현했지만, 델파이 2.0부터는 Win32 API로 구현하고 있다. 오브젝트 파스칼의 예외는 클래스 형태를 가지며, 예외 클래스들로부터 에러의 특성이나 발생한 위치 등의 정보를 얻을 수 있으므로 예외 처리를 매우 편리하게 할 수 있다. 예외조차도 클래스로 다룰 수 있다는 점은 오브젝트 파스칼의 객체 지향적 성격을 더욱 빛나게 하는 장점이다.

델파이에는 메모리 부족이나 0으로 나누기, 연산 오버플로우나 언더플로우, 파일 I/O 에러 같은 일반적인 에러 조건들에 대한 예외가 정의되어 있다. 또한, 개발자가 직접 예외 클래스를 정의할 수도 있다.

목록 2.3은 파일 I/O 과정에서 발생한 예외를 처리하는 예이다.

목록 2.3 예외 처리를 포함하는 파일 I/O 코드

```
program FileIO;

uses Classes, Dialogs'

{$APPTYPE CONSOLE}

var
 F: TextFile;
 S: string;
begin
 AssignFile(F, 'FOO.TXT');
 try
      Reset(F);
      try
           readln(F, S);
      finally
           CloseFile(F);
      end;
 except
      on EInOutError do
           ShowMessage('Error Accessing File!');
 end;
 end.
```

목록 2.3에서 안쪽의 try..finally 블록은 예외가 발생하든, 안하든 파일은 반드시 닫도록 보장하는 역할을 한다. 그 블록을 말로 표현하자면, "프로그램아, 프로그램아, try와 finally 사이의 코드를 수행하거라. 수행이 끝나거나 예외가 발생하면 finally와 end 사이의 코드를 수행해야 하느니라. 그리고 예외가 발생했다면 그 다음 예외 처리 블록으로 넘어가야 하느니라"이다. 결과적으로, 파일은 반드시 닫히며, 어떠한 에러가 발생하더라도(운영체제 자체의 오류가 아니라면) 프로그램 안에서 적절히 처리된다.



try..finally 블록의 finally 다음에 있는 코드는 예외가 발생하지 않더라도 수행된다. 따라서, finally 다음에 예외가 발생했다고 가정한 상황의 코드가 와서는 안된다. 또한, finally문은 예외 조건을 해제하지 않으므로 예외가 발생한 경우는 프로그램의 제어권이 다음의 예외 처리 부분으로 넘어간다.

바깥쪽 try..except 블록은 프로그램에서 발생한 예외를 처리하는 역할을 한다. 예외가 발생하면 finally 블록이 파일을 닫고, except 블록이 I/O 에러가 발생했다는 메시지를 표시하는 것이다.

이러한 예외 처리 방식이 기존의 예외 처리 방식보다 우월한 이유 중 하나는 에러 감지 코드와 에러 수정 코드를 명확히 분리할 수 있다는 것이다. 에러의 감지와 에러의 수정은 다른 개념이므로 두 블록을 명확히 분리할 수 있다는 것은 코드를 읽기 쉽고, 유지보수하기 쉽게 만들어준다.

try..finally 블록에서 특정한 예외를 꼭 집어서 잡아낼 수는 없다. 어떠한 예외가 발생하더라도 일단은 finally로 가게 된다. try..finally 블록을 사용한다는 것은 어떠한 예외가 생겨도 심각한 상황까지는 가지 말아야 한다는 뜻이므로 finally 블록에서는 주로 자원이나 메모리의 해제를 수행한다. 다시 말하면, finally 블록에는 예외가 생기든, 안생기든 반드시 수행해야 할 코드를 집어넣어야 한다. 반면, 특정한 예외 상황을 바로잡기 위해서는 try..except 블록을 사용한다. 목록 2.4는 try..except 블록의 예이다.

