CHAPTER _5

프로시저와 함수

이 장에서는 프로시저와 함수를 선언하고 호출하는 방법, 그리고 오버로드와호출 규칙, 파라미터 등에 대해서도 살펴봅니다.

■프로시저와 함수의 선언 ■파라미터

■ 프로시저 및 함수의 호출

흔히 합쳐서 루틴이라고 부르는 프로시저와 함수는, 프로그램의 다른 위치에서 호출할 수 있는 독립적인 문장 블럭입니다. 함수는 실행되었을 때 값을 리턴하는 루틴입니다. 프로시저는 값을 리턴하지 않는 루틴입니다.

함수 호출은 값을 리턴하므로 대입문과 연산에서 표현식으로 사용할 수 있습니다. 예를 들면,

```
I := SomeFunction(X);
```

위 코드는 SomeFunction을 호출하고 결과를 I에 대입합니다. 함수 호출은 대입문의 왼쪽에 나타날 수 없습니다.

프로시저 호출과 확장 문법 옵션이 활성화 ({$X+})되었을 때의 함수 호출은 완전한 문장으로 사용될 수 있습니다. 예를 들면,

```
DoSomething;
```

위 코드는 DoSomething 루틴을 호출합니다. DoSomething이 함수라면 리턴값은 버려집니다. 프로시저 및 함수를 재귀적으로 호출할 수 있습니다.

프로시저와 함수의 선언

프로시저나 함수를 선언할 때는 그 이름, 파라미터의 수와 타입을 지정하며, 함수의 경우에는 리턴값의 타입을 지정합니다. 선언의 이 부분을 프로토타입, 헤딩, 헤더라고 합니다. 다음으로 프로시저나 함수가 호출될 때마다 실행될 코드 블럭을 작성합니다. 선언의 이 부분을 루틴의 바디 (body) 또는 블럭이라고 합니다.

프로시저 선언

프로시저 선언의 구조는 다음과 같습니다.

```
procedure procedureName(parameterList); directives;
localDeclarations;
begin
  statements
end;
```

여기서, *procedureName*은 유효한 식별자이고, *statements*는 프로시저가 호출될 때 실행되는 일련의 문장이며, *parameterList*, *directives*, *localDeclarations*는 옵션입니다.
다음은 프로시저 선언의 예입니다.

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S:
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

이와 같이 선언되었을 경우, 다음과 같이 NumString 프로시저를 호출할 수 있습니다.

```
NumString(17, MyString);
```

이 프로시저 호출은 값 "17"을 MyString (string 변수여야 함)에 대입합니다.
프로시저의 문장 블럭에서 해당 프로시저의 *localDeclarations* 부분에서 선언된 변수와 다른 식별자들을 사용할 수 있습니다. 또 파라미터 리스트의 파라미터 이름을 사용할 수도 있습니다(위 예제의 N이나 S). 파라미터 리스트는 지역 변수 정의와 동일한 역할므로 *localDeclarations* 영역에서 파라미터 이름을 재선언하면 안됩니다. 마지막으로, 프로시저 선언이 속한 유효 범위(scope) 내의 모든 식별자를 사용할 수 있습니다.

함수 선언

함수 선언은 리턴 타입과 리턴 값을 지정하는 것을 제외하면 프로시저 선언과 비슷합니다. 함수 선언의 구조는 다음과 같습니다.

```
function functionName(parameterList): returnType; directives;
localDeclarations;
begin
  statements
end;
```

여기서, *functionName*은 유효한 식별자이고, *returnType*은 타입 식별자이며, *statements*는 함수가 호출될 때 실행되는 일련의 문장이며, *parameterList*, *directives*, *localDeclarations*는 옵션입니다.
함수의 문장 블럭에는 프로시저에 적용되는 것과 같은 규칙이 적용됩니다. 문장 블럭에서 함수의 *localDeclarations* 부분에서 선언된 변수와 다른 식별자, 파라미터 리스트의 파라미터 이름, 그리고 함수 선언이 속하는 유효 범위(scope) 내의 모든 식별자를 사용할 수 있습니다.
확장 문법이 활성화된 상태에서는({$X+}), 모든 함수에서 Result가 암시적으로 선언됩니다. Result를 다시 선언하려고 하지 마십시오. 예를 들면,

```
function WF: Integer;
begin
  WF := 17;
end;
```

위 코드는 파라미터가 없고 항상 정수값 17을 리턴하는 WF라는 상수 함수를 정의합니다. 이 선언은 다음과 동일합니다.

```
function WF: Integer;
begin
  Result := 17;
end;
```

다음과 같이 더 복잡한 함수 선언도 있습니다.

```
function Max(A: array of Real; N: Integer): Real;
var
  X: Real;
  I: Integer;
begin
  X := A[0];
  for I := 1 to N - 1 do
    if X < A[I] then X := A[I];
  Max := X;
end;
```

값의 타입이 선언된 리턴 타입과 일치하기만 하면, 문장 블럭 내에서 Result 혹은 함수 이름에 값을 여러 번 대입할 수 있습니다. 함수의 실행이 종료되면 Result 혹은 함수 이름에 마지막으로 대입된 값이 그 함수의 리턴값이 됩니다. 다음은 함수 내에서 리턴 값을 다루는 예입니다.

```
function Power(X: Real; Y:Integer): Real;
var
  I:Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;
```

Result와 함수 이름은 항상 같은 값입니다. 따라서,

```
function MyFunction: Integer;
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;
```

위 코드는 값 11을 리턴합니다. 그러나 Result가 완전히 함수 이름으로 대체 가능한 것은 아닙니다. 함수 이름이 대입문의 왼쪽에 나타나면 컴파일러는 함수 이름이 Result처럼 리턴 값을 기록하는 데 사용되고 있다고 이해합니다. 그러나 함수 이름이 문장 블럭의 다른 곳에서 나타나면 컴파일러는 함수 이름을 그 함수 자체의 재귀 호출로 해석합니다. 한편, Result는 연산, 타입 캐스트, 집합 생성자, 인덱스, 다른 루틴 호출 등에서 변수처럼 사용될 수 있습니다.

Result나 함수 이름에 값을 지정하지 않은 상태로 함수의 실행이 종료되면 그 함수의 리턴 값은 정의되지 않습니다.

호출 규칙

프로시저나 함수를 선언할 때, 지시어 register, pascal, cdecl, stdcall, safecall 중의 하나를 사용하여 호출 규칙(Calling convention)을 지정할 수 있습니다. 예를 들면, 다음과 같습니다.

```
function MyFunction(X, Y: Real): Real; cdecl;
```

호출 규칙은 파라미터를 루틴으로 전달하는 순서를 결정합니다. 또한 호출 규칙은 스택에서 파라미터 제거 방법, 파라미터 전달에서 레지스터 사용 여부, 에러와 예외 처리 등에 영향을 줍니다. 기본 호출 규칙은 register입니다.

