CHAPTER _4

데이터 타입, 변수 및 상수

4장에서는 델파이 언어에서 지원하는 수많은 데이터 타입들에 대해 살펴보고, 그 타입들을 다루는 방법과 변수와 상수에 대해서도 알아봅니다.

■타입의 분류 ■단순 타입(Simple type) ■문자열 타입(String type) ■구조 타입 ■포인터와 포인터 타입 ■프로시저 타입 ■Variant 타입 ■타입 호환 및 동등성 ■타입 선언 ■변수 ■선언된 상수

데이터 타입이란 본질적으로 데이터 종류에 대한 이름입니다. 변수를 선언할 때는 변수의 타입을 지정해주어야 하며, 변수의 타입은 변수가 가질 수 있는 값들과 변수에서 수행될 수 있는 동작들을 결정합니다. 모든 함수들과 마찬가지로, 모든 표현식은 특정 타입의 데이터를 리턴합니다. 대부분의 함수와 프로시저는 특정 타입의 파라미터를 필요로 합니다.

델파이 언어는 "강력하게 타입이 지정된(strongly typed)" 언어입니다. 즉, 다양한 데이터 타입을 구별하며, 항상 한 타입에 다른 타입을 대체할 수는 없습니다. 덕분에 일반적으로 컴파일러에서 데이터를 지능적으로 처리하고 코드를 더 철저히 검증하여, 진단하기 어려운 런타임 에러를 예방할 수 있습니다. 하지만 보다 높은 유연성이 필요한 경우에는 강력한 타입 지정을 우회할 수 있는 방법도 있습니다. 이러한 방법으로는 타입 캐스트(3장의 "타입 캐스트" 절 참조), 포인터("포인터와 포인터 타입" 절 참조), Variant 타입("Variant 타입" 절 참조), 레코드의 variant 타입 부분("구조 타입" 절의 "레코드의 가변 부분" 참조) 및 변수의 절대 주소 지정(4장의 "변수" 절의 "절대 주소" 참조) 등이 있습니다.

타입의 분류

다음은 델파이 데이터 타입들을 분류하는 몇 가지 방식들입니다.

- 일부 타입은 이미 정의된(기본 제공된) 타입이며, 이들 타입은 선언을 하지 않아도 컴파일러에서 자동으로 인식됩니다. 이책에서 설명하는 거의 모든 타입들은 이미 정의된 타입입니다. 그 외의 타입들은 선언에 의해 생성되며, 사용자 정의 타입과 라이브러리에 정의된 타입이 있습니다.
- 타입은 기본(fundamental) 또는 일반(generic)으로 분류될 수 있습니다. 기본 타입의 범위와 형식은 기본 CPU 및 운영 체제와 상관 없이 델파이 언어의 모든 구현에서 동일합니다. 일반 타입의 범위와 형식은 플랫폼에 따라 다르며, 여러 구현으로 달라질 수 있습니다. 이미 정의된 타입의 대부분은 기본 타입이지만, 일부 정수, 문자, 문자열 및 포인터 타입은 일반 타입(generic type)입니다. 일반 타입은 최적의 성능과 함께 이식성도 제공하기 때문에 가능하다면 일반 타입을 사용하는 것이 좋습니다. 하지만 일반(generic) 타입의 한 구현에서 다른 구현으로의 저장 형식에 변화 때문에 호환성 문제가 발생할 수도 있습니다(예를 들어, 데이터를 타입과 버전 정보 없이 그대로 바이너리 파일에 스트리밍 하려는 경우).
- 타입은 단순 타입(simple type), 문자열 타입, 구조 타입, 포인터 타입, 프로시저 타입 또는 Variant 타입으로 분류될 수 있습니다. 또한, 타입 식별자 자체도 특정 함수(High, Low 및 SizeOf 등)에 파라미터로 전달할 수 있기 때문에 특수한 "타입"에 속하는 것으로 간주될 수 있습니다.

다음은 델파이 데이터 타입들의 분류법입니다.

단순 타입 (simple)
- 서수 (ordinal)
- 정수 (integer)
- 문자 (character)
- 부울 (Boolean)
- 열거 (enumerated)
- 부분범위 (subrange)
- 실수 (real)

문자열 타입 (string)

구조 타입 (Structured)

집합 (set)
배열 (array)
레코드 (record)
파일 (file)
클래스 (class)
클래스 참조 (class reference)
인터페이스 (interface)

포인터 (pointer)
프로시저 (procedural)
Variant 타입
타입 식별자 (type identifier)

표준 함수 SizeOf는 모든 변수와 타입 식별자에서 사용할 수 있습니다. 이 함수는 지정된 타입의 데이터를 저장하기 위해 필요한 메모리의 양(바이트)을 나타내는 정수를 리턴합니다. 예를 들어, Longint 변수는 4바이트의 메모리를 사용하므로 SizeOf(Longint)는 4를 리턴합니다. 타입 선언은 다음 단원에서 설명합니다. 타입 선언에 대한 일반적인 내용은 "타입 선언" 절을 참조하십시오.

단순 타입(Simple type)

단순 타입에는 서수 타입과 실수 타입이 있으며, 순서가 있는 값들을 정의합니다.

서수 타입

서수 타입(ordinal type)에는 정수 타입, 문자 타입, 부울 타입, 열거 타입, 부분범위 타입이 있습니다. 서수 타입은 순서가 있는 값들을 정의하며, 처음 값을 제외한 모든 값에 유일한 이전 값(predecessor)이 있고 마지막 값을 제외한 모든 값에는 유일한 다음 값(successor)이 있습니다. 또한 각 값은 타입의 순서를 결정하는 순서성(ordinality)를 가집니다. 대부분의 경우 값에 순서 n이 있으면, 이값의 이전 값은 $n-1$ 순서를 가지고 다음 값은 $n+1$ 순서를 가집니다.

• 정수 타입의 경우에는 값 자체가 값의 순서입니다.

- 부분범위 타입은 그 타입의 기반 타입(base type)의 순서를 유지합니다.
- 다른 서수 타입의 경우, 기본적으로 처음 값이 순서 0이고 다음 값은 순서 1, 순서 2 등과 같이 됩니다. 열거 타입의 선언은 명시적으로 이런 기본값을 무시할 수 있습니다.

순서 값과 타입 식별자를 다루기 위한 몇몇 이미 정의된 함수들이 있습니다. 이미 정의된 함수에서 중요한 내용은 다음과 같이 요약할 수 있습니다.

함수	파라미터	리턴 값	주의
Ord	서수 표현식	표현식 값의 순서	Int64 인수를 사용하지 말 것
Pred	서수 표현식	표현식 값의 이전 값	
Succ	서수 표현식	표현식 값의 다음 값	
High	서수 타입 식별자 또는 서수타입의 변수	타입의 최고값	짧은 문자열 타입과 배열에 대해서도 동작함
Low	서수 타입 식별자 또는 서수타입의 변수	타입의 최저값	짧은 문자열 타입과 배열에 대해서도 동작함

예를 들어, High(Byte)는 바이트 타입의 최고값이 255이기 때문에 255를 리턴하며, Succ(2)는 2의 다음 값이 3이기 때문에 3을 리턴합니다.
표준 프로시저 Inc와 Dec는 서수 변수의 값을 증가시키거나 감소시킵니다. 예를 들어, Inc(I)는 I := Succ(I)와 동일하며, I가 정수 변수라면 I := I + 1입니다.

■정수 타입(integer type)
정수 타입은 전체 숫자의 서브셋을 나타냅니다. 일반적인 정수 타입은 Integer와 Cardinal입니다. 기본 CPU와 운영 체제에서 정수는 최적의 수행 성능을 제공하기 때문에, 가능한 모든 경우에 정수 타입을 사용하는 것이 좋습니다. 다음 표는 델파이 컴파일러의 정수 타입 범위와 저장 형식을 보여줍니다.