**목록 2.4** try..except 예외 처리 블록

```
Program HandleIt;

{$APPTYPE CONSOLE}

var
 R1, R2: Double;
begin
 while True do begin
 try
     Write('실수를 입력하세요: ');
```

```
      ReadLn(R1);
      Write('실수를 또 하나 입력하세요: ');
      Writeln('이제, 첫 번째 수를 두 번째 수로 나눕니다...');
      Writeln('나누기 결과: ', (R1 / R2):5:2);
  except
      On EZeroDivide do
          Writeln('0으로 나누면 안되죠!');
      On EInOutError do
          Writeln('실수가 아니군요!');
  end;
end;
end.
```

try..except 블록에 else를 사용할 수도 있다. 특정한 예외 이외의 모든 예외들에 대해서 수행할 작업이 있다면 else절에 써주면 된다. 다음이 그 예이다.

```
try
  // 작업
except
  On ESomeException do
      // 특정 예외에 대한 처리
  else
      // 기본 예외 처리
end;
```




try..except..else 구조를 사용할 때 else는 모든 예외를 잡아낸다는 점을 기억하기 바란다. 즉, 메모리 부족이라든지, 런타임 라이브러리 예외들까지도 else문에서 잡아낸다. 따라서, else는 충분한 고려 후에 사용해야 한다. 무조건적인 예외 루틴을 사용할 때에는 항상 예외를 다시 발생시켜야 한다. 이에 대해서는 "예외의 재발생" 부분에서 자세히 설명하겠다.

try..except 블록에서 예외 클래스를 지정하지 않으면 try..except..else 구조와 같은 효과가 생긴다. 다음이 그 예이다.

```
try
  // 작업
except
  예외 처리     // else와 거의 같은 효과
end;
```

## 예외 클래스

예외란, 단지 객체들의 특별한 인스턴스일 뿐이다. 예외가 발생하면 해당 예외 객체의 인스턴스가 생성되며, 예외가 처리되면 인스턴스도 파괴된다. 기반 예외 객체는 Exception이며, 다음과 같이 정의되어 있다.

```
type
  Exception = class(TObject)
  private
       FMessage: string;
       FHelpContext: Integer;
  public
       constructor Create(const Msg: string);
       constructor CreateFmt(const Msg: string; const Args: array of const);
       constructor CreateRes(Ident: Integer);
       constructor CreateResFmt(Ident: Integer; const Args: array of const);
       constructor CreateHelp(const Msg: string; HelpContext: Integer);
       constructor CreateFmtHelp(const Msg: string; const Args: array of
            const; HelpContext: Integer);
       constructor CreateResHelp(Ident, HelpContext: Integer);
       constructor CreateResFmtHelp(Ident: Integer; const Args: array of
            const; HelpContext: Integer);
       property HelpContext: Integer read FHelpContext write FHelpContext;
       property Message: string read FMessage write FMessage;
end;
```

Exception 객체에서 중요한 요소는 string형인 Message 속성이다. Message에는 예외에 대한 정보나 설명이 들어 있다. Message가 제공하는 정보는 발생한 예외의 종류에 따라 다르다.



예외 객체를 직접 정의할 때에는 Exception(또는, 그 자손 객체들) 같은 미리 정의된 객체로부터 상속받아야 한다. 일반 예외 처리 블록에서도 개발자가 직접 정의한 예외를 잡아낼 수 있으려면 그렇게 해야 한다.

except 블록에서 특정 종류의 예외를 처리하는 경우, 그 종류의 예외로부터 파생된 다른 예외들도 잡아낸다. 예를 들어서, EMathError를 잡아내는 예외 처리 블록은 그의 자손들인 EZeroDivide나 EOverflow도 잡아낸다. 다음 코드는 EMathError를 잡아내는 예이다.

```
try
  // 작업
except
  on EMathError do // EmathError뿐만 아니라, 자손 예외 객체들도 모두 잡는다.
end;
```

코드에서 명시적으로 처리하지 않는 다른 예외들은 모두 델파이 런타임 라이브러리의 기본 예외 처리부로 넘어간다. 델파이의 기본 예외 처리부는 메시지 상자로 예외가 발생했음을 알려준다. 4장, “애플리케이션 프레임웍과 설계 개념들”에서 델파이의 기본 예외 처리 동작을 변경하는 방법을 설명하니 참고하기 바란다.