- register와 pascal 규칙은 파라미터를 왼쪽에서 오른쪽으로 전달합니다. 즉, 가장 왼쪽에 있는 파라미터가 가장 먼저 계산, 전달되고 가장 오른쪽에 있는 파라미터는 마지막에 계산, 전달됩니다. cdecl, stdcall, safecall 규칙은 파라미터를 오른쪽에서 왼쪽으로 전달합니다.
- cdecl을 제외한 모든 규칙에서 리턴할 때 프로시저나 함수가 스택에서 파라미터를 제거합니다. cdecl 규칙에서는 리턴할 때 루틴 호출자가 스택에서 파라미터를 제거합니다.

- register 규칙에서는 파라미터를 전달하기 위해 CPU 레지스터를 최대 3개까지 사용하며, 다른 규칙들에서는 모든 파라미터를 스택을 통해 전달합니다.
- safecall 규칙은 예외 "방화벽"을 구현합니다. Windows에서 이 규칙은 프로세스간(interprocess) COM 에러 알림(notification)을 구현합니다.

아래의 표는 호출 규칙을 요약한 것입니다.

표 5.1 호출 규칙

지시어	파라미터 순서	클린 업	레지스터로 파라미터 전달?
register	왼쪽에서 오른쪽	루틴	예
pascal	왼쪽에서 오른쪽	루틴	아니오
cdecl	오른쪽에서 왼쪽	호출자	아니오
stdcall	오른쪽에서 왼쪽	루틴	아니오
safecall	오른쪽에서 왼쪽	루틴	아니오

기본 register 규칙은 보통 스택 프레임을 생성하지 않기 때문에 가장 효율적입니다. (published 속성에 대한 액세스 메소드는 반드시 register를 사용해야 합니다.) cdecl 규칙은 C 또는 C++ 언어로 개발된 DLL의 함수를 호출할 때 유용하며, stdcall 및 safecall 규칙은 일반적으로 외부 코드를 호출할 때 사용하는 것이 좋습니다.
Win32에서 운영체제 API는 stdcall 및 safecall 규칙을 사용합니다. 기타 운영체제는 일반적으로 cdecl 규칙을 사용합니다. (stdcall 규칙이 cdecl 규칙보다 효율적입니다.)
safecall 규칙은 듀얼 인터페이스 메소드(11장 "인터페이스" 참조)를 선언할 때 반드시 사용해야 합니다. pascal 규칙은 하위 호환성을 위해 유지됩니다. 호출 규칙에 대한 자세한 내용은 13장 "프로그램 제어"를 참조하십시오.
지시어 near, far, export는 16비트 Windows 프로그래밍에서의 호출을 위해 사용됩니다. Win32에서는 아무런 영향을 미치지 않고 오직 하위 호환성을 위해서만 유지됩니다.

forward 선언과 interface 선언

forward 지시어는 프로시저나 함수 선언에서 함수 블럭과 지역 변수 선언, 문장을 대신합니다. 예를 들면,

```
function Calculate(X, Y: Integer): Real; forward;
```

위 코드는 Calculate라는 함수를 선언합니다. 루틴은 forward 선언 다음 어딘가에서 실제 블럭을 포함한 정의적 선언(defining declaration)으로 다시 선언되어야 합니다. Calculate에 대한 정의적 선언은 다음과 같습니다.

```
function Calculate;
  { 선언들 }
begin
  { 문장 블럭 }
end;
```

일반적으로 정의적 선언에서는 루틴의 파라미터 리스트나 리턴 타입을 다시 명시하지 않아도 됩니다. 그러나 정의적 선언이 루틴 파라미터 리스트나 리턴 타입을 다시 표시하는 경우에는 forward 선언문에 있는 것과 정확하게 일치해야 합니다. forward 선언문이 오버로드된 프로시저 또는 함수("프로시저와 함수 오버로드" 참조)를 지정하면 정의적 선언은 반드시 파라미터 리스트를 반복해야 합니다.

forward 선언과 그 정의적 선언은 같은 타입 선언 영역에 있어야 합니다. 다시 말해, 새 영역 (var 영역이나 const 영역 등)을 forward 선언과 그 정의적 선언 사이에 추가할 수 없습니다. 정의적 선언은 external 또는 assembler 선언이 될 수 있지만, 또 다른 forward 선언문은 될 수 없습니다.

forward 선언의 목적은 프로시저 또는 함수 식별자의 유효 범위를 소스 코드에서 더 앞까지 확장하기 위한 것입니다. 이렇게 하면 forward 선언 루틴이 실제로 정의되기 전에 다른 프로시저 및 함수에서 이를 호출할 수 있습니다. 코드를 더 유연하게 해주는 것 외에, forward 선언문은 상호 재귀(mutual recursion)를 위해서도 필요합니다.

forward 지시어는 유닛의 interface 섹션에서는 아무 효과도 없습니다. interface 섹션의 프로시저 및 함수 헤더는 forward 선언처럼 동작하며 implementation 섹션에 그 정의적 선언이 있어야 합니다. interface 섹션에서 선언된 루틴은 유닛의 어디에서나 사용할 수 있으며, 선언된 유닛을 uses하는 다른 유닛이나 프로그램에서도 사용할 수 있습니다.

external 선언

external 지시어는 프로시저나 함수 선언의 블럭을 대체하며, 개발중인 프로그램과는 별도로 컴파일된 루틴을 호출할 수 있습니다. 외부 루틴은 오브젝트 파일(.obj)이나 DLL에서 호출할 수 있습니다.

파라미터의 수가 가변적인 C++ 함수를 가져오려면 varargs 지시어를 사용하면 됩니다.

예를 들면, 다음과 같습니다.

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

varargs 지시어는 external 루틴과 cdecl 호출 규칙에만 작동합니다.

