19장 테스팅과 디버깅

* 일반적인 프로그램 버그들
* 내장 디버거 사용법
* 다중 프로세스/리모트 디버깅

프로그래밍 실력과 경험만 충분하다면 디버깅 능력을 키우려고 노력할 필요는 없다고 말하는 개발자들도 있다. 그러나 프로그래밍 실력만큼이나 디버깅 실력도 뛰어나다면 더 바랄 것이 없을 것이다. 특히, 하나의 프로젝트를 여러 개발자가 함께 진행하는 경우, 팀에 디버깅 전문가가 하나 끼어 있으면 정말 눈물나게 고마운 일이 생기게 마련이다. 어쨌든, 인간이 하는 일에는 실수가 없을 수 없다는 점을 강조하고 싶다.

그런데 "내 코드를 컴파일할 때 아무 에러도 없었으니까 내 프로그램에는 버그가 없어"라고 말하는 사람들이 상상외로 많다는 사실이 놀랍다. 그 말은 틀린 말이다. 프로그램의 컴파일 에러와 프로그램의 버그는 전혀 다른 것이다. 문법적으로 올바른 코드와 버그가 없는 코드, 즉 논리적으로 올바른 코드 사이에는 상당히 큰 차이가 있다. 또한, 어떤 코드가 오늘 우리 집 컴퓨터에서 제대로 돌아간다고 해서 그 코드가 내일 옆집 컴퓨터에서도 잘 돌아갈 것이라고 생각해서는 안된다. 디버깅의 세계에서는 한 줄 한 줄의 코드가 모두 의심의 대상이 된다.

애플리케이션의 개발 과정에서는 컴파일러가 제공하는 버그 점검 기능을 얼마든지 활용할 수 있다. 예를 들어서, Project | Options 대화상자의 Complier 페이지를 통해서(그림 19.1), 또는 코드에 적절한 지시문을 써서 런타임 에러 점검 기능을 활성화시키면 된다. 그리고 Show Hints나 Show Warning 옵션을 활성화시키면 컴파일러가 코드의 잠재적인 버그들에 대한 정보를 제공한다. 사실, 사람의 일반적인 심리를 볼 때 "일단은 제대로 돌아가는 애플리케이션"의 코드에서 굳이 버그를 찾아내려고 노력하기는 힘든 일이다. 컴파일러는 사람보다 더 꼼꼼하게 버그 가능성을 찾아내므로(물론, 사람만큼 지능적이지는 않지만) 기본적인 코드의 오류는 컴파일러에게 의존하는 것도 나쁘지 않다.

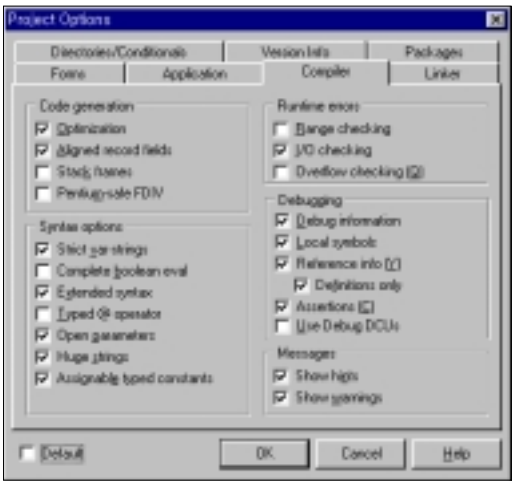


그림 19.1
Project Options 대화상자의 Complier 페이지

표 19.1은 델파이의 런타임 에러 점검 옵션들에 대한 설명이다.

표 19.1 델파이 런타임 에러 점검 옵션들

런타임 에러 옵션	지시자	기능
Range Checking	{$R+}	배열 색인이 배열의 범위를 넘어가지 않도록 하거나, 스칼라 변수에 배정하는 값이 그 변수의 범위를 넘어가지 않도록 하는 등 범위에 관련된 에러를 점검한다.
Stack Checking	{$S+}	사용할 수 있는 스택보다 더 많은 스택을 사용하지 않도록 점검한다.
I/O Checking	{$I+}	I/O 루틴(ReadLn(), WriteLn() 등등)을 호출할 때마다 입·출력 에러를 점검한다. 이 옵션은 항상 켜 놓는 것이 좋다.
Overflow Checking	{$Q+}	연산 결과가 레지스터 크기를 넘지 않도록 점검한다.



이 옵션들을 선택해 놓으면 프로그램의 수행 성능에 손해를 본다는 점을 기억하기 바란다. 일반적으로, 개발 과정에서는 옵션들을 모두 선택하고, 디버깅이 완료되어서 정식 제품을 빌드할 때에는 옵션들을 제거하는 방식을 많이 사용한다. 최종 제품에는 I/O Checking을 제외한 모든 옵션들을 제거하는 것이 관례이다.

일반적인 프로그램 버그들

이번에는 개발자들이 흔히 저지르는 실수들에 대해서 이야기하겠다. 다음에 나오는 내용들을 염두에 두고 코드를 작성한다면 버그를 대폭 줄일 수 있을 것이다.

생성되지도 않은 클래스 변수를 사용하는 경우

델파이의 클래스들을 다룰 때 흔히 저지르는 실수는 아직 생성되지 않은 클래스 인스턴스의 속성이나 메소드를 사용하는 것이다. 다음 코드를 보자.