예외를 처리할 때 예외의 Message 속성이나 그 외의 기타 속성들을 참조하여 발생할 예외 객체의 인스턴스를 얻어야 한다. 예외 객체의 인스턴스를 얻는 방법은 두 가지이다. 첫 번째는 on ESomeException 구조에서 추가적인 식별자를 사용하는 것이고, 두 번째는 ExceptObject() 함수를 사용하는 것이다.

except 블록의 on ESomeException 부분에 식별자를 넣으면 현재 발생한 예외의 인스턴스를 얻을 수 있다. 다음과 같이 예외의 형 앞에 임의의 식별자와 콜론(:)을 붙이면 된다.

```
try
  // 작업
except
  on E:ESomeException do
       ShowMessage(E.Message);
end;
```

식별자(위의 경우, E)는 현재 발생한 예외의 인스턴스가 된다. 그 식별자는 항상 : 뒤에 오는 예외와 동일한 형이다.

ExceptObject() 함수는 현재 발생된 예외의 인스턴스를 돌려주는 함수이며, 인자는 없다. ExceptObject()는 예외를 TObject형으로 돌려주므로 다음 코드와 같이 직접 형 변환을 해줘야 하는 게 조금 불편한 점이다.

```
try
  // 작업
except
  on ESomeException do
       ShowMessage(ESomeException(ExceptObject).Message);
end;
```

현재 발생한 예외가 없는 경우, ExceptObject()는 Nil을 돌려준다.

예외를 발생시키는 문법은 객체의 인스턴스를 생성시키는 문법과 비슷하다. 다음은 EBadStuff라는 사용자 정의 예외를 발생시키는 예이다.

```
Raise EBadStuff.Create('뭔가 잘못 되었음.');
```

## 예외의 흐름

하나의 예외가 발생하면 프로그램의 실행 흐름은 예외 인스턴스가 최종적으로 처리되어서 파괴될 때까지 차례로 다음의 예외 처리부들에 전파된다. 그 과정은 호출 스택을 통해서 결정되므로 예외 처리의 범위는 프로그램 전체가 된다(즉, 한 프로시저나 한 유닛에 국한되지 않는다). 목록 2.5는 예외가 발생했을 때 프로그램의 실행이 어떻게 흘러가는지를 보여주는 코드이다. 이 목록은 버튼이 하나 있는 폼 하나로 구성된 델파이 애플리케이션의 메인 유닛이다. 버튼을 클릭하면 Buuron1Click() 메소드는 Proc1()을 호출하고, Proc1()은 Proc2()를 호출하며, Proc2()는 Proc3()을 호출한다. 예외는 Proc3()에서 발생하는데, 각 예외 처리 블록에서 메시지 상자를 표시하므로 예외의 흐름을 알 수 있다. 예외가 최종적으로 처리되는 곳은 Button1Click()의 try..finally 블록이다.



목록 2.5를 델파이 개발 환경에서 실행시키는 경우, Tools | Debugger Options | Preferences | Break on Exception 체크박스를 지워서 디버거가 예외를 가로채지 못하도록 해야 예외의 흐름을 확실하게 볼 수 있다.



목록 2.5 예외가 발생했을 때의 프로그램 실행

```
unit Main;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

type
  EBadStuff = class(Exception);

procedure Proc3;
begin
  try
    raise EBadStuff.Create('결국 여기까지 왔군요!');
  finally
    ShowMessage('Proc3에서 예외 발생!');
  end;
end;

procedure Proc2;
begin
  try
    Proc3;
  finally
    ShowMessage('Proc2에서도 예외 발견!');
  end;
end;

procedure Proc1;
begin
  try
    Proc2;
  finally
    ShowMessage('Proc1에서도 예외 발견!');
```

```
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
const
  ExceptMsg = '호출한 프로시저에서 예외를 처리했음.'
    + ' 예외 메시지 : "%s"';
begin
  ShowMessage('이 메소드는 Proc3을 호출하는 Proc2를 호출하는'
    + ' Proc1을 호출함.');
  try
    Proc1;
  except
    on E:EBadStuff do
      ShowMessage(Format(ExceptMsg, [E.Message]));
  end;
end;

end.
```

## 예외의 재발생

프로그램을 만들다 보면 기존 try..except 블록 안에서 예외에 대해 특별한 처리를 한 후에도 예외를 끝내지 않고 다음 예외 처리부로 예외를 넘겨야 하는 경우도 생긴다. 그런 기법을 예외 재발생(reraising exception)이라고 한다. 목록 2.6은 예외 재발생의 예이다.