■ obj 파일과의 링크

별도로 컴파일된 오브젝트 파일의 루틴을 호출하려면 먼저 $L (또는 $LINK) 컴파일러 지시어를 사용하여 오브젝트 파일을 애플리케이션과 연결하십시오. 예를 들면,

```
{$L BLOCK.OBJ}
```

위의 지시어를 사용하면 BLOCK.OBJ를 현재 프로그램이나 유닛으로 연결합니다. 다음으로, 호출하려는 함수와 프로시저를 선언합니다.

```
procedure MoveWord(var Source, Dest; Count: Integer) external;
procedure FillWord(var Dest; Data: Integer; Count: Integer) external;
```

이제 BLOCK.OBJ에서 MoveWord와 FillWord 루틴을 호출할 수 있습니다.
위와 같은 선언은 종종 어셈블리어로 작성된 외부 루틴을 액세스하는 데 사용됩니다. 어셈블리어 루틴을 직접 델파이 소스 코드에 사용할 수도 있습니다. 자세한 내용은 14장 "인라인 어셈블리 코드"를 참조하십시오.

■ DLL 함수의 임포트

DLL에서 루틴을 임포트하려면 다음과 같은 형식의 지시어를 일반적인 프로시저 또는 함수 헤더의 끝에 붙입니다.

```
external stringConstant;
```

여기서 *stringConstant*는 작은 따옴표로 둘러싼 DLL 파일의 이름입니다. 예를 들어,

```
function SomeFunction(S: string): string external 'strlib.dll';
```

위 코드는 strlib.dll에서 SomeFunction이라는 함수를 임포트합니다.
DLL에 있는 루틴 이름이 아닌 다른 이름으로 루틴을 가져올 수 있습니다. 이렇게 하려면 external 지시어 뒤에 원래의 이름을 지정하십시오.

```
external stringConstant1 name stringConstant2;
```

여기서, *stringConstant1*는 DLL 파일의 이름이고 *stringConstant2*는 루틴의 원래 이름입니다.
예를 들어, 다음 선언은 Windows API의 일부인 user32.dll 에서 함수를 임포트합니다.

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer;
stdcall; external 'user32.dll' name 'MessageBoxA';
```

함수의 원래 이름은 MessageBoxA이지만, MessageBox라는 이름으로 가져옵니다.
이름 대신 번호를 사용하여 임포트하려는 루틴을 식별할 수 있습니다.

```
external stringConstant index integerConstant;
```

여기서 *integerConstant*는 export 테이블에 있는 루틴의 인덱스입니다.
import 선언에서 루틴 이름의 철자와 대소문자가 정확하게 일치하도록 해야 합니다. 이런 선언 이후에 임포트한 루틴을 호출할 때에는 이름의 대소문자를 구분하지 않습니다.
DLL에 대한 자세한 내용은 10장 "DLL과 패키지"를 참조하십시오.

프로시저와 함수 오버로드

같은 유효 범위(scope) 내에서 둘 이상의 루틴을 같은 이름으로 선언할 수 있습니다. 이를 오버로드(overload)라고 합니다. 오버로드된 루틴은 overload 지시어로 선언해야 하여 파라미터 리스트로 구분합니다. 예를 들어, 다음의 선언들을 살펴봅시다.

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X;
end;

function Divide(X, Y: Integer): Integer; overload;
```

```
begin
  Result := X div Y;
end;
```

위 코드에서는 Divide라는 이름을 가진 함수를 두 개 만드는데, 두 함수는 파라미터 타입이 다릅니다. Divide를 호출하면 컴파일러는 호출에서 전달된 실제 파라미터들을 봐서 둘 중 어떤 함수를 호출할 지를 결정합니다. 예를 들어, Divide(6.0, 3.0)는 해당 인수가 실수이기 때문에 첫 번째 Divide 함수를 호출합니다.

오버로드된 루틴에 그 루틴 선언들의 파라미터들과 타입이 동일하지 않은 파라미터들을 전달할 수 있지만, 파라미터와 대입 호환이 가능한 파라미터들을 가진 루틴이 있어야 합니다. 이는 루틴이 여러 정수 타입이나 실수 타입으로 오버로드될 때 가장 자주 발생합니다.

```
procedure Store(X: Longint) overload;
procedure Store(X: Shortint) overload;
```

이 경우, 모호함 없이 가능하다면, 컴파일러는 호출 시 넘겨진 파라미터를 포함하면서 타입 범위가 가장 작은 파라미터가 있는 루틴을 호출합니다. (실수 타입의 상수 표현식은 항상 Extended 타입이라는 것을 기억하십시오.)

오버로드된 루틴은 파라미터의 수 또는 파라미터의 타입으로 구별될 수 있어야 합니다. 따라서 다음의 두 선언들은 컴파일 에러를 일으킵니다.

```
function Cap(S: string): string; overload;
  ...
procedure Cap(var Str: string); overload;
  ...
```

그러나 다음과 같은 선언은 적합합니다.

```
function Func(X: Real; Y: Integer): Real; overload;
  ...
function Func(X: Integer; Y: Real): Real; overload;
  ...
```

오버로드된 루틴들이 forward 선언 또는 interface 선언에서 선언되면 정의적 선언에서는

반드시 루틴의 파라미터 리스트를 지정해야 합니다.
컴파일러는 같은 파라미터 위치에 string(UnicodeString)을 포함하는 함수, AnsiString을 포함하는 함수, 그리고 WideString을 포함하는 함수가 오버로드되어 있을 때 이들을 구별할 수 있습니다. 이렇게 오버로드된 상황에서 전달된 문자열 상수나 리터럴은 네이티브 문자열 타입, 즉 string(UnicodeString)으로 해석됩니다. PChar(PWideChar)를 포함하는 함수와 PAnsiChar를 포함하는 함수가 오버로드되어 있을 경우에도 마찬가지로 네이티브 문자 타입인 PChar (PWideChar)으로 해석됩니다.

```
procedure test(const S: String);  overload;
procedure test(const S: AnsiString);  overload;
procedure test(const W: WideString); overload;
var
  a: string;
  b: AnsiString
  c: WideString
begin
  a := 'a';
  b := 'b';
  c := 'c';
  test(a);    // String 버전을 호출
  test(b);    // AnsiString 버전을 호출
  test(c);    // WideString 버전을 호출
  test('abc');    // String 버전을 호출
  test(AnsiString('abc'));   // AnsiString 버전을 호출
  test(WideString('abc'));   // WideString 버전을 호출
end;
```

variant 타입도 오버로드된 함수 선언들에서 파라미터로 사용될 수 있습니다. variant는 모든 단순 타입들보다 더 일반적인 것으로 간주됩니다. variant 타입보다는 정확하게 타입이 일치하는 함수가 항상 선택됩니다. variant 타입의 표현식이 이런 오버로드된 상황에서 파라미터로 전달되고, 또 그 파라미터 위치에 variant를 파라미터로 받는 오버로드된 함수가 존재할 경우, variant 타입으로 정확하게 일치하는 것으로 간주됩니다.
이것은 실수 타입들을 다룰 때 작은 부작용을 일으킬 수 있습니다. 실수 타입들은 크기로 매치됩니다. 오버로드 호출로 전달된 실수 변수에 해당하는 정확하게 일치하는 루틴이 없고, variant 파라미터를 가진 오버로드 함수가 있다면, 더 작은 실수 타입의 파라미터를 가진 루틴보다는 variant 파라미터를 가진 루틴이 선택됩니다.
다음 예를 살펴봅시다.

```
procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: variant); overload;
var
  v: variant;
begin
  foo(1);        // 정수 버전
  foo(v);        // variant 버전
  foo(1.2);      // variant 버전(실수 상수 -> extended 타입)
end;
```