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList.Assign(ListBox1.Items);
end;
```

MyStringList라는 TStringList 변수는 선언만 되었을 뿐인데, Assign() 메소드를 호출해서 뭔가 복사하려고 하는 코드이다. 이렇게 하면 접근 위반(access violation)이 발생한다. 일단 다음과 같이 Create()로 인스턴스를 생성해야 하는데, 다음 코드 역시 버그가 포함되어 있다. 찾아보기 바란다.

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;
  MyStringList.Assign(ListBox1.Items);
end;
```

일단 Create를 넣었기 때문에 접근 위반은 생기지 않지만, 그래도 버그가 있다. MyStringList는 이 프로시저의 지역 변수이므로 프로시저 안에서 해제해 줘야 하는데, 그 부분이 빠져 있다. 사실, 아무런 문제 없이 실행되는 버그가 더 무서운 법이다. 이 프로시저를 백 번 호출한다면 백 개의 TStringList 인스턴스가 생성되며, 그 인스턴스들에 할당된 메모리는 아무런 역할도 하지 않고 그냥 자리만 차지하고 있는 셈이 된다(이런 것을 메모리가 샌다고 말한다). 물론, 애플리케이션이 종료되면 Win32가 알아서 애플리케이션의 메모리 공간을 비우긴 하지만, 애플리케이션의 실행중에 계속 메모리가 새는 것은 그냥 넘어갈 문제가 아니다. 예를 들어서, 한 애플리케이션이 계속 메모리를 잡아먹다 보면 운영체제 전체의 메모리가 고갈되어서 디스크 스와핑을 위해 하드 디스크를 계속 긁어댈 것이다. 그러면 컴퓨터를 아예 다시 부팅시키는 편이 낫다.

앞의 코드에서 버그를 제거한 코드가 다음에 나와 있다. 물론, 이 코드도 좀더 개선시킬 수 있다.

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create; // 생성
  MyStringList.Assign(ListBox1.Items); // 사용
  { TStringList 인스턴스를 사용하는 코드가 여기에 들어간다. }
  MyStringList.Free; // 해제
end;
```

클래스 인스턴스의 해제를 보장하는 방법

위의 코드는 일견 완벽해 보이는 것 같지만, TStringList가 생성된 다음에 예외가 생겼다고 생각해 보자. 그러면 프로시저는 끝까지 수행되지 못하고 종료되므로 Free() 메소드가 실행되지 않는다. 결국, 메모리가 새게 된다. 클래스 인스턴스를 사용할 때에는 항상 try..finally 구조를 이용해서 인스턴스의 해제를 보장하는 것이 좋다. 다음이 그 예이다.

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create; // 생성
  try
      MyStringList.Assign(ListBox1.Items); // 사용
      { TStringList 인스턴스를 사용하는 코드가 여기에 들어간다. }
  finally
      MyStringList.Free; // 해제
  end;
end;
```

이번 장의 "정지점" 부분을 읽고 난 후에 다음의 문장을 MyStringList.Assign 바로 아래 줄에 삽입해 보기 바란다.

```
raise Exception.Create('테스트 예외');
```

그런 다음, 메소드의 첫 번째 줄에 정지점을 설정하고 한 줄씩 코드를 수행시켜 보기 바란다. 예외가 발생해도 TStringList는 해제가 된다는 것을 알 수 있을 것이다.

포인터 다루기

C/C++에서 버그의 주범은 항상 포인터라는 말에 동감하는 사람이 많을 것이다. 델파이에서도 포인터를 만만하게 보면 안된다. 포인터 관련 버그는 포인터가 유효한 메모리 공간을 가리키지 않기 때문에 생기는데, 포인터가 유효한 메모리 공간을 가리키지 않은 이유는 크게 두 가지이다. 하나는 초기화되지 않은 포인터를 사용하기 때문이고, 또 하나는 이미 폐기된 포인터를 사용하기 때문이다.

포인터가 초기화되지 않았다는 것은 포인터 변수에 아직 메모리가 할당되지 않았다는 것을 말한다. 그 상태에서는 포인터가 무엇을 가리키고 있는지 알 수 없다. 다음이 초기화되지 않은 포인터를 사용하는 코드이다.

```
var
  P: ^Integer;
begin
  P^ := 1971; // P는 아직 초기화되지 않았음.
```

폐기된 포인터란, 이전에는 할당되어 있었지만 이제는 해제된 메모리 블록을 가리키고 있는 포인터를 말한다. 다음 예를 보자.

```
var
  P: ^Integer;
begin
  New(P);
  P^ := 1971;
  Dispose(P);
  P^ := 4; // P는 폐기되었음.
```

이 코드를 수행할 때 접근 위반이 발생하면 적어도 뭔가 잘못되었다는 것을 알 수 있기 때문에 다행이지만, 그렇지 않다면 4라는 값이 다른 중요한 데이터를 덮어썼을지도 모르므로 애플리케이션의 전반적인 결과가 의심스럽게 된다. 이런 종류의 버그

는 정말 잡아내기 힘들다. 어떤 컴퓨터에서는 아무런 문제 없이 돌아갈 수도 있지만, 다른 컴퓨터에서는(또는, 같은 컴퓨터라도 프로그램이 다른 과정을 거쳐서 실행되었다면) 다운이 되거나 잘못된 결과를 보일 수 있다. 그러한 문제에 빠진다면 지금까지 배웠던 여러 가지 멋진 프로그래밍 기법들이 무용지물이 된다. 코드 한 줄 한 줄마다 ShowMessage()로 변수들의 값을 일일이 점검하는 개발자도 있지만, 어쨌든 코드를 더 추가하면 할수록 프로세스의 메모리 사용 구조가 변하기 때문에 P는 이전과는 다른 데이터를 덮어쓰게 되므로 애플리케이션은 또 다른 이상한 행동을 하게 될 수도 있다. 결국, 포인터를 사용할 때에는 애초에 신경을 써서 버그가 없도록 코드를 작성하는 것이 최고라는 점을 명심하기 바란다. 포인터를 사용해서 메모리를 할당해야 한다면 코드를 두세 번씩 반복해서 점검하기 바란다.

초기화되지 않은 PChar를 사용하는 경우

PChar는 문자열형이 아니라 문자열을 가리키는 포인터란 점을 명심하기 바란다. PChar에 메모리를 할당할 때에는 StrAlloc()나 GetMem(), StrNew(), GlobalAlloc(), VirtulAlloc() 등의 함수를 사용하고, 해제할 때에는 FreeMem(), StrDispose(), GlobalFree(), VirtualFree() 등의 함수를 사용한다.