표 4.1 일반(generic) 정수 타입

타입	범위	형식
Integer	-2147483648..2147483647	부호를 가진 32비트
Cardinal	0..4294967295	부호 없는 32비트

기본 정수 타입에는 Shortint, Smallint, Longint, Int64, Byte, Word, Longword, UInt64가 있습니다.

표 4.2 기본(Fundamental) 정수 타입

타입	범위	형식
Shortint	-128..127	부호를 가진 8비트
Smallint	-32768..32767	부호를 가진 16 비트
Longint	-2147483648..2147483647	부호를 가진 32 비트
Int64	-2^63..2^63-1	부호를 가진 64 비트
Byte	0..255	부호 없는 8 비트
Word	0..65535	부호 없는 16 비트
Longword	0..4294967295	부호 없는 32 비트
UInt64	0..2^64?1	부호 없는 64 비트

일반적으로 정수에서의 산술 연산은 정수 타입 값을 리턴하며, 이것은 32비트 Longint와 같습니다. 하나 이상의 Int64 피연산자를 가지고 수행되는 연산에서만 Int64 타입 값을 리턴합니다. 따라서 다음과 같은 코드는 잘못된 결과를 발생시킵니다.

```
var
  I: Integer;
  J: Int64;
begin
  ...
  I := High(Integer);
  J := I + 1;
```

이런 코드에서 Int64 리턴 값을 얻으려면, I를 Int64로 타입 캐스트하면 됩니다.

```
...
J := Int64(I) + 1;
```

자세한 내용은 3장의 "표현식" 절의 "산술 연산자"를 참조하십시오.
정수 타입의 마지막 값을 증가시키거나 처음 값을 감소시키는 경우에는 결과값이 범위의 시작 값이나 끝 값으로 돌아갑니다. 예를 들어 Shortint 타입은 범위가 -128..127입니다. 따라서 다음과 같이 코드를 실행할 경우,

```
var I: Shortint;
...
I := High(Shortint);
I := I + 1;
```

Note

정수 인수를 요구하는 일부 표준 루틴은 Int64 값을 32비트 값으로 잘라냅니다. 그러나 High, Low, Succ, Pred, Inc, Dec, IntToStr, IntToHex 루틴은 Int64 인수를 완벽하게 지원합니다. 또한 Round, Trunc, StrToInt64, StrToInt 64Def 함수는 Int64 값을 리턴합니다. 몇몇 루틴들은 Int64 값을 전혀 받을 수 없습니다.

이 경우에 I의 값은 -128입니다. 하지만 컴파일러의 범위 검사(range-checking)를 활성화하면 이 코드는 런타임 에러가 발생합니다.

■문자 타입 (character type)

기본 문자 타입은 AnsiChar와 WideChar입니다. AnsiChar 값은 바이트 크기(8비트)의 문자입니다. 보통 멀티바이트인 지역 문자셋에 따라 정렬됩니다. AnsiChar은 본래는 이름을 그대로 ANSI 문자셋을 따라 설계된 것이지만, 지금은 현재의 지역 문자셋을 따르도록 확장되었습니다.

WideChar 문자는 각 문자들을 나타내는데 2바이트 이상을 사용합니다. 현재의 문자셋 구현에서, WideChar는 유니코드 문자셋에 따라 정렬되는 워드 크기(16비트)의 문자입니다 (향후의 구현에서는 더 길어질 수도 있습니다). 처음 256 유니코드 문자는 ANSI 문자와 일치합니다.

일반적인 문자 타입은 Char 타입으로, 이것은 WideChar와 같습니다. Char 타입의 구현은 변경될 수 있기 때문에 크기가 다양한 문자를 취급해야 하는 프로그램을 작성할 때는 하드코딩된 상수를 사용하기 보다는 표준 함수인 SizeOf를 사용하는 것이 바람직합니다. Char가 WideChar이므로 SizeOf(Char)가 2를 리턴합니다.

정수와 마찬가지로 문자값은 값을 증가시키거나 감소시킬 때 범위를 벗어 나면, 처음 값이나 마지막 값으로 되돌아갑니다 (범위 검사가 활성화되지 않은 경우). 예를 들어, 다음과 같이 코드를 실행할 경우

Note

'A'와 같이 길이가 1인 문자열 상수는 문자 값으로 나타낼 수 있습니다. 이미 정의된 함수 Chr는 AnsiChar 또는 WideChar 범위 내에서 모든 정수를 문자값으로 리턴합니다. 예를 들어, Chr(65)는 문자 A를 리턴합니다.

```
var
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
  Inc(Letter);
end;
```

Letter 값은 A(ASCII 65)가 됩니다. Letter는 Char, 즉 WideChar이므로 High(Letter)는 #$FFFF입니다. 첫번째 Inc(Letter) 호출로 Letter의 값은 0으로 되돌아가고, 뒤이은 Inc(Letter) 호출로 'A' 문자의 코드 값만큼 증가됩니다.

■부울 타입 (Boolean type)

부울 타입으로는 Boolean, ByteBool, WordBool, LongBool의 네가지가 있습니다. 일반적으로 Boolean 타입을 가장 많이 사용합니다. 다른 부울 타입들은 다른 언어나 다른 운영체제와의 호환성을 위해 존재합니다.

부울 변수는 1바이트의 메모리를 사용하고, ByteBool 변수도 1바이트를 사용하며, WordBool 변수는 2바이트(1워드)를 사용하고, LongBool 변수는 4바이트(2워드)의 메모리를 사용합니다.

부울 변수는 이미 정의된 상수인 True 및 False로 표시됩니다. 다음은 부울 타입의 관계를 보여 줍니다.

Boolean	ByteBool, WordBool, LongBool
False 〈 True	False ◊ True
Ord(False) = 0	Ord(False) = 0
Ord(True) = 1	Ord(True) ◊ 0
Succ(False) = True	Succ(False) = True
Pred(True) = False	Pred(False) = True

ByteBool, LongBool 또는 WordBool 타입의 값은 이 타입의 순서가 0이 아닐 경우 True로 간주됩니다. 부울 값이 있어야 할 위치에 0이 아닌 순서 값이 있으면 컴파일러는 자동으로 True로 변환합니다.

위의 설명은 값 자체가 아닌 부울 값의 순서와 관련된 것입니다. 델파이에서 부울 표현식은 정수나 실수와 동등하게 다룰 수 없습니다. 따라서 X가 정수 변수인 경우 다음 문법은,

```
if X then ...;
```

컴파일 에러를 냅니다. 변수를 부울 타입으로 타입 캐스트하는 것은 신뢰성이 떨어지지만 다음의 두 문법 모두가 가능합니다.

```
if X <> 0 then ...;    { 부울 값을 리턴하는 더 긴 표현식을 사용함 }

var OK: Boolean;       { 부울 변수를 사용함 }
...
if X <> 0 then OK := True;
if OK then ...;
```

■열거 타입 (Enumerated type)

열거 타입은 값을 나타내는 식별자를 단순히 나열함으로써 순서를 가진 값의 집합을 정의합니다. 값에는 특별한 고유의 의미가 없습니다. 열거 타입을 선언하려면 다음 문법을 사용합니다.

```
type typeName = (val1, ..., valn)
```

여기서 *typeName*과 각 *val*은 유효한 식별자입니다. 예를 들어 다음과 같이 선언할 경우

```
type Suit = (Club, Diamond, Heart, Spade);
```