이 예제에서 double 버전의 foo가 아니라 variant 버전의 foo가 호출됩니다. 이것은 상수 1.2는 암시적으로 extended 타입이고, extended 타입은 double 타입과 정확하게 일치하지 않기 때문입니다. extended는 variant와도 정확히 일치하지는 않지만, variant는 더 일반적인 타입으로 간주되기 때문입니다.(반면 double은 extended보다 작은 타입입니다.)

```
foo(Double(1.2));
```

이런 타입 캐스트는 가능하지 않습니다. 대신 타입 지정 상수(typed constant)를 사용하면 됩니다.

```
const d: double = 1.2;
begin
   foo(d);
end;
```

위의 코드는 정확하게 동작하며, double 버전을 호출합니다.

```
const s: single = 1.2;
begin
  foo(s);
end;
```

위의 코드도 역시 double 버전의 foo를 호출합니다. single은 variant보다 double에 더 잘 맞기 때문입니다.

오버로드된 루틴들을 선언할 때, 실수가 variant로 바뀌어버리는 문제를 피하는 가장 좋은

방법은 variant 버전과 함께 각각의 실수 타입마다(Single, Double, Extended) 오버로드된 함수를 선언하는 것입니다.
오버로드된 루틴에서 기본 파라미터(default parameter)를 사용한다면 파라미터 리스트가 모호해지지 않도록 주의하십시오. 자세한 내용은 "기본 파라미터와 오버로드된 루틴"을 참조하십시오.
호출할 때 루틴의 이름을 한정하여 오버로드의 잠재적인 효과를 제한할 수 있습니다. 예를 들어, Unit1.MyProcedure(X, Y)는 Unit1에서 선언된 루틴만 호출할 수 있습니다. Unit1에 호출의 이름 및 파라미터 리스트와 일치하는 루틴이 없으면 에러가 발생합니다.
클래스 계층 구조에서 오버로드된 메소드 구별에 대한 자세한 내용은 6장의 "메소드 오버로드"를 참조하십시오. DLL에서 오버로드된 루틴 엑스포트에 대한 자세한 내용은 10장의 "exports 절"을 참조하십시오.

지역 선언

함수나 프로시저의 바디 부분은 루틴의 문장 블럭에서 사용할 지역 변수들의 선언으로 시작하는 경우가 많습니다. 이런 선언에는 상수, 타입 및 기타 루틴도 포함합니다. 로컬 식별자의 유효 범위는 그 식별자가 선언된 루틴으로 제한됩니다.

■ 중첩된 루틴

함수 및 프로시저는 블럭의 지역 선언 영역에 다른 함수나 프로시저를 포함할 수 있습니다(nested routine). 예를 들어, 다음의 DoSomething 프로시저의 선언에는 중첩된 프로시저가 포함되어 있습니다.

```
procedure DoSomething(S: string);
var
  X, Y: Integer;
  procedure NestedProc(S: string);
  begin
    ...
  end
begin
  ...
  NestedProc(S);
  ...
end;
```

중첩 루틴의 유효 범위는 루틴이 선언된 프로시저 또는 함수로 제한됩니다. 위 예제에서

NestedProc은 DoSomething 내에서만 호출될 수 있습니다.
중첩 루틴의 실제 예제를 보려면 SysUtils 유닛에 있는 DateTimeToString 프로시저, ScanDate 함수 등을 살펴보십시오.

파라미터

대부분의 프로시저 및 함수 헤더에는 파라미터(parameter) 리스트가 있습니다. 예를 들어,

```
function Power(X: Real; Y: Integer): Real;
```

위의 헤더에서, 파라미터 리스트는 (X: Real; Y: Integer)입니다.
파라미터 리스트는 세미콜론으로 구분하고 괄호로 묶은 일련의 파라미터 선언입니다. 각 선언에서는 콤마로 구분된 파라미터 이름과 뒤이어 대부분 콜론과 타입 식별자가 나오며, 어떤 경우에는 = 기호와 기본값이 오기도 합니다. 파라미터 이름은 유효한 식별자여야 합니다. 선언 앞에는 예약어 var, const, out 중 하나가 있을 수 있습니다. 예를 들면, 다음과 같습니다.

```
(X, Y: Real):
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer}
```

파라미터 리스트는 루틴이 호출될 때 전달되어야 할 파라미터의 개수, 순서와 타입을 지정합니다. 루틴에 파라미터가 없다면 선언에서 식별자 리스트와 괄호를 생략하십시오.

```
procedure UpdateRecords;
begin
  ...
end;
```

프로시저 또는 함수의 바디 내에서, 파라미터 이름(위 첫 번째 예제의 X와 Y)은 지역 변수로 사용할 수 있습니다. 파라미터 이름을 프로시저나 함수 바디의 지역 선언 영역에서 다시 선언하지 마십시오.

파라미터 의미 구조

파라미터는 다음과 같은 방법으로 분류됩니다.

- 모든 파라미터는 값, 변수, 상수, 출력 파라미터로 분류합니다. 기본은 값 파라미터며, 예약어 var, const, out은 각각 변수, 상수, 출력 파라미터를 나타냅니다.
- 값 파라미터는 항상 타입이 지정되며, 상수, 변수, 출력 파라미터는 타입을 가질 수도 가지지 않을 수도 있습니다.
- 배열 파라미터에는 특수한 규칙이 적용됩니다. 5장의 "배열 파라미터"를 참조하십시오.

파일과 파일을 포함한 구조 타입의 인스턴스는 변수 파라미터(var)로만 전달할 수 있습니다.