문자열이 필요하다면 포인터인 PChar 대신 가능한 한 String을 사용하는 것이 좋다. String은 메모리 할당과 해제가 자동으로 일어나므로 개발자가 신경쓸 필요가 없으며, PChar로도 쉽게 형 변환할 수 있다.
특히, 델파이 1.0의 애플리케이션을 델파이 5로 이식할 때 String형의 위력을 실감할 수 있을 것이다. 델파이 1.0에서 PChar는 필요악 같은 존재였다. 사실, 32비트 델파이에서는 PChar를 사용할 일이 거의 없다. 델파이 1의 애플리케이션을 델파이 5로 이식할 때에는 PChar들을 최대한 모두 String으로 변환하는 것이 좋을 것이다.

nil 포인터의 참조 제거

포인터 버그에는 nil(0에 해당하는 값) 포인터를 참조 제거하는 것도 포함된다. nil 포인터의 참조 제거(dereferencing)란, 아무 것도 가리키지 않는 포인터에 뭔가를 쓰려고 하는 것을 말한다. nil 포인터를 참조 제거하면 운영체제는 항상 접근 위반 오류를 일으킨다. 물론, 아무 것도 가리키지 않는 곳에 뭔가를 쓰는 것이기 때문에 메모리가 깨지거나 하지는 않으므로 그냥 예외 처리를 해주는 것만으로 문제를 해결할 수 있다. 다음은 nil 포인터의 참조 제거와 예외 처리를 보여주는 코드이다.

```
procedure I_AV;
var
  P: PByte;
begin
  P := Nil;
  try
      P^ := 1;
  except
      on EAccessViolation do
      MessageDlg('접근 에러', mtError, [mbOk], 0);
  end;
end;
```

어떤 프로그램이 이 프로시저를 호출하면 예외를 알리는 메시지 상자가 나타나긴 하지만, 프로그램은 여전히 제대로 작동하게 된다.

내장 디버거 사용법

델파이 통합 개발 환경에는 다양한 기능을 가진 디버거도 내장되어 있다. 내장 디버거의 기능들은 대부분 Run 메뉴를 통해서 사용한다. 내장 디버거의 기능에는 애플리케이션의 명령행 인자 지정, 정지점(breakpoint) 설정, 추적(trace)/단계적(step) 실행, 감시점(watch) 보기, 데이터의 평가(evaluate)와 수정 호출, 지역 변수 보기, 스택 정보 보기 등등 전문적인 디버거가 가지고 있는 기능들이 대부분 포함되어 있다.

명령행 인자 설정

명령행 인자를 이용하는 애플리케이션이라면 Run | Parameters 대화상자로 명령행 인자를 시험해 볼 수 있다. 시험해 볼 명령행 인자를 대화상자에 입력하면 된다.

정지점

정지점(breakpoint)이란, 특정 조건을 만족했을 때 프로그램의 실행을 일시 정지시키는 지점을 말한다. 가장 많이 사용되는 정지점은 소스 정지점이다. 소스 코드의 어떤 행을 소스 정지점으로 설정하면 프로그램이 실행되다가 그 코드를 만나면 실행이 일시 정지된다. 코드 편집기(Code Editor) 창의 제일 왼쪽 부분을 클릭하면 클릭한 위치의 코드 행이 소스 정지점이 된다. 마우스 오른쪽 버튼 클릭으로 띄운 팝업 메뉴나 메인 메뉴 Run에서 Add Breakpoint를 선택해서 소스 정지점을 설정할 수도 있다. 어쨌든, 소스 정지점은 의심스러운 함수나 프로시저의 행동을 점검할 때 아주 유용하다. 그림 19.2는 코드의 한 행을 소스 정지점으로 설정한 예이다.

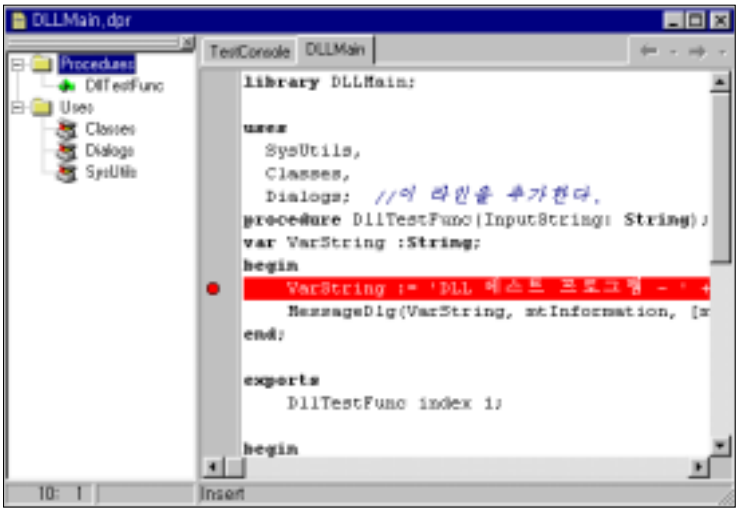


그림 19.2
코드 편집기에서 소스 중단점을 설정한 모습

조건부 정지점

프로그램의 실행이 소스 정지점에 왔을 때 매번 프로그램의 실행이 정지되는 것이 불편할 수도 있다. 그래서 델파이 디버거에서는 프로그램의 실행이 소스 정지점을 만나더라도, 특정한 조건을 만족하는 경우에만 실행을 정지하도록 조건을 설정할 수 있다. 그런 것을 조건부 정지점이라고 한다. 예를 들어서, 어떤 루프 안에 조건부 정지점을 적절히 설정하면 루프 횟수가 100번째일 때만 프로그램의 실행을 멈추도록 할 수도 있는 것이다. 그렇게 하면 <F9>키(Run)를 백 번 누를 필요가 없다. 조건부 정지점을 설명하기 위해서 다음과 같이 폼의 어떤 버튼의 클릭 이벤트 처리 메소드를 작성해 보았다고 가정하자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 100 do
  begin
      Caption := IntToStr(I); // 폼의 제목을 갱신
      Button1.Caption := IntToStr(I); // 버튼의 제목을 갱신
      Application.ProcessMessages; // 다른 메시지들을 처리한다.
  end;
end;
```

그리고 다음 행에 소스 정지점을 설정한다.