목록 2.6 예외의 재발생

```
try       // 바깥쪽 예외 처리 블록
 { 작업 }
 { 작업 }
 { 작업 }
 try      // 안쪽 예외 처리 블록
     { 특별한 처리가 필요한 작업 }
 except
```

```
    on ESomeException do
    begin
        { 안쪽 블록에서 예외에 대한 특정 처리를 수행 }
        raise;       // 바깥쪽 블록에 대해 예외를 다시 발생시킨다.
    end;
end;
except
    // 바깥쪽 블록에서 예외에 대한 기본 처리를 수행
    on ESomeException do { 기본 처리 작업 }
end;
```

# 런타임형 정보(Runtime Type Information, RTTI)

런타임형 정보란, 실행 시점에서 객체에 대한 정보를 얻기 위한 델파이의 기능이다. RTTI는 델파이 컴포넌트와 델파이 개발 환경을 연결시켜 주는 역할도 하지만, 개발자도 얼마든지 활용할 수 있다는 점이 중요하다.

델파이의 모든 객체들은 TObject의 자손들이며, 각각의 객체마다 자신의 RTTI에 대한 포인터가 담겨 있다. 그리고 RTTI의 정보를 얻는 데 필요한 몇 가지 메소드들을 TObject로부터 물려받는다. 다음은 RTTI에서 특정 객체 인스턴스에 대한 정보를 뽑아내는 데 필요한 TObject의 메소드들이다.

| 함수 | 리턴형 | 리턴값 |
|---|---|---|
| ClassName() | String | 객체의 클래스 이름 |
| ClassType() | TClass | 객체의 형 |
| InheritsFrom() | Boolean | 현재 클래스가 인자로 제공한 클래스의 후손인지 여부 |
| ClassParent() | TClass | 부모 객체의 형 |
| InstanceSize() | Word | 인스턴스의 크기(바이트 단위) |
| ClassInfo() | Pointer | 메모리에 있는 객체의 RTTI에 대한 포인터 |

객체의 안전한 형 변환을 위해서는 as 연산자를 사용한다. as를 사용하면 조상쪽 객체를 후손쪽 객체로 변환할 수 있으며, 변환을 할 수 없는 경우이면 예외가 발생한다. 예를 들어서, 다음과 같이 어떤 형의 객체라도 받아들일 수 있는 함수가 있다고 하자.

```
Procedure Foo(AnObject: TObject);
```

인자 AnObject로 어떤 특정한 작업을 하려면 그 작업에 맞게 AnObject를 구체적인 객체로 변환시켜야 한다. 예를 들어서, 호출한 쪽에서 AnObject로 텍스트 상자(TEdit)를 보낸다면 다음과 같은 코드를 통해서 텍스트 상자의 내용을 바꿀 수 있다.

```
(AnObject as TEdit).Text := 'Hello World.';
```

형 비교 연산자인 is는 두 객체의 형이 같으면 True, 그렇지 않으면 False를 돌려준다. 위의 함수에서 만일 AnObject가 TEdit가 아니라면 문제가 생긴다. 따라서, 주어진 객체 AnObject가 특정한 형의 객체 TEdit인지 확인하는 작업이 필요하다. 그럴 때 다음과 같이 is 연산자를 이용하면 된다.

```
If (AnObject is TEdit) then
  TEdit(AnObject).Text := 'Hello World.';
```

이번에는 as를 쓰지 않았다. if문에서 이미 is로 형을 검사했으므로 굳이 as를 다시 쓸 필요가 없기 때문이다. as는 실행 시점에서 RTTI를 참조해야 하기 때문에 일반적인 형 변환보다 더 느리다.

## 요약

무턱대고 델파이 개발 환경을 주무르는 것보다는 델파이의 기반인 오브젝트 파스칼에 대해 충분히 알아두는 것이 실력 향상에 도움이 된다. 이번 장에서는 변수, 연산자, 함수, 프로시저, 형, 제어문 등에 대해 이야기했다. 또한, OOP와 객체, 필드, 속성, 메소드, TObject, 인터페이스, 예외 처리, RTTI에 대해서도 이야기했다.

이제, 델파이의 객체 지향적 언어인 오브젝트 파스칼에 대한 전반적인 윤곽이 생겼을 것이다. 그럼 좀더 나아가서 Win32 API와 VCL(Visual Component Library)에 대해 알아보자.