■ 값 및 변수 파라미터

대부분의 파라미터는 값 파라미터(기본) 또는 변수(var) 파라미터입니다. 값파라미터는 값에 의해 전달되는 반면, 변수 파라미터는 참조에 의해 전달됩니다. 이것이 의미하는 것을 알아보기 위해 다음의 예제를 살펴봅시다.

```
function DoubleByValue(X: Integer): Integer; // X는 값 파라미터
begin
  X := X * 2;
  Result := X;
end;

function DoubleByRef(var X: Integer): Integer; // X는 변수 파라미터
begin
  X := X * 2;
  Result := X;
end;
```

이 함수들은 같은 결과를 리턴하지만, 두 번째 함수인 DoubleByRef만 파라미터로 전달된 변수의 값을 변경할 수 있습니다. 다음과 같이 이 함수들을 호출한다고 생각해봅시다.

```
var
  I, J, V, W: Integer;
begin
  I := 4;
  V := 4;
  J := DoubleByValue(I); // J = 8, I = 4
  W := DoubleByRef(V); // W = 8, V = 8
end;
```

코드 실행이 끝난 다음, DoubleByValue로 전달된 변수 I는 처음에 대입한 값과 같은 값을 가집니다. 그러나 DoubleByRef로 전달된 변수 V는 다른 값을 가지게 됩니다.
값 파라미터는 프로시저나 함수 호출에 의해 전달된 값으로 초기화된 지역 변수처럼 동작합니다. 변수를 값 파라미터로 전달하면 프로시저나 함수는 그 변수의 값을 복사합니다. 이 복사본을 변경하더라도 원래의 변수에는 아무런 영향을 주지 않으며, 프로그램 실행이 호출했던 부분으로 리턴된 후에는 변경되었던 내용을 잃어버립니다.
반면, 변수 파라미터는 복사본이 아니라 포인터처럼 동작합니다. 함수나 프로시저의 바디에서 파라미터를 변경하면, 프로그램 실행이 호출했던 부분으로 리턴되어 파라미터 이름 자체는 유효 범위를 벗어난 후에도 그 변경 내용은 계속 유지됩니다.
같은 변수를 둘 이상의 var 파라미터로 전달하더라도 복사본은 만들어지지 않습니다. 이는 아래의 예제에서 설명합니다.

```
procedure AddOne(var X, Y: Integer);
begin
  X := X + 1;
  Y := Y + 1;
end;

var I: Integer;
begin;
  I := 1;
  AddOne(I, I);
end;
```

이 코드가 실행된 후에 I의 값은 3이 됩니다.
루틴 선언에서 var 파라미터를 지정하면 루틴을 호출할 때 대입 가능한 표현식을 전달해야 합니다. 여기서 대입 가능한 표현식에는 변수, 타입 지정 상수({$J+} 상태), 역참조(dereference) 포인터, 인덱스된 변수가 해당됩니다. 이전 예제의 경우를 다시 예로 들면, DoubleByValue(7)는 적합하지만, DoubleByRef(7)는 에러를 일으킵니다.
DoubleByRef(MyArray[I])처럼 var 파라미터로 전달되는 인덱스 혹은 포인터 역참조는 루틴이 실행되기 전에 한번 계산됩니다.

■ 상수 파라미터

상수(const) 파라미터는 지역 상수 또는 읽기 전용 변수와 비슷합니다. 상수 파라미터는 프로시저나 함수의 바디에서 값을 대입할 수 없다는 것과 또다른 루틴에 값을 var 파라미터로 전달할 수 없다는 것만 제외하면, 상수 파라미터는 값 파라미터와 유사합니다. (그러나 객체

참조를 상수 파라미터로 전달하는 경우, 객체의 속성을 수정할 수는 있습니다.)
const를 사용하면 컴파일러는 구조 타입 파라미터와 문자열 타입 파라미터에 대해 코드를 최적화할 수 있습니다. 또한 의도하지 않게 파라미터를 참조로(var) 다른 루틴으로 전달하는 것도 막을 수 있습니다.
다음은 SysUtils 유닛에 있는 CompareStr 함수의 헤더입니다.

```
function CompareStr(const S1, S2: string): Integer;
```

S1 및 S2는 CompareStr의 바디에서 수정되지 않기 때문에 상수 파라미터로 선언될 수 있습니다.

■ 출력 파라미터

out 파라미터는 변수 파라미터처럼 참조에 의해 전달됩니다. 그러나 out 파라미터에서는 참조되는 변수의 초기 값은 루틴으로 전달할 때 버려집니다. out 파라미터는 출력용으로만 사용합니다. 즉, 함수나 프로시저에게 출력을 어디에 저장할 지는 알려주지만, 입력에 대한 정보는 알려주지 않습니다.
예를 들어, 다음과 같은 프로시저 헤더를 살펴봅시다.

```
prodedure GetInfo(out Info: SomeRecordType);
```

GetInfo를 호출하려면 반드시 SomeRecordType 타입 변수를 전달해야 합니다.

```
var MyRecord: SomeRecordType;
...
GetInfo(MyRecord);
```

그러나 GetInfo 프로시저로 데이터를 전달하는 데 MyRecord를 사용하지 않습니다. MyRecord는 GetInfo가 만들어낼 정보를 저장할 컨테이너일 뿐입니다. GetInfo를 호출하면 프로그램의 제어가 프로시저로 넘어가기 전에 MyRecord가 사용하는 메모리를 해제합니다.
Out 파라미터는 COM 같은 분산 객체 모델에서 함께 자주 사용됩니다. 초기화되지 않은 변수를 함수나 프로시저로 전달할 때에도 out 파라미터를 사용해야 합니다.

■ 타입 미지정 파라미터

var, const, out 파라미터를 선언할 때 타입 지정을 생략할 수 있습니다. (반면 값 파라미터는 반드시 타입이 지정되어야 합니다.) 예를 들면,

```
procedure TakeAnything(const C);
```

위 코드는 모든 타입의 파라미터를 다 전달받을 수 있는 TakeAnything이라는 프로시저를 선언합니다. 이런 루틴을 호출할 때 숫자나 타입이 지정되지 않은 숫자 상수를 넘길 수 없습니다.

프로시저나 함수의 바디 내에서 타입 미지정 파라미터(untyped parameter)는 어떤 타입과도 호환되지 않습니다. 타입 미지정 파라미터를 연산하려면 타입 캐스트를 해야 합니다. 일반적으로, 컴파일러는 타입 미지정 파라미터의 연산이 유효한지 확인할 수 없습니다.

다음 예에서는 Equal이라는 함수에서 타입 미지정 파라미터를 사용합니다. 이 Equal 함수는 두 변수의 특정 바이트 수를 비교합니다.