```
Caption := IntToStr(I); // 폼의 제목을 갱신
```

정지점을 설정한 다음에 View | Debug Windows | Breakpoints를 선택하면 Breakpoint List 대화상자가 나타난다. 목록에는 방금 설정한 정지점도 포함되어 있다. 정지점을 마우스 오른쪽 버튼으로 클릭한 다음, Properties를 선택하면 그림 19.3과 같은 Breakpoint Properties 대화상자가 나타난다. Condition 필드에 I = 50을 입력하고 OK를 클릭하면 이후부터는 I가 50일 때만 프로그램의 실행이 정지된다.

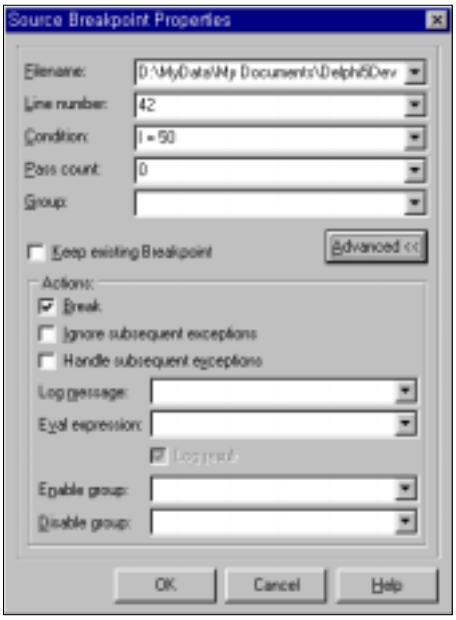


그림 19.3
Source Breakpoint Properties 대화상자

데이터 정지점

데이터 정지점(data breakpoint)은 메모리의 특정 주소의 값이 변할 때 프로그램을 정지시키는 정지점이다. 포인터를 다루거나 기타 세밀한 디버깅 작업이 필요할 때 사용하면 된다. 데이터 정지점을 설정하려면 실행 디버그 모드에서 메인 메뉴 중 Run | Add Breakpoint | Data Breakpoint를 선택하여 Add Data Breakpoint 대화상자를 띄워야 한다(그림 19.4). 이 대화상자에 감시하고자 하는 메모리 영역의 주소와 길이를 입력한다. 메모리 영역의 길이는 감시하고자 하는 데이터형에 맞게 지정하면 된다. 예를 들어서, Char이면 길이를 1바이트로 하고, Integer이면 4바이트 등이다. 또한, 소스 정지점에서와 같이 프로그램이 정지할 조건도 설정할 수 있다. 예를 들어서, 메모리 영역이 10번 변한 시점에서 프로그램을 정지시킬 수도 있다.

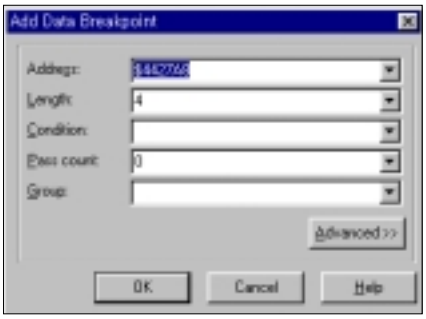


그림 19.4
Add Data Breakpoint 대화상자

주소 정지점

주소 정지점(address breakpoint)이란, 코드를 담고 있는 특정 주소가 실행되면 프로그램이 정지하도록 하는 정지점이다. 다른 사람이 만든 유닛(dcu)의 소스를 구하지 못한다면 소스 정지점을 사용할 수 없으므로 이러한 주소 정지점을 이용해서 프로그램을 디버깅해야 한다. 보통, CPU 뷰의 지역 메뉴에서 주소 정지점을 설정한다. 주소 정지점 역시 다른 정지점과 마찬가지로, 조건을 지정할 수 있다.

모듈 로드 정지점

모듈 로드 정지점(Module Load Breakpoint)이란, 말 그대로 특정 모듈이 적재되면 프로그램이 정지하도록 하는 정지점이다. 애플리케이션이 어떤 DLL이나 패키지를 적재하면 프로그램을 정지시키고 싶을 때 사용한다. 보통, Modules 창의 지역 메뉴에서 모듈 로드 정지점을 설정하나, 메인 메뉴의 Run | Add Breakpoint 항목으로도 모듈 로드 정지점을 설정할 수 있다.

코드를 한 줄씩 실행시키려면

코드를 한 줄씩 실행시킬 때에는 Run | Step Over나 Run | Trace Into 메뉴를 사용한다(델파이 IDE의 기본적인 키 설정의 경우, 각각 <F8>, <F7>키에 해당함). Trace Into의 경우, 프로시저나 함수가 호출되면 호출된 프로시저나 함수의 코드 안으로 들어가는 반면, Step Over는 프로시저나 함수를 그냥 하나의 실행 단위로 취급하고 넘어간다. 일반적으로, 정지점을 이용해서 일단 어떤 지점에서 프로그램을 정지시킨 다음에 이 두 기능을 사용한다. <F7>키와 <F8>키에 익숙해지기 바란다. 프로젝트의 규모가 커지다 보면 이 두 키가 큰 도움이 될 것이다.

또한, 코드 편집기 창에서 현재 커서가 있는 지점까지만 프로그램을 실행시키려면 Run | Run to Cursor <F4>키를 사용한다. 코드에 루프가 있는 경우, <F7>키나 <F8>키를 여러 번 눌러야 하므로 그럴 때에는 <F4>키를 이용해서 루프 다음 줄까지 실행시킨 후 <F7>키나 <F8>키를 사용하면 된다. 그리고 프로그램이 실행중이라도 코드 편집기 안에서 정지점을 설정할 수 있다는 점을 기억하기 바란다. 정지점을 설정하기 위해서 굳이 프로그램을 종료할 필요가 없는 것이다.

[Run | Program Pause] 메뉴를 선택하면 프로그램이 일시 정지되므로 정지점을 설정하기 편하다. 그리고 프로그램이 무한루프에 빠졌는지 의심스러울 때 프로그램을 일시 정지시키는 것이 도움이 된다. 그런데 실제로 기계어 코드를 생성해 내는 행에서만 프로그램을 정지시킬 수 있다는 점을 주의하기 바란다.