Suit라는 열거 타입을 정의하며, Club, Diamond, Heart, Spade의 값을 가질 수 있습니다. 여기서 Ord(Club)의 리턴값은 0이 되고, Ord(Diamond)의 리턴값은 1이 되는 식입니다. 열거 타입을 선언할 때는 각각의 *val*이 *typeName* 타입의 상수가 되도록 선언합니다. *val* 식별자가 같은 유효 범위(scope) 내에서 다른 목적으로 사용되는 경우에는 이름 충돌이 발생합니다. 예를 들어 다음과 같이 타입을 선언한다고 가정하십시오.

```
type TSound = (Click, Clack, Clock);
```

공교롭게도 Click은 TControl과 그로부터 파생된 VCL의 모든 객체에 대해 이미 정의된 메소드의 이름이기도 합니다. 따라서 애플리케이션을 작성할 때 다음과 같이 이벤트 핸들러를 작성할 경우,

```
procedure TForm1.DBGrid1Enter(Sender: TObject);
var Thing: TSound;
begin
  ...
  Thing := Click;
end;
```

이 문법은 컴파일 에러가 발생합니다. 컴파일러는 프로시저의 유효 범위 내에서 Click을 TForm의 Click 메소드로 해석합니다. 이러한 문제는 식별자에 한정자(qualifier)를 추가하여 해결할 수 있습니다. 즉, TSound가 MyUnit에서 선언된다면 다음과 같이 합니다.

```
Thing := MyUnit.Click;
```

하지만, 더 좋은 해결 방법은 다른 식별자와 충돌되지 않을 상수 이름을 선택하는 것입니다. 예를 들면, 다음과 같습니다.

```
type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe);
```

변수 선언에서 (*val*$_1$, ..., *val*$_n$) 문법을 마치 데이터 타입 이름인 것처럼 직접 사용할 수 있습니다.

```
var MyCard: (Club, Diamond, Heart, Spade);
```

하지만 MyCard를 이런 방식으로 선언하면 같은 유효 범위 내에서 이들 상수 식별자를 사용하여 다른 변수를 선언할 수 없습니다. 따라서

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

이 문법은 컴파일 에러가 발생합니다. 그러나,

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

이 문법은 정확하게 컴파일되며, 다음의 예와 같은 의미를 가지게 됩니다.

```
type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;
```

■순서값이 지정된 열거 타입

기본적으로 열거 값의 순서는 0에서 시작하고, 타입 선언에 식별자가 나열된 순서를 따릅니다. 선언의 일부 또는 모든 값에 순서 값을 명시적으로 지정하여 순서를 무시하고 재지정할 수 있습니다. 값에 순서를 지정하려면 식별자 뒤에 =*constantExpression*을 추가합니다. 여기서 *constantExpression*은 정수값을 리턴하는 상수 표현식입니다(4장의 "선언된 상수" 절의 "상수 표현식" 참조). 예를 들면 다음과 같습니다.

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

이 문법은 Size라는 타입을 정의하고, 이 타입의 가능한 값은 Small, Medium, Large입니다. 여기서 Ord(Small)은 5를 리턴하고, Ord(Medium)은 10을 리턴하고, Ord(Large)는 15를 리턴합니다.

사실, 열거 타입은 선언의 상수들의 최소 및 최대 순서값을 최소 및 최대 값으로 가지는 부분 범위입니다. 위 예제에서 Size 타입은 11개의 가능한 값을 가지며, 이들 순서는 5에서 15까지입니다. (따라서 array[Size] of Char는 11 문자의 배열을 나타냅니다.) 이들 값 중 단 3개만 이름을 가지고 있지만, 이름이 없는 나머지 값들도 타입 캐스트나 Pred, Succ, Inc, Dec 같은 루틴들을 통해서 액세스할 수 있습니다. 다음 예제에서, Size의 범위 내의 "익명" 값은 변수 X에 지정됩니다.

```
var X: Size;
X := Small;     // Ord(X) = 5
X := Size(6);   // Ord(X) = 6
Inc(X);         // Ord(X) = 7
```

순서 값이 명시적으로 지정되지 않은 모든 값은 리스트에서 이전 값의 순서 값보다 1이큰 순서 값을 가집니다. 처음 값에 순서 값이 지정되지 않은 경우, 처음 값의 순서 값은 0입니다. 즉, 다음과 같은 선언에서,

```
type SomeEnum = (e1, e2, e3 = 1);
```

SomeEnum은 2개의 값만 가집니다. Ord(e1)은 0을 리턴하고, Ord(e2)는 1을 리턴하고, Ord(e3) 역시 1을리턴합니다. e2와 e3은 같은 순서를 갖고 있기 때문에 이들은 같은 값을 나타냅니다.

다음의 예처럼, 특정 값이 지정되지 않은 열거 상수는 런타임 타입 정보(RTTI)를 가집니다.

```
type SomeEnum = (e1, e2, e3);
```

반면 다음과 같이 특정 값이 지정된 열거 상수는 RTTI를 가지지 않습니다.

```
type SomeEnum = (e1 = 1, e2 = 2, e3 = 3);
```

■부분범위 타입 (Subrange type)

부분범위 타입은 다른 서수 타입(즉, 기반(base) 타입) 값들의 서브셋을 나타냅니다. *Low..High* 형식의 문법에서 *Low*와 *High*가 같은 서수 타입의 상수 표현식이고 *Low*가 *High*보다 작은 경우, 이런 형식의 모든 문법은 *Low*와 *High* 사이의 모든 값을 포함하는 부분범위 타입입니다. 예를 들어, 다음과 같은 열거 타입을 선언하면,

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

다음과 같이 부분범위 타입을 정의할 수 있습니다.

```
type TMyColors = Green..White;
```

여기서 TMyColors는 Green, Yellow, Orange, Purple, White 값을 포함합니다.

부분범위 타입을 정의하기 위해 숫자 상수와 문자(길이가 1인 문자열 상수)를 사용할 수도 있습니다.

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

숫자 또는 문자 상수를 사용하여 부분범위를 정의할 때 기반 타입은 지정된 범위를 포함하는 가장 작은 정수나 문자 타입입니다.

Low..High 문법 자체가 타입 이름과 같은 역할을 하기도 합니다. 따라서 이를 직접 변수 선언문에 사용할 수 있습니다. 예를 들면,

```
var SomeNum: 1..500;
```

이 코드는 1부터 500 사이 범위의 값이 될 수 있는 정수 변수를 선언합니다.
부분범위의 각 값의 순서 값은 기반(base) 타입으로부터 유지됩니다. 위의 첫 예제에서 Color가 Green 값을 가지는 변수라면, Ord(Color)는 Color가 TColors 타입이든 TMyColors 타입이든 상관 없이 2를 리턴합니다. 기본 타입이 정수 타입이나 문자 타입이더라도 값은 부분범위의 시작 또는 끝으로 랩어라운드되지 않습니다. 부분 범위의 경계를 넘어선 값으로 증가시키거나 감소시키면 이 값은 기반 타입으로 변환됩니다. 따라서 다음과 같은 경우,

```
type Percentile = 0..99;
var I: Percentile;
...
I := 100;
```