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N: Integer;
begin
  N := 0
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

다음과 같이 선언되었을 경우,

```
type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;
```

다음과 같이 Equal을 호출할 수 있습니다.

```
Equal(Vec1, Vec2, SizeOf(TVector));            // Vec1과 Vec2를 비교
Equal(Vec1, Vec2, SizeOf(Integer) * N);        // Vec1과 Vec2의 앞 N 요소를 비교
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5); // Vec1의 첫 5 요소와 마지막 소를 비교
Equal(Vec1[1], P, 4);                          // Vec1[1]과 P.X를, Vec1[2]과 P.Y를 비교
```

문자열 파라미터

짧은 문자열 파라미터를 갖는 루틴을 선언할 경우, 파라미터 선언에서 길이 지정을 할 수 없습니다. 즉, 다음 선언은,

```
procedure Check(S: string[20]); // 문법 에러
```

컴파일 에러를 일으킵니다. 그러나,

```
type TString20 = string[20];
procedure Check(S: TString20);
```

이 선언은 유효합니다. 특수 식별자 OpenString은 다양한 길이의 짧은 문자열 파라미터를 갖는 루틴을 선언하는 데 사용됩니다.

```
procedure Check(S: OpenString);
```

{$H-} 및 {$P+} 컴파일러 지시어를 모두 사용하면 예약어 string은 파라미터 선언에서 OpenString과 같습니다.
짧은 문자열, OpenString, $H 및 $P는 하위 호환성을 위해서만 지원됩니다. 새로운 코드에서는 긴 문자열을 사용하여 이러한 문제를 피할 수 있습니다.

배열 파라미터

배열 파라미터를 가진 루틴을 선언할 때는 파라미터 선언에 인덱스 타입 지정자 를 포함할 수 없습니다. 즉, 다음 선언은,

```
procedure Sort(A: array[1..10] of Integer); // 문법 에러
```

컴파일 에러를 일으킵니다. 그러나,

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: Tdigits);
```

이 선언은 유효합니다. 또다른 방법은 개방형 배열 파라미터(open array parameter)를 사용하는 것입니다.
델파이 언어는 동적 배열에 대해 값 의미 구조를 구현하지 않으므로, 동적 배열의 경우 값 파라미터라고 해도 동적 배열의 내용을 완전하게 복사하지 않습니다. 다음의 예제에서,

```
type
  TDynamicArray = array of Integer;
  procedure p(Value: TDynamicArray);
  begin
    Value[0] := 1;
  end;

  procedure Run;
var
    a: TDynamicArray;
begin
  SetLength(a, 1);
  a[0] := 0;
  p(a);
  Writeln(a[0]); // '1' 출력
end
```

Value가 값 파라미터임에도 불구하고, 루틴 P의 Value[0]에 대한 대입은 루틴을 호출한 측의 동적 배열의 내용을 수정한다는 것을 주목하십시오. 동적 배열을 완전히 복사할 필요가 있다면, 동적 배열의 값을 복사하기 위해 Copy 표준 프로시저를 사용하면 됩니다.

■ 개방형 배열 파라미터

개방형 배열 파라미터(open array parameter)를 사용하면 크기가 다른 배열들을 프로시저나 함수로 전달할 수 있습니다. 개방형 배열 파라미터를 가진 루틴을 정의하려면, 파라미터 선언에서 array[X..Y] of type이 대신 array of type을 사용하십시오. 예를 들면,

```
function Find(A: array of Char): Integer;
```

위 코드는 크기에 상관없는 문자 배열을 전달받아 정수를 리턴하는 Find라는 함수를 선언합니다.
개방형 배열 파라미터의 문법은 동적 배열 타입의 문법과 유사하지만 동일한 의미를 가지는 것은 아닙니다. 위의 예제에서는 동적 배열을 포함하여 모든 문자 배열을 전달받을 수 있는 함수를 만듭니다. 동적 배열만을 지정하는 파라미터를 선언하려면 타입 식별자를 지정해야 합니다.

```
type TDynamicCharArray = array of Char;
function Find(A: TdynamicCharArray): Integer;
```

동적 배열에 대한 자세한 내용은 4장의 "동적 배열"을 참조하십시오.
루틴의 바디 안에서 개방형 배열 파라미터에는 다음 규칙이 적용됩니다.

- 개방형 배열 파라미터의 인덱스는 항상 0에서 시작합니다. 첫 번째 요소는 0이고 두 번째 요소는 1, 이런 식입니다. 표준 Low 및 High 함수는 각각 0과 길이-1을 리턴합니다. SizeOf 함수는 루틴으로 전달한 실제 배열 크기를 리턴합니다.
- 개방형 배열 파라미터는 요소에 대해서만 액세스할 수 있습니다. 전체 개방형 배열 파라미터에 대한 대입은 허용되지 않습니다.
- 다른 프로시저나 함수로 전달할 때는 개방형 배열 파라미터 혹은 타입을 지정하지 않은 var 파라미터로만 전달할 수 있습니다. SetLength의 파라미터로는 전달할 수 없습니다.
- 배열 대신에 개방형 배열 파라미터의 기반(base) 타입 변수를 전달할 수 있습니다. 이런 경우 길이 1의 배열로 간주되게 됩니다.

배열을 개방형 배열 값 파라미터로 전달하면 컴파일러는 루틴의 스택 프레임에 배열의 로컬 복사본을 만듭니다. 아주 큰 배열을 전달할 경우 스택이 오버플로우(overflow)될 수도 있으므로 주의하십시오.
다음 예에서는 실수 배열에 있는 각 요소에 0을 대입하는 Clear 프로시저와 실수 배열에 있는 요소의 합을 계산하는 Sum 함수를 정의하기 위해 개방형 배열 파라미터를 사용합니다.

```
procedure Clear(var A: array of Real);
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0;
end
```