아예 프로그램의 종료를 강제적으로 수행하고자 할 경우, Run | Program Reset 메뉴를 선택하면 된다.



애플리케이션을 디버깅할 때 코드 편집기 창의 왼쪽 줄(gutter)을 보면 조그만 파란 점들이 있을 것이다. 파란 점은 그 줄의 코드가 실제로 기계어 코드로 컴파일되었다는 뜻이다. 파란 점이 없는 줄은 실제로 실행되는 기계어 코드가 없으므로 정지점을 설정할 수 없다. 이전 버전의 델파이와는 달리, 델파이 5는 애플리케이션이 컴파일된 후 실행중이어야만 파란 점들을 표시한다.

로컬 변수

[View | Debug Windows | Local Variables]는 일일이 Watch List 창에 변수나 속성의 값을 관찰할 필요 없이 현재 디버깅 라인에 해당되는 범위의 로컬 변수를 자동적으로 보여주는 옵션이다.

Watch List 사용법

Watch란, 어떤 변수나 속성의 값을 감시/추적할 때 사용하는 것이다. Watch List 창에 변수나 속성의 값이 제대로 나타나려면 프로그램이 일시 정지해야 한다(정지점 등을

이용). Watch를 추가할 때에는 Add Watch 대화상자에 유효한 오브젝트 파스칼 표현식이나 레지스터 이름을 입력하면 된다. 그림 19.5가 그 예이다.



그림 19.5
Watch List 창의 모습

디버그 조사기

디버그 조사기(Debug Inspector)라는 것을 이용하면 Watch 창으로 할 수 있는 것들보다 좀더 다양한 일들을 더욱 쉽게 할 수 있다. Run | Inspector 메뉴를 선택하면 간단한 대화상자가 나오는데, 여기에 오브젝트 파스칼 표현식을 입력하고 OK를 클릭하면 그 표현식에 대한 디버그 조사기가 나타난다. 그림 19.6은 폼 하나만으로 구성된 델파이 애플리케이션의 메인 폼에 대한 디버그 조사기이다.

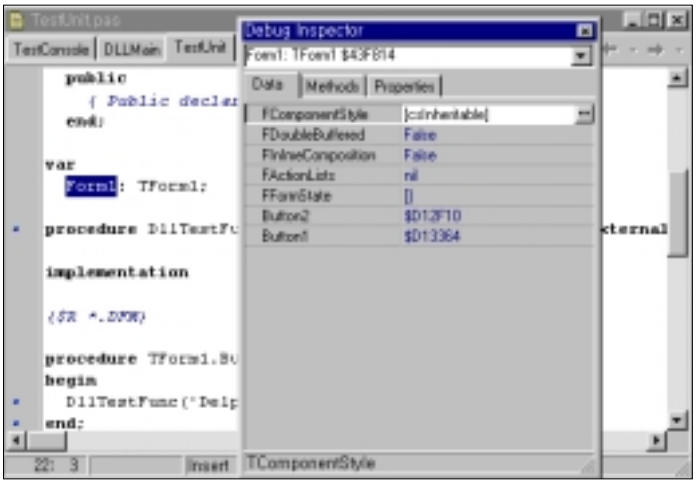


그림 19.6
디버그 조사기로 폼의 상태를 조사하는 모습

디버그 조사기로는 클래스나 레코드 등 여러 가지 개별 요소들로 이루어진 데이터를 편리하게 조사할 수 있다. 객체 조사기(Object Inspector)를 사용하는 것처럼 각 필드의 오른쪽 컬럼에 직접 값을 입력하거나, 드롭다운 목록에서 원하는 값을 선택해서 값을 변경할 수도 있다.

Evaluate/Modify 사용법

[Run | Evaluate/Modify] 메뉴를 사용하면 프로그램의 실행중에 레코드나 배열 등을 포함한 변수들이나 컴포넌트의 속성들의 값을 보거나 변경할 수 있으며, 변수나 속성이 포함된 표현식의 값도 평가(evaluate)할 수 있다. 그러나 현재 프로그램이 실행되는 곳의 범위를 벗어나는 변수나 함수의 값을 평가하거나 수정할 수는 없다는 점을 기억하기 바란다.



변수를 평가하거나 수정하는 것은 매우 편리한 기능이긴 하지만, 그만큼 위험이 따르기도 한다. 변수의 값을 잘못 수정하면 프로그램의 실행에 악영향을 미칠 수도 있기 때문이다.

호출 스택 보기

메인 메뉴의 View | Debug Windows | Call Stack을 선택하면 호출 스택의 내용을 볼 수 있다. Call Stack 창에는 아직 리턴되지 않은 함수와 프로시저 이름들과 각 인자들의 값이 나타나므로 복잡한 프로시저의 흐름을 추적할 때 매우 유용하다. 그림 19.7은 Call Stack 창의 모습이다.

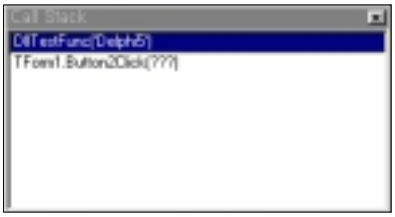


그림 19.7
Call Stack

Tip

Call Stack 창에 나온 함수나 프로시저를 마우스 오른쪽 버튼으로 클릭하면 그 함수나 프로시저의 코드로 쉽게 이동할 수 있다. 코드 편집기 창을 일일이 뒤지는 것보다 훨씬 편하다.

스레드 상태 보기

메인 메뉴의 View | Debug Windows | Thread를 선택하면 Thread Status 창이 나타나는데, 여기에는 애플리케이션에 속한 스레드들의 상태가 나타난다. 애플리케이션을 일시 정지시키면(메뉴나 정지점을 통해서) Thread Status 창의 팝업 메뉴를 통해서 현재 스레드를 변경하거나 스레드에 관련된 소스 코드를 볼 수 있다. 현재 스레드를 변경하는 경우, 그 이후의 재실행(Run이나 Step Over 등)은 그 스레드에 기반해서 작동한다는 점을 기억하기 바란다. 그림 19.8은 Thread Status 창의 모습이다.