이 문법은 에러가 발생합니다. 반면,

```
...
I := 99;
Inc(I);
```

이 문법은 I에 값 100을 대입합니다. (컴파일러의 범위 검사가 활성화되지 않은 경우.)
부분범위의 정의에서 상수 표현식을 사용하게 되면 문법상 혼란의 여지가 있습니다. 이것은, 모든 타입 선언에서 = 다음의 첫 의미 있는 문자가 왼쪽 괄호인 경우, 컴파일러는 열거 타입이 정의되는 것으로 간주하기 때문입니다. 따라서 다음과 같은 코드에서는,

```
const
  X = 50;
  Y = 10;
type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

에러가 발생합니다. 이 문제를 피해가려면, 타입 선언에서 다음과 같이 시작 괄호를 사용하

지 않으면 됩니다.

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

실수 타입(Real type)

실수 타입은 부동 소수점으로 표현할 수 있는 일련의 숫자를 정의합니다. 다음 표는 기본 실수 타입의 범위와 저장 형식을 나타낸 것입니다.

표 4.3 기본적인 실수 타입

타입	범위	유효 자리수	바이트 크기
Real48	-2.9×10^{-39} .. 1.7×10^{38}	11–12	6
Single	-1.5×10^{-45} .. 3.4×10^{38}	7–8	4
Double	-5.0×10^{-324} .. 1.7×10^{308}	15–16	8
Extended	-3.6×10^{-4951} .. 1.1×10^{4932}	10–20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	10–20	8
Currency	-922337203685477.5808.. 922337203685477.5807	10–20	8

일반적인 타입인 Real은 현재의 구현에서는 Double과 동일합니다.

표 4.5 일반적인 실수 타입

타입	범위	유효 자리수	바이트 크기
Real	-5.0×10^{-324} .. 1.7×10^{308}	15–16	8

Note

이전 버전의 오브젝트 파스칼에서 Real은 6바이트 Real48 타입이었습니다. 6바이트 실수 타입을 사용하는 이전의 코드를 다시 컴파일하는 경우, 이를 Real48로 변경해야 할 수도 있을 것입니다. 또한 {$REALCOMPATIBILITY ON} 컴파일러 지시어를 이용하여 Real을 다시 6바이트 타입으로 인식하게 할 수도 있습니다.

다음의 설명은 기본 실수 타입에 적용됩니다.

- Real48은 하위 호환성을 위해 유지됩니다. 이 타입의 저장 형식은 Intel 프로세서 아키텍처의 네이티브 형식이 아니므로 다른 부동 소수점 타입보다 낮은 성능을 보이게 됩니다.
- Extended는 다른 실수 타입보다 더 큰 정밀도를 제공하지만, 이식성이 떨어집니다. 플랫폼들 사이에 공유할 데이터 파일을 쓰는 경우에는 Extended 사용에 주의하십시오.
- Comp (computational) 타입은 Intel 프로세서 아키텍처의 네이티브 형식이며 64비트 정수를 표현합니다. 하지만 이 타입은 서수(ordinal) 타입처럼 동작하지 않기 때문에 실수로 분류됩니다. (예를 들어 Comp 값은 증가시키거나 감소시킬 수 없습니다.) Comp는 하위 호환성을 위해서만 유지됩니다. 성능을 향상시키기 위해 Int64 타입을 사용하십시오.

- Currency는 금액 계산에서 반올림 에러를 최소화하는 고정 소수점 데이터 타입입니다. 이 타입은 소수점 위치를 나타내는 4개의 최하위 숫자(least significant digits LSD)를 가진 64비트 정수로 저장됩니다. 대입문과 표현식에서 다른 실수 타입과 같이 사용되면, Currency 값은 자동적으로 10000으로 나눠지거나 곱해집니다.

문자열 타입(String type)

문자열이란 일련의 문자를 나타냅니다. 오브젝트 파스칼은 다음의 이미 정의된 문자열 타입을 지원합니다.

표 4.5 문자열 타입

타입	최대 길이	필요 메모리	용도
ShortString	255 문자	2 ~ 256 바이트	하위 호환성
AnsiString	~2^31 문자	4바이트 ~ 2GB	8비트(ANSI) 문자, DBCS ANSI, MBCS ANSI, 유니코드 문자 등
UnicodeString	~2^30 문자	4바이트 ~ 2GB	유니코드 문자, 8비트(ANSI) 문자, 멀티 유저 서버 및 멀티 언어 애플리케이션
WideString	~2^30 문자	4바이트 ~ 2GB	유니코드 문자 멀티 유저 서버 및 멀티 언어 애플리케이션. 일반적으로 UnicodeString을 추천.

Note

WideString은 COM BSTR 타입과의 호환성을 위해 제공됩니다. COM 애플리케이션이 아닌 경우 일반적으로는 UnicodeString을 사용해야 합니다. UnicodeString은 대부분의 목적에 추천되는 스트링 타입입니다. 디폴트로 기본 string 타입은 UnicodeString입니다.

문자열 타입들은 대입문이나 표현식에서 서로 섞어 사용할 수 있습니다. 컴파일러는 자동으로 필요한 변환을 수행합니다. 그러나, 함수나 프로시저에 참조로(by reference) 전달된 문자열(var 및 out 매개 변수 등)은 적절한 타입이어야 합니다. 문자열은 다른 문자열 타입으로 명시적으로 타입 캐스트될 수 있습니다. 하지만 멀티 바이트 스트링(UnicodeString, WideString)에서 단일 바이트 스트링(AnsiString, ShortString)으로 타입 캐스트할 경우 데이터 손실이 생길 수 있습니다. 3장의 "타입 캐스트" 절을 참조하십시오.
다음과 같은 언급해둘 만한 몇가지 특수한 문자열 타입들도 있습니다.

- 코드 페이지를 가진 AnsiString은 다음과 같이 정의됩니다. type mystring = AnsiString (CODEPAGE) 이것은 특정 코드 페이지의 내부 데이터를 유지하기 위한 친화성(affinity)을 가진 AnsiString입니다.
- RawByteString 타입은 AnsiString($FFFF)으로 정의되어 있습니다. RawByteString은 어떤

코드 페이지의 문자열 데이터이든 코드 페이지 변환 없이 전달이 가능하도록 해줍니다. RawByteString은 const나 값 타입 파라미터, 혹은 함수의 리턴 타입으로만 사용되어야 합니다. 이 타입은 참조로(by reference) 전달되어서는 안되며, 변수로서 생성해서도 안됩니다.

- UTF8String은 UTF-8(가변 바이트 수의 유니코드)로 인코딩된 문자열을 나타냅니다. 이 타입은 UTF-8에 해당하는 코드 페이지 값을 가진 AnsiString입니다.

예약어 string은 일반적인 타입 식별자와 같은 기능을 합니다. 예를 들면,

```
var S: string;
```

위 코드는 문자열을 갖는 변수 S를 생성합니다. 컴파일러는 string 뒤에 뒤이어 괄호로 둘러싸인 숫자가 없을 경우 string을 UnicodeString으로 해석합니다.
{$H-} 지시어를 사용하면 string을 ShortString으로 인식합니다. 이것은 예전의 16비트 델파이 코드나 터보 파스칼 코드를 현재의 프로그램에서 사용할 경우 유용할 수 있습니다.
예약어 string은 특정 길이를 갖는 ShortString을 선언할 때도 사용된다는 점을 알아두십시오.
표준 함수 Length는 문자열 내의 문자 수를 리턴합니다. SetLength 프로시저는 문자열의 길이를 지정합니다. SizeOf 함수는 문자열의 바이트 수를 리턴하므로, SizeOf의 리턴 값은 Length와는 다른 결과가 나올 수 있다는 점을 유의하십시오.
문자열들의 비교는 각 문자열 내의 상호 해당 위치의 문자 순서값에 의해 결정됩니다. 길이가 서로 다른 두 문자열에서는, 짧은 문자열에 해당 문자가 없는 긴 문자열의 문자들이 더 큰 값을 가집니다. 예를 들어, 'AB'는 'A'보다 더 큽니다. 즉, 'AB' 〉 'A'는 True를 리턴합니다. 길이가 0인 문자열은 가장 작은 값을 가집니다.
문자열 변수를 배열처럼 인덱스로 다룰 수 있습니다. S가 UnicodeString이 아닌 문자열 변수이고 i가 정수 표현식인 경우, S[i]는 S의 i번째 바이트이지만, i번째 문자가 아니거나 완전한 문자 자체가 아닐 수도 있습니다. 하지만, UnicodeString에서 인덱스를 한 결과는 항상 문자를 리턴합니다. S가 ShortString이나 AnsiString일 경우 S[i]는 AnsiChar 타입입니다. S가 WideString이면 S[i]는 WideChar 타입입니다. 단일 바이트(웨스턴) 로캘(locale)에서, MyString[2] := 'A'; 문은 MyString의 두 번째 문자에 값 A를 대입합니다. 다음 코드는 표준 AnsiUpperCase 함수를 사용하여 MyString을 대문자로 변환합니다.