```
function Sum(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
    Sum := S;
end;
```

개방형 배열 파라미터를 사용하는 루틴을 호출할 때 개방형 배열 생성자를 전달할 수 있습니다. "개방형 배열 생성자"를 참조하십시오.

■ Variant 개방형 배열 파라미터

Variant 개방형 배열 파라미터를 사용하면 타입이 다른 표현식 배열을 단일 프로시저나 함수에 전달할 수 있습니다. 가변 개방형 배열 파라미터를 가진 루틴을 정의하려면 파라미터의 타입으로 array of const를 지정하십시오. 따라서,

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

위 코드는 여러 타입의 요소를 가진 배열에 대해 동작할 수 있는 DoSomething이라는 프로시저를 선언합니다.

array of const는 array of TVarRec와 같습니다. System 유닛에 선언되어 있는 TVarRec은 정수, 부울, 문자, 실수, 문자열, 포인터, 클래스, 클래스 참조, 인터페이스, 가변 타입 등의 값을 가질 수 있는 가변 부분이 있는 레코드를 나타냅니다. TVarRec의 VType 필드는 배열의 각 요소들의 타입을 나타냅니다. 일부 타입들은 값이 아닌 포인터로 넘겨집니다. 특히, 긴 문자열은 포인터로 전달되며 string으로 타입 캐스트되어야 합니다.

다음의 예제에서는 함수로 전달된 각 요소의 문자열 표현을 만들고 그 결과를 단일 문자열로 연결시키는 함수에서 가변 개방형 배열 파라미터를 사용합니다. 이 함수로 호출한 문자열 처리 루틴은 SysUtils에서 정의합니다.

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
```

```
  Result := '';
  for I := 0 to 66 do
  with Args[I] do
  case VType of
    vtInteger:    Result := Result + IntToStr(VInteger);
    vtBoolean:    Result := Result + BoolChars[VBoolean];
    vtChar:       Result := Result + VChar;
    vtExtended:   Result := Result + FloatToStr(VExtended^);
    vtString:     Result := Result + VString^;
    vtPChar:      Result := Result + VPChar;
    vtObject:     Result := Result + VObject.ClassName;
    vtClass:      Result := Result + VClass.ClassName;
    vtAnsiString: Result := Result + string(VAnsiString);
    vtCurrency:   Result := Result + CurrToStr(VCurrency^);
    vtVariant:    Result := Result + string(VVariant^);
    vtInt64:      Result := Result + IntToStr(VInt64^);
  end;
end;
```

개방형 배열 생성자를 사용하여 이 함수를 호출할 수 있습니다. 예를 들면,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

위와 같이 호출할 수 있으며 "test100 T3.14159TForm"이라는 문자열을 리턴합니다.

■ 기본 파라미터

프로시저나 함수의 헤더에서 기본 파라미터(default parameter) 값을 지정할 수 있습니다. 기본 값은 타입 지정 상수와 값 파라미터에 대해서만 허용됩니다. 기본값을 지정하려면, 파라미터 선언의 끝에 = 기호와 그 뒤를 이어 상수 표현식을 덧붙이면 됩니다. 이 상수 표현식은 파라미터 타입에 대입 호환(assignment-compatible) 가능한 타입이어야 합니다.
예를 들어, 다음과 같이 선언했다고 가정하면,

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

다음의 두 프로시저 호출은 같은 의미가 됩니다.

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

복수 파라미터를 한번에 선언한 경우 기본값을 지정할 수 없습니다. 따라서,

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

이것은 적합합니다. 그러나,

```
function MyFunction(X, Y: Real = 3.5): Real; // 문법 에러
```

이것은 적합하지 않습니다.
기본값을 가진 파라미터는 반드시 파라미터 리스트의 끝에 나와야 합니다. 즉, 기본값을 가진 첫 파라미터 뒤의 모든 파라미터에는 역시 기본값이 있어야 합니다. 따라서 다음 선언은 에러가 납니다.

```
procedure MyProcedure(I: Integer = 1; S: string) // 문법 에러
```

프로시저 타입에서 지정된 기본값은 실제 루틴에서 지정된 기본값보다 우선하여 적용됩니다. 따라서, 다음과 같이 선언한 경우,

```
type TResizer = function(X: Real; Y: Real = 1.0):Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
  F: TResizer;
  N: Real;
```

다음과 같은 문장에서,

```
F := Resizer;
F(N);
```

Resizer로 전달되는 값은 (N, 1.0) 입니다.
기본 파라미터는 상수 표현식으로 지정 가능한 값으로 제한됩니다.(4장의 "상수 표현식" 참조) 따라서 동적 배열 타입, 프로시저 타입, 클래스 타입, 클래스 참조 타입, 인터페이스타입의 파라미터는 기본값으로 nil 이외의 값은 가질 수 없습니다. 레코드 타입, variant 타

입, 파일 타입, 정적 배열 타입, 객체 타입의 파라미터는 기본값을 아예 가질 수 없습니다. 기본 파라미터 값이 있는 루틴 호출에 대한 자세한 내용은 5장의 "프로시저와 함수의 호출"을 참조하십시오.

■ 기본 파라미터와 오버로드된 루틴

오버로드된 루틴들에서 기본 파라미터 값을 사용할 경우, 모호한 파라미터 시그너처(signature)를 피해야 합니다. 예를 들어,

```
procedure Confused(I: Integer) overload;
  ...
procedure Confused(I: Integer; J: Integer = 0) overload;
  ...
Confused(X); // 어느 프로시저가 호출될까요?
```

위 코드에서는 사실상 어느 프로시저도 호출되지 않습니다. 이코드는 컴파일 에러를 일으킵니다.

■ forward 선언과 interface 선언의 기본 파라미터

루틴에 forward 선언이 있거나 루틴이 유닛의 interface 섹션에 나타나면 기본 파라미터 값은 forward 선언이나 interface 선언에서 지정해야 합니다. 이런 경우에 기본값은 정의적 선언(구현 선언)에서 생략될 수 있습니다. 그러나 정의적 선언에 기본값이 있으면 해당 기본값은 forward 선언이나 interface 선언에 있는 기본값과 정확하게 일치해야 합니다.

프로시저 및 함수의 호출

프로시저나 함수를 호출하면 프로그램 제어는 호출이 발생한 지점에서 루틴의 바디로 넘어갑니다. 선언된 루틴의 이름을 사용하거나 루틴을 가리키는 프로시저 변수를 사용하여 호출할 수 있습니다. 두 경우 모두, 루틴이 파라미터와 함께 선언되었다면 호출할 때 선언했던 파라미터들을 해당하는 순서와 타입으로 파라미터를 전달해야 합니다. 루틴으로 전달된 파라미터는 실제 파라미터(actual parameter)라고 하며, 루틴의 선언에 있는 파라미터는 형식 파라미터(formal parameter)라고 합니다.

루틴을 호출할 때 다음 사항들에 주의해야 합니다.

- 타입 지정 상수와 값 파라미터는 해당 형식 파라미터(formal parameter)의 타입과 대입 호환

가능한 타입이어야 합니다.

- 형식 파라미터가 타입이 지정되지 않은 경우가 아니라면, var 및 out 파라미터를 전달하는 표현식은 해당 형식 파라미터와 같은 타입이어야 합니다.
- var 및 out 파라미터로는 대입 가능한 표현식만 사용할 수 있습니다.
- 루틴의 형식 파라미터에 타입이 지정되지 않은 경우, 숫자 및 순수 상수를 실제 파라미터로 사용할 수 없습니다.