그림 19.8
Thread Status 창

이벤트 기록

델파이 5에는 여러 가지 이벤트 발생을 기록하는 기능이 도입되었다. 그림 19.9는 View | Debug Windows | Event Log를 선택했을 때 나타나는 창이다. 이 Event Log 창의 설정은 팝업 메뉴나 Tools | Debugger Options 대화상자의 Event Log 페이지에서 변경하면 된다.

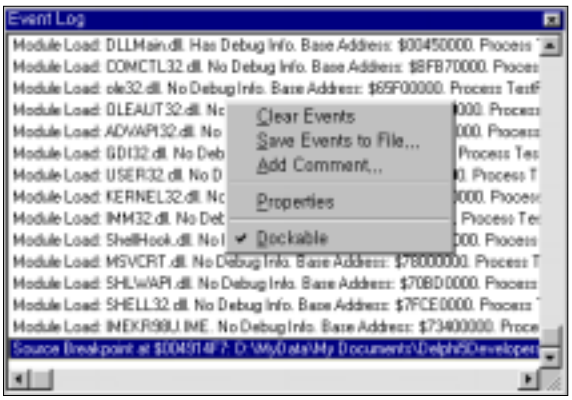


그림 19.9
Event Log 창

이 창은 일반적인 이벤트뿐만 아니라, 프로세스 시작이나 프로세스 중지, 모듈 로드 정지점, 애플리케이션이 받은 윈도우즈 메시지, 그리고 OutputDebugString()을 사용한 애플리케이션의 출력 내용 등 프로세스 정보들도 기록한다. 이러한 로그 정보를 팝업 메뉴를 통해 파일로 저장할 수도 있다.



OutputDebugString() API 함수는 애플리케이션의 디버깅에 매우 유용한 함수이다. 이 함수의 유일한 인자는 PChar이다. 이 인자로 지정한 문자열은 디버거에게 넘겨지는데, 델파이의 경우 내장 디버거가 그 문자열을 Event Log 창에 추가한다. 이 함수를 사용하면 Watch나 정지점을 쓰지 않고도 애플리케이션의 내부 상태를 쉽게 확인할 수 있다.

모듈 상태 보기

그림 19.10은 Modules 뷰이다. 이 뷰에는 디버깅중인 애플리케이션이 로드한 모듈들(EXE, DLL, DPL)에 대한 정보가 나타난다. 이 뷰에서 여러 가지 모듈 관련 정보를 확인할 수도 있고, 또 앞에서 말한 모듈 로드 정지점을 설정할 수도 있다.

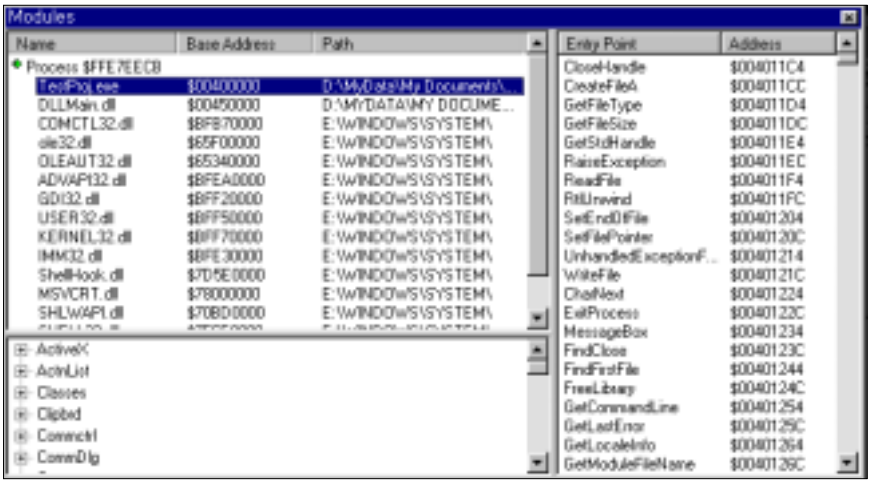


그림 19.10
Modules 뷰 창

DLL의 디버깅

델파이 내장 디버거를 이용하면 임의의 애플리케이션을 호스트로 사용해서 DLL 프로젝트를 디버깅할 수 있다. DLL 프로젝트를 열고, Run | Parameters를 선택한 다음 Run Parameters 대화상자에서 호스트 애플리케이션을 지정하면 된다(그림 19.11).

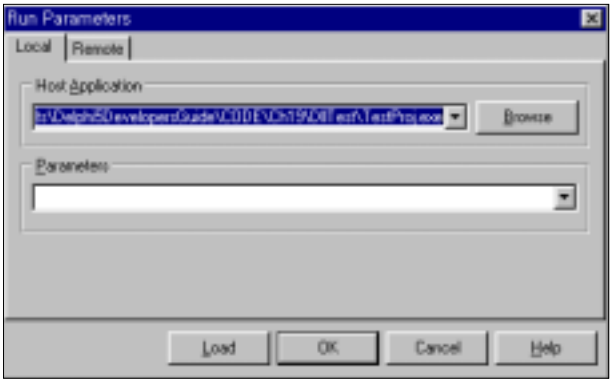


그림 19.11
호스트 응용 프로그램을 지정

물론, 지정한 호스트 애플리케이션은 디버깅하고자 하는 DLL을 사용하는 실행 파일이어야 한다. 일단 호스트 애플리케이션을 적절히 지정한 다음부터는 일반 애플리케이션을 디버깅할 때와 마찬가지 방법으로(정지점, Step, Trace 등) DLL을 디버깅하면 된다.

다른 프로세스의 문맥하에서만 실행되는 DLL이나 ActiveX 컨트롤, 또는 프로세스 내부(in-process) COM 서버들은 모두 이런 식으로 디버깅해야 한다.

CPU 뷰와 FPU 뷰

델파이 5의 내장 디버거는 CPU 뷰도 공식적으로 지원한다. CPU 뷰는 다섯 개의 창으로 이루어져 있다. CPU 창, 메모리 덤프 창, 레지스터 창, 플래그 창, 그리고 스택 창이다(그림 19.12). 이 창들을 통해서 저수준의 아주 세밀한 디버깅에 적합한 정보들을 얻을 수 있다.

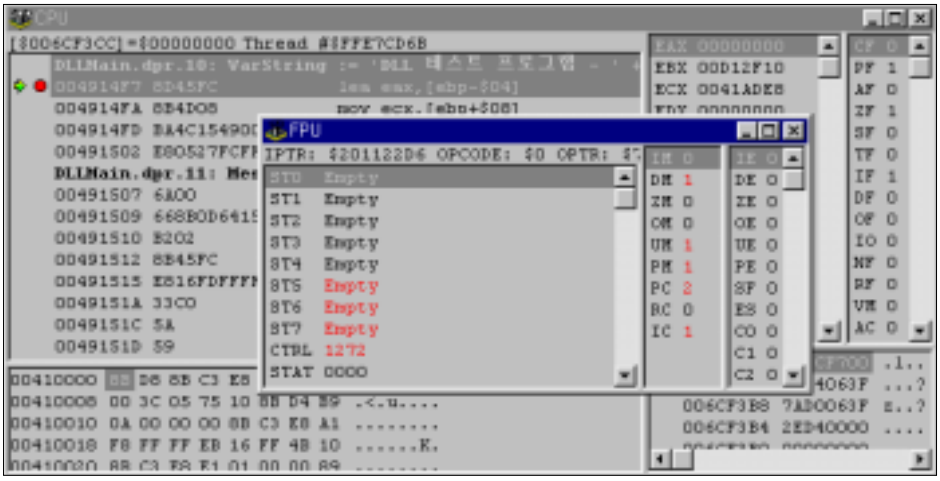


그림 19.12
CPU 뷰와 FPU 뷰

CPU 창에는 실행중인 프로세스의 기계어 코드와 니모닉(어셈블리) 코드들이 나타난다. 이 창의 내용을 이해하려면 어셈블리 코드에 대해 잘 알고 있어야 한다. 어셈블리 언어를 잘 아는 개발자라면 이 창을 통해서 버그를 잡아낼 수도 있고, 델파이 컴파일러가 얼마나 효율적인 코드를 생성해내는지도 알 수 있을 것이지만, 어셈블리 언어에 대해 익숙하지 않은 개발자라면 별로 얻을 것이 없을 것이다.

CPU 뷰의 팝업 메뉴를 이용하면 창의 표시 방법을 바꾸거나 다른 주소를 볼 수도 있으며, 현재 명령 포인터(EIP) 위치로 되돌아가거나 코드를 검색할 수도 있고, 기계어 코드에 대응하는 소스 코드를 볼 수도 있다. 또한, 조사하고자 하는 스레드를 선택할 수도 있다. 그리고 운영체제의 예외 목록도 제공한다.

메모리 덤프 창에는 메모리의 내용이 그대로 나타난다. 메모리 덤프 창의 표시 방식은 Byte, Word, DWORD, Single, Double, Extended 중 하나로 변경할 수 있다. 또한, 특정 바이트들을 검색하거나 메모리 내용을 수정할 수도 있으며, 메모리에 관련된 코드나 데이터 포인터를 추적할 수도 있다.