```
var I: Integer;
begin
  I := Length(MyString);
  while I > 0 do
  begin
    MyString[I] := AnsiUpperCase(MyString[I]);
    I := I - 1;
  end;
end;
```

이러한 방식으로 문자열을 인덱싱하는 것은 문자열의 마지막을 겹쳐쓰게 되어 액세스 위반(access violation)을 일으킬 수도 있기 때문에 주의해야 합니다. 또, 긴 문자열 인덱스를 var 파라미터로 전달하는 것도 비효율적인 코드가 될 수 있으므로 피하십시오.
변수에 문자열 상수 값, 또는 문자열을 리턴하는 다른 어떤 표현식이든 대입할 수 있습니다. 문자열의 길이는 문자열 값이 대입될 때마다 동적으로 변합니다. 예를 들면 다음과 같습니다.

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';    { 공백 }
MyString := '';     { 빈 문자열 }
```

자세한 내용은 3장의 "문자열" 및 "문자열 연산자"를 참조하십시오.

짧은 문자열(Short string)

ShortString은 0에서 255까지의 길이를 가질 수 있는 단일 바이트 문자로 된 문자열입니다. ShortString의 길이는 동적으로 변할 수 있지만, 이 문자열의 메모리는 정적으로 256바이트가 할당되어 있습니다. 첫 바이트에는 문자열의 길이가 저장되고, 나머지 255바이트는 문자들을 저장하는데 사용됩니다.
S가 ShortString 변수인 경우, Ord(S[0])은 Length(S)와 같이 S의 길이를 리턴합니다. S[0]에 값을 대입하면 SetLength를 호출한 것처럼 S의 길이가 변합니다. ShortString은 하위 호환성을 위해서만 유지됩니다.
델파이 언어는 사실상 ShortString의 하위 타입인 짧은 문자열 타입들을 지원합니다. 이런 짧은 문자열 타입들에서 길이의 최대값은 0에서 255 문자까지 지정 가능하며, 예약어 string에 대괄호로 묶은 숫자를 추가하여 표현합니다. 예를 들면,

```
var MyString: string[10];
```

위 코드는 최대 길이가 100 문자인 MyString이라는 변수를 만듭니다. 이것은 다음의 선언과 동일합니다.

```
type CString = string[100];
var MyString: CString;
```

이러한 방식으로 선언된 변수는 타입에서 필요로 하는 만큼의 메모리가 할당됩니다. 즉, 지정한 최대 길이에 1바이트를 더한 크기가 할당됩니다. 위 예에서, 이미 정의된 ShortString 타입 변수의 경우 256바이트를 사용하는 반면, MyString은 101바이트를 사용합니다.
짧은 문자열 변수에 값을 지정할 때, 타입의 최대 길이를 초과하는 경우에는 문자열이 잘립니다.
표준 함수 High와 Low는 짧은 문자열 타입 식별자와 변수에 동작합니다. High는 짧은 문자열 타입의 최대 길이를 리턴하고 Low는 0을 리턴합니다.

AnsiString

긴 문자열이라고 부르기도 하는 AnsiString은 동적 할당 문자열로서 최대 길이는 사용 가능한 메모리에 의해서만 제한됩니다.
긴 문자열 변수는 문자열 정보를 저장하고 있는 구조입니다. 긴 문자열 변수가 비어 있으면, 즉 0 길이의 문자열을 가지고 있으면, 포인터는 nil이며 그 문자열은 추가 메모리를 사용하지 않습니다. 긴 문자열 변수가 비어있지 않으면 포인터는 동적으로 할당된 메모리 블럭을 가리키며, 이 메모리에 문자열 값이 들어 있게 됩니다. 이 메모리는 힙(heap)에 할당되지만 완전히 자동으로 관리되기 때문에 사용자 코드가 필요하지 않습니다. AnsiString 구조는 32비트 길이의 인디케이터, 32비트 참조 카운트, 문자당 바이트 수를 나타내는 16비트 데이터 길이, 그리고 16비트 코드 페이지를 포함합니다.
AnsiString 변수는 주로 단일 바이트 데이터에 사용되지만, 유니코드 데이터도 가질 수 있습니다.
긴 문자열 변수는 포인터를 가지기 때문에, 둘 이상의 긴 문자열 변수가 추가로 메모리를 쓰지 않고도 같은 값을 참조할 수 있습니다. 컴파일러는 이러한 점을 이용하여 리소스를 절약하고 더 빨리 대입을 수행합니다. 긴 문자열 변수가 파괴되거나 새로운 값이 지정될 때마다

이전 문자열(변수의 이전 값)의 참조 카운트는 감소되고 새로운 값의 참조 카운트는 증가됩니다. 문자열의 참조 카운트가 0에 이르면 메모리는 해제됩니다. 이러한 프로세스를 참조 카운팅(reference counting)이라고 합니다. 문자열의 인덱스를 이용하여 문자열에서 한 문자의 값을 변경할 경우 참조 카운트가 1보다 크면 문자열의 복사본이 만들어집니다. 이것을 copy-on-write 방식이라고 합니다.

UnicodeString

UnicodeString 타입은 동적 할당 문자열로서 최대 길이는 사용 가능한 메모리에 의해서만 제한됩니다.

단일 바이트 문자셋(SBCS)에서, 문자열의 각 바이트는 1문자를 나타냅니다. 멀티바이트 문자셋(MBCS)에서는 일부 문자들은 1바이트로 표현되고 다른 문자들은 2바이트 이상으로 표현됩니다. 멀티바이트 문자셋(특히 2바이트 문자셋(DBCS))은 아시아권 언어들에서 광범위하게 사용됩니다.

유니코드는 멀티바이트 문자 인코딩들의 집합입니다. 유니코드 문자와 문자열은 와이드 문자와 와이드 문자 문자열로도 불립니다.

Win32 플랫폼은 단일 바이트 및 멀티 바이트 문자셋 뿐만 아니라 유니코드도 지원합니다. 윈도우 운영체제는 유니코드 UCS-2를 지원합니다.

유니코드 문자셋에서는, 모든 문자는 하나 혹은 그 이상의 바이트로 표시됩니다. 따라서 유니코드 문자열은 개별 바이트들의 연속이 아니라 멀티 바이트 워드들의 연속입니다. 문자 크기는 유니코드 인코딩의 타입에 따라 달라집니다.

예를 들어 UTF-8에서는, 문자들은 1바이트에서 4바이트까지가 될 수 있습니다. UTF-8에서 첫 128 유니코드 문자들은 US-ASCII 문자들과 일치합니다.

UTF-16도 역시 자주 사용되는 유니코드 인코딩이며, 그 문자들은 2바이트 혹은 4바이트로 되어 있습니다. 전세계 문자들의 대다수가 기본 다국어 평면(Basic multilingual plane, BMP)에 있고, 2바이트로 표현될 수 있습니다. 남은 문자들은 2개의 2바이트 문자들인 서로게이트 페어(surrogate pairs)를 필요로 합니다.

더 자세한 정보를 위해서는, 유니코드 표준(http://www.unicode.org/standard/standard.html)을 살펴보시기 바랍니다.