기본 파라미터 값을 사용하는 루틴을 호출할 경우, 기본값을 택할 첫 파라미터 다음에 오는 모든 실제 파라미터들도 기본값을 사용해야 합니다. SomeFunction(,,X)와 같은 형식의 호출은 맞지 않습니다.
루틴의 모든 파라미터를 생략하여 기본값만이 전달될 경우 괄호를 생략할 수 있습니다. 예를 들어, 프로시저가 다음과 같을 경우,

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string);
```

다음 호출들은 동일합니다.

```
DoSomething();
DoSomething;
```

개방형 배열 생성자

개방형 배열 생성자를 사용하면 함수나 프로시저 호출 내에서 직접 배열을 생성할 수 있습니다. 개방형 배열은 개방형 배열 파라미터 또는 variant 개방형 배열 파라미터로만 전달할 수 있습니다.
개방형 배열 생성자는 집합 생성자처럼 콤마로 구분된 표현식들을 대괄호([])로 묶은 것입니다. 예를 들어, 다음과 같이 선언한 경우,

```
var I, J: Integer;
procedure Add(A: array of Integer);
```

다음과 같이 Add 프로시저를 호출할 수 있습니다.

```
Add([5, 7, I, I + J]);
```

이것은 다음과 동일합니다.

```
var Temp: array[0..3] of Integer;
...
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

개방형 배열 생성자는 값 파라미터 또는 const 파라미터로만 전달할 수 있습니다. 생성자의 표현식들은 배열 파라미터의 기반 타입과 대입 호환 가능한 타입이어야 합니다. 가변 개방형 배열 파라미터의 경우 표현식이 다른 타입일 수 있습니다.

inline 지시어의 사용

델파이 컴파일러는 성능을 높이기 위해 함수나 프로시저에 inline 표시를 하는 것을 허용합니다. 함수나 프로시저가 어떤 기준에 맞으면 컴파일러는 루틴을 호출하는 대신 코드를 직접 삽입합니다. 인라이닝은 성능 최적화로서, 더 빠른 코드를 만들어낼 수 있지만 공간 면에서는 비효율적입니다. inline 루틴을 사용하면 컴파일러가 생성하는 바이너리 파일은 더 커지게 됩니다. inline 지시어는 다른 지시어들처럼 함수나 프로시저의 선언과 정의에서 사용됩니다.

```
procedure MyProc(x: Integer); inline;
begin
    // ...
end;

function MyFunc(y: Char) : String; inline;
begin
   // ..
end;
```

inline 지시어는 컴파일러에게 하나의 제안일 뿐입니다. 컴파일러가 특정 루틴을 인라이닝으로 처리할지에 대한 보장은 없습니다. 왜냐하면 인라이닝이 될 수 없는 여러 상황들이 있

기 때문입니다. 다음의 리스트는 인라이닝이 일어나거나 일어나지 못하는 조건들을 설명합니다.

- 모든 형태의 지연 바인딩 메소드에서는 인라이닝이 일어날 수 없습니다. 여기에는 virtual, dynamic, message 메소드가 포함됩니다.
- 어셈블리 코드를 포함하는 루틴은 인라인되지 않습니다.
- 생성자와 파괴자는 인라인되지 않습니다.
- 메인 프로그램 블록, 유닛의 initialization 및 finalization 블록은 인라인될 수 없습니다.
- 사용하기 전에는 정의되지 않는 루틴은 인라인되지않습니다.
- 개방형 배열 파라미터를 가진 루틴은 인라인될 수 없습니다.
- 패키지 내의 코드는 인라인될 수 있지만, 패키지 경계를 넘어서는 인라인되지 않습니다.
- 순환적으로 의존하는 유닛들 사이에는 인라이닝이 되지 않습니다. 여기에는 간접 순환 의존도 포함됩니다. 예를 들어, 유닛 A는 유닛 B를 uses 하고, 유닛 B 는 유닛 C를 uses하는데 C가 다시 A를 uses할 경우입니다. 이 예에서 유닛 B나 C의 코드는 유닛 A에서 인라인될 수 없습니다.
- 인라인될 코드가 순환 관계의 바깥에 있는 유닛에 있는 경우에는 순환 의존관계에 있는 유닛이라도인라인될 수 있습니다. 위의 예에서, 유닛 A가 유닛 D도 uses 한다면, 유닛 D의 코드는 순환 의존관계에 포함되지 않으므로 A에서 인라인될 수 있습니다.
- 루틴이 interface 섹션에서 정의되고 implementation 섹션에서 정의된 심볼을 액세스하면 그 루틴은 인라인될 수 없습니다.
- inline으로 표시된 루틴이 다른 유닛들로부터 외부 심볼을 사용하면 그런 모든 유닛들은 uses 문에서 지정되어야 하며, 그렇지 않을 경우 해당 루틴은 인라인될 수 없습니다.
- while-do 혹은 repeat-until 문의 조건 표현식에서 사용된 프로시저나 함수들은 인라인될 수 없습니다.
- 유닛 안에서, 인라인 함수의 바디는 함수 호출이 이루어지기 전에 정의되어야 합니다. 그렇지 않으면 컴파일러가 호출하는 코드에 이르렀을 때 함수의 바디는 컴파일러가 아직 모르는 상태이므로인라인될 수 없습니다.

인라인 루틴의 구현을 수정하면 해당 함수를 사용하는 모든 유닛들에 대해 재컴파일을 일으키게 됩니다. 이것은 유닛 interface 섹션의 수정에 의해서만 리빌드(rebuild)가 일어난다는 전통적인 리빌드 규칙과 다른 것입니다.
컴파일러 지지어 {$INLINE}을 사용하면 인라이닝을 더 잘 제어할 수 있습니다.

{$INLINE} 지시어는 인라인 루틴 호출 부분에서뿐만 아니라 해당 루틴의 정의 부분에서도 사용될 수 있습니다. 아래는 가능한 값들과 그 의미입니다.

값	정의에서의 의미	호출 측에서의 의미
{$INLINE ON}(기본)	iniline 지시어가 지정되었다면 해당 루틴이 인라인 가능하도록 컴파일합니다.	가능한 경우 인라인됩니다.
{$INLINE AUTO}	{$INLINE ON}처럼 동작하며, 추가로 inline으로 표시되지 않은 루틴들도 코드 사이즈가 32 바이트 이하이면 인라인됩니다.	호출하는 측에서는 {$INLINE AUTO}는 아무런 효과가 없습니다.
{$INLINE OFF}	해당 루틴은 inline으로 표시되었다고 하더라도 인라인되지 않습니다.	해당 루틴은 인라인되지 않습니다.