레지스터와 플래그 창은 말 그대로 CPU의 레지스터와 플래그들의 내용이 나타나는 곳이다. 레지스터나 플래그의 값을 직접 수정할 수도 있다.

스택 창에는 프로세스의 스택 내용이 나타난다. 값을 변경할 수 있으며, 주소를 추적할 수도 있다.

델파이 5에서 추가된 디버깅 옵션이 FPU 뷰라는 것이 있다. 이것은 인텔의 MMX상의 CPU에서 플로팅 소수점 유닛의 내용을 볼 수 있게끔 해주는 추가 옵션이다.

다중 프로세스/리모트 디버깅

델파이 5에서 추가된 디버깅 중 가장 주목할 만한 것들은 다중 프로세스 디버깅과 리모트 디버깅이 가능하다는 점이다.

다중 프로세스 디버깅

통합 디버거는 NT상에서 다중 프로세스 디버깅을 지원한다. NT가 아닌 경우, 즉 95/98인 경우에는 다중 프로세스 디버깅을 지원하지 않는다. 그러나 각각의 프로젝트들을 독립적으로 컴파일하고 실행시킬 수는 있다. 이렇게 하기 위해서는 Tools | Debugger Options에서 하단부의 Integrated Debugging 옵션을 제거하면 된다. NT에서 프로세스를 디버깅하기 위해서는 다음의 여러 가지 방법 중에서 하나를 선택할 수 있다. 다중 프로세스를 디버깅하는 동안에는 컴파일할 수 없기 때문에 디버깅을 시작하기 전에 반드시 Project | Build All로 모두 컴파일해야 한다.

현재의 프로세스를 설정하고 변경해 줄 수 있는 스레드 상태 윈도우를 보여주기 위해 View | Debug Windows | Threads를 사용한다. 초록색 화살표는 현재의 프로세스를 보여준다. 빨간색 화살표는 현재의 프로세스가 아닌 것을 보여준다. 새로운 프로세스 디버깅을 위해서는 Run | Parameters를 사용한다. 이미 실행중인 프로세스를 첨가하기 위해서는 Run | Attach to Process를 사용한다.

다중 프로세스 디버깅은 리모트 디버깅을 포함한다. 예를 들면, IDE는 컴퓨터 A의 프로세스, 컴퓨터 B의 프로세스, 로컬 컴퓨터 프로세스 등 총 3개를 디버그할 수 있다.

리모트 디버깅

델파이 5는 디버그 기호로 빌드된 EXE, DLL, 패키지의 리모트 디버깅을 지원한다. 리모트 디버그 서버는 서비스로서 Windows 95/98상에서는 지원되지 않는다. 다만, 실행 파일 모드로는 제공된다. Windows NT에서는 서비스나 프로그램으로서 리모트 디버깅 서비스가 실행될 수 있다. 그러나 Windows 95/98에서는 리모트 디버깅은 프로그램으로서 실행될 수 있다.

프로그램으로 실행하기 위해서는 도스 모드 박스나 단축키로 “Bordbg50.exe -listen”이라고 입력한다. 만약 NT 서비스로 설치했다면 디버거와 리모트 애플리케이션 사이의 통신이 자동적으로 제어된다.

IDE에서 로컬 머신상에서 리모트 디버깅 세션을 시작하기 위해서는 다음의 옵션들을 설정해야만 한다.

디버깅 세션에 대한 유저 인터페이스를 제공하는 “클라이언트”인 IDE(이것은 BCB.exe)와 디버그할 머신상에는 디버그 서버(bordbg50.exe)로 구성된다. 서버의 기능은 네트웍 연결을 통해 디버깅을 IDE와 같이 작동되는 것이다. 디버그 서버는 반드시 디버그 커널 dll(bordbk50.exe)과 평가기 dll(dcc50.dll)을 접속할 수 있어야 한다. 정품 CD의 RDEBUG 디렉토리로부터 setup.exe를 실행시키면 모든 것이 자동으로 설치된다.

리모트 머신에 연결시키기 위해서는 로컬과 리모트 머신은 반드시 TCP/IP로 연결되어 있어야만 한다. 리모트 디버깅 통신은 TCP 소킷을 사용하고, DNS에 의존한다. 이것은 로컬 머신이 반드시 리모트 머신에 대한 IP 어드레스를 얻을 수 있어야만 한다. 커맨드 라인에서 “nslookup name”은 어떤 특정한 이름에 바운드될 수 있는 IP를 찾아준다. DNS와 마이크로소프트 네트워킹 이름은 이와는 다르다. 양쪽 다 제어 패널에 있는 네트워킹 애플릿으로부터 얻을 수 있다.

클라이언트 IDE와 디버그 서버는 현재는 기본적으로 port 8000을 연결 포트로 사용한다.

그러면 클라이언트에서 리모트 머신상의 프로젝트를 어떻게 디버그하는지 살펴보자.

1. 제일 먼저 리모트 머신(다른 컴퓨터)상에서 커맨드 라인 모드에서 다음과 같이 입력해 리모트 디버깅 프로그램을 구동시킨다. 이 프로그램은 Delphi5\bin\ 디렉토리에 있다.