UnicodeString 타입은 AnsiString 타입과 완전히 똑 같은 구조를 가지고 있습니다. UnicodeString의 데이터는 UTF-16으로 인코딩됩니다.

UnicodeString 타입에서 유니코드 데이터가 더 권장되기는 하지만, ANSI 문자열 데이터도 가질 수 있습니다.

UnicodeString과 AnsiString은 같은 구조를 가지고 있으므로 두 타입은 매우 비슷하게 동작합니다. UnicodeString 변수가 비어 있으면 추가 메모리를 사용하지 않습니다. 비어있지 않을 때는 문자열 값을 가진 동적으로 할당된 메모리 블럭을 가리키며, 이런 메모리 관리는 사용자에게 투명하게 처리됩니다. UnicodeString 변수는 참조 카운트되며, 추가 메모리 소모 없이 둘 이상의 UnicodeString 문자열이 동일 값을 참조할 수 있습니다.

UnicodeString의 변수는 인덱스로 문자를 다룰 수 있습니다. 인덱스는 AnsiString에서처럼 1에서 시작합니다.

UnicodeString은 다른 모든 문자열 타입들과 대입 호환이 됩니다. 하지만, AnsiString과 UnicodeString 사이의 대입은 적절한 변환을 하게 됩니다. UnicodeString 타입을 AnsiString 타입에 대입하는 것은 권장되지 않으며 데이터 손실을 가져올 수 있습니다.

델파이는 AnsiString, WideChar, PWideChar, WideString을 통해서도 유니코드 문자와 문자열을 지원할 수 있습니다.

유니코드의 사용에 대한 정보를 더 찾아보시려면, 온라인 헬프에서 "Unicode in RAD Studio Enabling Unicode in Your Application" 부분을 참고하십시오.

WideString

WideString 타입은 16비트 유니코드 문자들을 가지는 동적 할당 문자열을 나타냅니다. 어떤 측면에서 WideString은 AnsiString과 비슷합니다. Win32에서 WideString은 COM BSTR 타입과 호환됩니다.

WideString은 COM 애플리케이션에서 사용하기에 적당합니다. 하지만 WideString은 참조 카운팅이 지원되지 않으며, UnicodeString은 COM이 아닌 애플리케이션들에 더 유연하고 효율적입니다.

WideString 멀티바이트 문자열의 인덱스 동작은 신뢰할만 하지 못합니다. S[i]는 S에서 i번째 문자가 아니라 i번째의 바이트를 나타내기 때문입니다. 하지만 표준 문자열 관리 함수들에는 멀티바이트기능의 해당 함수들도 있으며, 이 함수들은 로캘 특정의 문자 순서도 구현합니다. (멀티바이트 함수들의 이름은 보통 Ansi-로 시작합니다. 예를 들어 StrPos 함수의 멀티바이트 버전은 AnsiStrPos입니다.) 멀티바이트 문자 지원은 운영체제에 의존하며 현재 로캘에 기반합니다.

델파이에서 Char와 PChar는 기본적으로 각각 WideChar와 PWideChar입니다.

Null 종료 문자열

C와 C++를 포함한 많은 프로그래밍 언어들에는 문자열 전용 데이터 타입이 없습니다. 이

들 언어와 그런 언어로 구축된 환경은 Null 종료 문자열(null-terminated string)에 의존합니다. Null 종료 문자열은 인덱스가 0부터 시작하고 NULL (#0)로 끝나는 문자 배열입니다. 이런 배열에는 길이 표시 방법이 없기 때문에 처음 나타나는 NULL 문자가 문자열의 마지막을 표시합니다. Null 종료 문자열을 사용하는 시스템과 데이터를 공유해야 하는 경우, 델파이 문법들과 SysUtils 유닛의 특수 루틴들을 사용하면(9장 "표준 루틴과 I/O" 참조) 이런 문자열을 처리할 수 있습니다.
예를 들어, Null 종료 문자열을 저장하기 위해 다음과 같은 타입 선언을 사용할 수 있습니다.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

확장 문법이 활성화되면({$X+}) 정적으로 할당된 0으로 시작하는(zero-based) 문자 배열에 문자열 상수를 대입할 수 있습니다. (동적 배열은 이러한 목적으로는 동작하지 않습니다.) 배열의 선언된 길이보다 짧은 문자열로 배열 상수를 초기화하면, 남은 문자들은 #0으로 설정됩니다. 포인터에 대한 자세한 내용은 "구조타입"절에서 "배열"을 참조하십시오.

■포인터, 배열 및 문자열 상수 사용
Null 종료 문자열을 조작하려면 포인터를 사용해야 하는 경우가 자주 있게 됩니다. (4장의 "포인터와 포인터 타입" 참조) 문자열 상수는 Char와 WideChar 값의 NULL 종료 배열을 가리키는 포인터인 PChar나 PWideChar 타입과 대입 호환이 가능합니다. 예를 들면, 다음 코드에서,

```
var P: PChar;
  ...
P := 'Hello world!';
```

P는 'Hello world!'의 Null 종료 복사본을 가진 메모리 영역을 가리킵니다. 이것은 다음과 동일합니다.

```
const TempString: array[0..12] of Char = 'Hello world!';
var P: PChar;
...
P := @TempString[0];
```

또한, StrUpper('Hello world!')와 같이 문자열 상수를 PChar 또는 PWideChar 타입의 값 파라미터 혹은 const 파라미터를 가지는 모든 함수에 전달할 수 있습니다. PChar에 대입하는 것처럼, 컴파일러는 문자열의 Null 종료 복사본을 만들고 함수에 그 복사본으로의 포인터를 전달합니다. 마지막으로, PChar 또는 PWideChar 상수를 초기화하기 위해 단독으로 혹은 구조 형태의 문자열 상수 표기를 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of Char = ('Zero', 'One', 'Two', 'Three', 'Four',
                                 'Five', 'Six', 'Seven', 'Eight', 'Nine')
```

0에서 시작하는 문자 배열은 PChar 및 PWideChar와 호환됩니다. 포인터 값 대신에 문자 배열을 사용하면 컴파일러는 배열을 배열의 첫 요소의 주소를 가진 포인터 값으로 변환합니다. 예를 들면,

```
var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

이 코드는 같은 값으로 SomeProcedure를 두 번 호출합니다.

문자 포인터는 배열처럼 인덱스로 참조할 수 있습니다. 위의 예제에서 MyPointer[0]는 H를 리턴합니다. 이런 경우, 인덱스는 포인터를 역참조(dereference)하기 전에 포인터에 더해질 오프셋을 지정합니다. (PWideChar 변수의 경우, 인덱스는 자동적으로 두 배가 됩니다.) 따라서, 문자 포인터 P에 대해, P[0]은 P^와 동일하고 배열에서 첫 문자을 지정하고, P[1]은 배열의 두 번째 문자, 이런 식입니다. P[-1]은 P[0]의 바로 왼쪽에 있는 문자를 지정합니다. 컴파일러는 이런 인덱스에서 범위 검사를 수행하지 않습니다.

StrUpper 함수는 Null 종료 문자열에 대해 포인터 인덱스를 순환적으로 사용하는 방법을 보여줍니다.

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while(I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

■ 델파이 문자열과 Null 종료 문자열의 혼합

표현식과 대입문에서 String과 Null 종료 문자열(PChar 값)을 섞어 쓸 수 있습니다. 그리고 PChar 값을 String 파라미터를 가지는 함수나 프로시저에 넘길 수 있습니다. 대입문 S := P(여기서 S는 문자열 변수이고 P는 PChar 표현식)는 Null 종료 문자열을 String에 복사합니다.

이항 연산에서 한 피연산자는 String이고 다른 피연산자는 PChar인 경우 PChar 피연산자는 긴 문자열로 변환됩니다.

PChar 값을 긴 문자열로 타입 캐스트할 수도 있습니다. 이것은 두 개의 PChar 값에 대해 문자열 연산을 수행해야 할 경우 유용합니다. 예를 들면, 다음과 같습니다.

```
S := string(P1) + string(P2);
```

또한 긴 문자열을 Null 종료 문자열로 타입 캐스트할 수도 있습니다. 다음 규칙이 적용됩니다.

- S가 String 표현식인 경우, PChar(S)는 S를 Null 종료 문자열로 타입 캐스트하고, S의 첫 문자를 가리키는 포인터를 리턴합니다. Str1과 Str2가 String인 경우, 다음과 같이 Win32 API의 MessageBox 함수를 호출할 수 있습니다. MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
- Pointer(S)를 사용하여 String을 타입이 지정되지 않은 포인터로 변환할 수도 있습니다. 하지만 S가 빈 문자열일 경우, 타입 캐스트의 결과는 nil이 됩니다.
- PChar(S)는 항상 메모리 블럭에 대한 포인터를 리턴합니다. S가 빈 문자열일 경우 결과는 #0 값을 가리키는 포인터 값입니다.

- String 변수를 포인터로 타입 캐스트하면 포인터는 변수에 새로운 값이 지정되거나 유효 범위(scope)를 벗어나기 전까지 유효합니다. 문자열 표현식을 포인터로 타입 캐스트하는 경우에는, 포인터는 타입 캐스트가 수행되는 문장 내에서만 유효합니다.
- String 표현식을 포인터로 변환하는 경우, 포인터는 보통 읽기 전용으로 간주됩니다. 포인터를 이용해서 String을 수정하려면, 다음의 조건들이 모두 충족되어야 합니다.
  - 표현식 타입 캐스트는 String 변수입니다.
  - 문자열이 비어 있지 않습니다.
  - 문자열이 고유(unique)합니다. 즉, 참조 카운트가 1입니다. 문자열이 고유하도록 하기 위해서는 SetLength, SetString, UniqueString 프로시저를 호출하면 됩니다.
  - 타입 캐스트 이후에 문자열이 수정되지 않았습니다.
  - 수정되는 문자들이 모두 문자열 내에 있습니다. 포인터에서 범위를 벗어나는 인덱스를 사용하지 않도록 주의하십시오.

구조 타입

구조 타입(structured type)의 인스턴스는 두개 이상의 값을 보유합니다. 구조 타입에는 집합(set), 배열, 레코드, 파일, 그리고 클래스, 클래스 참조, 인터페이스 타입 등이 포함됩니다. 클래스와 클래스 참조 타입에 대한 내용은 6장"클래스와 객체"를 참조하십시오. 인터페이스에 대한 내용은 11장 "인터페이스"를 참조하십시오. 서수(ordinal) 값만을 가질 수 있는 집합 타입을 제외하면, 구조 타입은 다른 구조 타입을 포함할 수 있습니다. 타입의 중첩적인 구조 단계는 무한히 중첩이 가능합니다.
기본적으로 구조 타입의 값은 더 빨리 액세스하기 위해 워드 또는 더블 워드 경계로 정렬됩니다. 구조 타입을 선언할 때 예약어 packed를 사용하면 압축 데이터 저장을 구현할 수 있습니다. 예를 들면, 다음과 같습니다.

```
type TNumbers = packed array[1..100] of Real;
```

packed를 사용하면 데이터 액세스가 느려지며, 문자 배열의 경우 타입 호환성에 영향을 줍니다. 자세한 내용은 12장"메모리 관리"를 참조하십시오.

집합

집합(set)은 같은 서수 타입을 가진 값의 컬렉션입니다. 각각의 값은 고유한 순서를 갖지 않으며 집합에 두 번 포함되는 것도 의미가 없습니다.

집합 타입의 범위는 기반(base) 타입으로 불리는 특정 서수 타입의 멱집합(power set)입니다. 즉, 집합 타입의 가능한 값들은 빈 집합을 포함한 기반 타입의 모든 서브셋입니다. 기본 타입은 256을 넘는 값을 가질 수 없으며, 기본 타입의 순서 값(ordinality)은 0에서 255사이여야 합니다. 집합은 다음의 형식을 가집니다.

```
set of baseType
```

여기서 *baseType*은 적절한 서수 타입입니다.

기반 타입에 대한 크기 제한 때문에 집합 타입은 대개 부분범위로 정의됩니다. 예를 들어, 다음의 선언은,

```
type
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

TIntSet이라는 집합 타입을 생성하는데, 값이 1부터 250사이의 정수들의 컬렉션입니다. 다음의 코드도 같은 결과를 얻습니다.

```
type TIntSet = set of 1..250;
```

이 선언을 이용하여 다음과 같은 집합을 만들 수 있습니다.

```
var Set1, Set2: TIntSet;
...
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10];
```

set of ... 문법을 변수 선언문에 직접 사용할 수도 있습니다.

```
var MySet: set of 'a'..'z';
...
MySet := ['a','b','c'];
```

집합 타입의 다른 예는 다음과 같습니다.

```
set of Byte;

set of (Club, Diamond, Heart, Spade);
set of Char;
```

in 연산자는 집합의 요소인지를 테스트합니다.

```
if 'a' in MySet then ... { 어떤 작업 } ;
```

모든 집합 타입은 빈 집합([]) 값을 가질 수 있습니다. 집합에 대한 자세한 내용은 3장의 "집합 생성자" 및 "집합 연산자"를 참조하십시오.

배열

배열은 같은 타입(기반 타입)을 가진 요소의 인덱스 가능한 컬렉션을 나타냅니다. 각 요소에는 고유한 인덱스가 있으므로, 배열은 집합과는 달리 같은 값을 두 번 이상 포함할 수 있습니다. 배열은 정적으로(static) 또는 동적으로(dynamic) 할당할 수 있습니다.

■정적 배열

정적 배열(static array)의 문법은 다음과 같습니다.

```
array[indexType1, ..., indexTypen] of baseType
```

여기서 각 *indexType*은 범위가 2GB를 넘지 않는 서수 타입입니다. *indexType*은 배열의 인덱스이기 때문에 배열이 가질 수 있는 요소의 수는 *indexType* 크기의 곱에 의해 제한됩니다. 실제 *indexType*은 보통 정수 부분범위입니다.

배열의 가장 간단한 예인 1차원 배열에서는 *indexType*이 하나만 있습니다. 다음의 예는,

```
var MyArray: array[1..100] of Char;
```

100개의 문자 값을 가지는 MyArray라는 배열 변수를 선언합니다. 이선언에서 MyArray[3]은 MyArray의 세 번째 문자를 의미합니다. 정적 배열을 생성하고 모든 요소에 값을 지정하지 않으면, 사용되지 않은 요소는 할당된 상태이지만 임의의 데이터를 포함하게 됩니다. 이것은 변수가 초기화되지 않았을 경우와 마찬가지입니다.
다차원 배열은 배열들의 배열입니다. 예를 들어 다음 코드는,

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

다음과 동일합니다.