```
prompt> bordbg50 -listen
```



BORDBG50.EXE는 3가지 커맨드 라인 옵션이 있다.
-listen (연결에 대해 서버가 듣는다; 이것은 서비스 모드는 아니다.)
-install (서비스로서 설치한다. NT에서만 가능하다.)
-remove (서비스를 제거한다. 필요하다면 멈춘다. 이것도 NT에서 가능하다.)

2. 현재 자신의 컴퓨터(로컬 컴퓨터)상에서 델파이를 구동시키고 빈 폼을 하나 만든다. 그 폼 위에 단순히 버튼 하나만 올려 놓고 그 이벤트를 다음과 같이 작성한다.

```
procedure TForm.Button1Click(Sender: TObject);
begin
  Caption := '테스트 스트링';
end;
```

3. 현재의 컴퓨터(로컬 컴퓨터)에서 델파이의 Project | Options의 Directory/Conditional 페이지에서 Output의 디렉토리들을 리모터 컴퓨터(다른 컴퓨터)의 물리적 연결 네트웍 드라이브로 정한다. 예를 들면, H:\와 같이 정할 수 있다.

4. 현재의 컴퓨터(로컬 컴퓨터)에서 델파이의 Run | Parameters의 Remote 페이지에서 path를 정한다. 이때 경로명은 로컬 컴퓨터의 실제 위치를 정하고, 디렉토리도 명시한다. 예를 들면, C:\temp\project1.exe와 같이 정한다(3번에서 H:\라는 논리적 드라이브를 정했다 해서 H:\temp\project1.exe라고 정하면 안 된다).

5. Remote Host를 리모트 서버나 IP로 설정한다. 예를 들면, xxx.xxx.xxx.xxx라고 정한다.

6. 리모트 머신상에서 Debug project를 검사하기 위해 반드시 Debug project on remote machine을 선택하고 OK를 클릭한다. 디버그 정보를 추가하는 것은 프로그램의 파일 크기를 증가시킨다. 그러나 프로그램의 디버그를 모두 마치고 테스트가 끝났다면 배포를 위해 디버그 정보 옵션을 제거하고, 다시 실행 파일을 만들 수도 있다.

7. 프로그램에서 Caption := '테스트 스트링'; 부분에 정지점을 정하고 프로그램을 실행(Run)시키면 원격 컴퓨터상에서 프로그램이 활성화되어 나타나게 된다. 그리고 그 프로그램에서 버튼을 클릭하면 현재 로컬 컴퓨터상에서는 정지점에 멈추게 되고, 디버그 상태로 프로그램을 관찰할 수 있게 되는 것이다.

요약

이번 장에서는 디버깅 작업에 대해 이야기했다. 애플리케이션을 개발할 때 자주 만나게 되는 일반적인 버그에 대해서 이야기했으며, 내장 디버거와 델파이 5에서 추가된 다중 프로세스/리모트 디버깅에 대해서도 설명했다. 애플리케이션의 개발에 있어서 디버깅은 코드를 작성하는 것만큼이나 중요한 작업임을 명심해야 한다. 지금부터라도 디버거를 자유자재로 다룰 수 있도록 시간을 투자하기 바란다.

이것으로 2부가 끝났다. 3부에서는 COM과 VCL 컴포넌트를 중심으로 한 컴포넌트 기반의 개발에 대해 다룬다.

쉬어가는 페이지