```
type TMatrix = array[1..10, 1..50] of Real;
```

위 두가지 중 어떤 방식으로 TMatrix를 선언하든 500개의 실수 값의 배열을 나타냅니다. TMatrix 타입의 MyMatrix라는 변수가 있다면, 배열의 특정 요소를 지정하기 위해 MyMatrix[2,45]처럼 하거나 MyMatrix[2][45]처럼 할 수 있습니다. 마찬가지로, 다음 코드는,

```
packed array[Boolean,1..10,TShoeSize] of Integer;
```

다음과 동일합니다.

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

표준 함수 Low 및 High는 배열 타입 식별자와 배열 변수를 다룹니다. 이 함수들은 배열의 첫번째 인덱스 타입의 최저 값과 최고 값을 리턴합니다. 표준 함수 Length는 배열에서 첫번째 차원의 요소 수를 리턴합니다.
Char 값을 가지는 1차원의 압축된(packed) 정적 배열을 압축 문자열(packed string)이라고 합니다. 압축 문자열 타입은 문자열 타입과 호환되며, 요소 수가 같은 다른 압축 문자열 타입과도 호환됩니다. "타입 호환 및 구분"을 참조하십시오.

인덱스가 0부터 시작하는 문자 배열, 즉 array[0..x] of Char 형태의 배열 타입을 흔히 0 기반(zero based) 문자 배열이라고 합니다. 0 기반 문자 배열은 Null 종료 문자열을 저장하기 위해 사용되며, PChar 값과 호환됩니다. "Null 종료 문자열 사용"을 참조하십시오.

■동적 배열

동적 배열(dynamic array)은 고정된 크기나 길이가 없습니다. 그 대신, 동적 배열은 값을 배열에 대입하거나 SetLength 프로시저로 넘길 때 메모리가 다시 할당됩니다. 동적 배열 타입은 다음과 같은 문법으로 표시됩니다.

```
array of baseType
```

예를 들면 다음 코드는,

```
var MyFlexibleArray: array of Real;
```

실수의 1차원 동적 배열을 선언합니다. 이 선언문은 MyFlexibleArray에 메모리를 할당하지 않습니다. 배열을 메모리에 생성하려면 SetLength를 호출합니다. 예를 들어 위와 같은 선언일 경우,

```
SetLength(MyFlexibleArray, 20);
```

이 코드는 0에서 19까지의 인덱스를 가지는 20개의 실수 배열을 할당합니다. 동적 배열은 항상 정수 인덱스를 가지며, 또 항상 0에서 시작합니다.

동적 배열 변수는 내부적으로는 포인터이며, 긴 문자열과 같은 참조 카운팅 기법으로 관리됩니다. 동적 배열을 해제하려면, 배열을 참조하는 변수에 nil을 대입하거나 참조 변수를 Finalize 함수로 전달합니다. 이두 방법 모두 다른 참조가 없으면 배열을 해제합니다. 동적 배열은 참조 카운트가 0이 되면 자동으로 해제됩니다. 길이가 0인 동적 배열은 nil 값을 가집니다. 동적 배열 변수에 역참조 연산자(^)를 적용하거나 New 또는 Dispose 프로시저에 전달하지 마십시오.

X와 Y가 같은 동적 배열 타입 변수인 경우, X := Y는 X가 Y와 같은 배열을 가리키도록 합니다. 이 작업을 수행하기 전에 X에 메모리를 할당할 필요가 없습니다. 문자열이나 정적 배열과는 달리, 동적 배열은 작성되기 전에 자동으로 복사되지 않습니다. 예를 들어, 다음 코

드를 실행합니다.

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

A[0]의 값은 2입니다. (A와 B가 정적 배열이라면, A[0]은 여전히 1일것입니다.)
동적 배열 인덱스에 대한 할당(예를 들면 MyFlexibleArray[2] := 7)은 배열을 다시 할당하지 않습니다. 또, 인덱스가 범위를 벗어나더라도 컴파일 중에는 알 수가 없습니다.
반대로, 동적 배열의 독립적인 복사본을 만들려면, 전역 함수인 Copy를 사용해야 합니다.

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := Copy(A);
  B[0] := 2; { B[0] <> A[0] }
end;
```

동적 배열 변수가 비교될 때는 배열 값들이 아니라 배열의 참조가 비교됩니다. 따라서 다음의 코드가 실행되고 나면,

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;
```

A = B는 False가 리턴되지만 A[0] = B[0]는 True가 리턴됩니다.
동적 배열을 자르려면, 배열을 SetLength 또는 Copy에 넘기고 그 결과를 배열 변수에 다

시 대입합니다. 일반적으로 SetLength 프로시저가 더 빠릅니다. 예를 들어, A가 동적 배열인 경우, A := SetLength(A, 0, 20)은 A의 처음 20개의 요소를 제외한 나머지를 잘라 버립니다.
동적 배열이 할당된 후에는 이를 표준 함수 Length, High, Low로 전달할 수 있습니다. Length는 배열의 요소 수를 리턴하고, High는 배열의 최대 인덱스(즉, Length-1)를 리턴하고, Low는 0을 리턴합니다. 길이가 0인배열의 경우, High는 -1을 리턴합니다(High < Low인 예외적인 경우입니다).

Note

함수 및 프로시저 선언에서 다음 코드처럼 배열 파라미터가 지정된 인덱스 타입 없이 array of baseType으로 지정된 경우가 있습니다.

```
function CheckStrings
 (A: array of string):
 Boolean;
```

이 코드는 크기나 인덱스 또는 정적이나 동적으로 할당되었는지에 상관없이 지정된 기반 타입의 모든 배열에서 함수가 적용된다는 것을 나타냅니다. 5장의 "개방형 배열 파라미터"를 참조하십시오.

■ 다차원 동적 배열

다차원 동적 배열을 선언하려면, array of ... 문법을 반복적으로 사용합니다. 예를 들면,

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

위 코드는 2차원 문자열 배열을 선언합니다. 이 배열을 인스턴스화하려면, SetLength를 호출하면서 두개의 정수 인수를 전달합니다. 예를 들어, I와 J가 정수 변수라면,

```
SetLength(Msgs, I, J);
```

위 코드는 I x J 배열을 할당하며, Msgs[0, 0]와 같이 해당 배열의 요소를 표현합니다.
반듯한 사각형이 아닌 다차원 동적 배열을 생성할 수도 있습니다. 첫 단계는 SetLength를 호출하면서 배열의 첫 번째 n 차원에 대한 파라미터를 전달하는 것입니다. 예를 들면,

```
var Ints: array of array of Integer;
SetLength(Ints, 10);
```

위 코드는 Ints에 대해 10개의 행(row)을 할당하지만 열(column)은 할당하지 않습니다. 그 다음으로, 길이가 서로 다른 열들을 한 번에 하나씩 할당할 수 있습니다. 예를 들면,

```
SetLength(Ints[2], 5);
```

위 코드는 Ints의 세 번째 열에 5개의 정수 요소를 할당합니다. 이 시점에서, 다른 열들은

할당하지 않았더라도 Ints[2, 4] := 6과 같이 이미 할당된 세 번째 열에는 값을 지정할 수 있습니다.
다음 예제는 동적 배열과 SysUtils 유닛에서 선언된 IntToStr 함수를 사용하여 문자열의 삼각형 매트릭스를 만듭니다.

■ 배열 타입과 대입문

배열은 서로 같은 타입을 가진 경우에만 대입 호환성이 있습니다(assignment-compatible). 델파이 언어는 타입에 대해 이름 동등성(name-equivalence)을 사용하기 때문에 다음 코드는 컴파일되지 않습니다.

```
var
  A: array of array of string;
  I, J: Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
  begin
    SetLength(A[I], I);
    for J := Low(A[I]) to High(A[I]) do
      A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
  end;
end;
```

대입이 가능하도록 하려면, 다음과 같이 변수를 선언합니다.

```
var
Int1: array[1..10] of Integer;
Int2: array[1..10] of Integer;
...
Int1 := Int2;
```

또는 다음과 같이 선언합니다.

```
var Int1, Int2: array[1..10] of Integer;
```

```
type IntArray = array[1..10] of Integer;
var
Int1: IntArray;
Int2: IntArray;
```

레코드

레코드(일부 언어들의 구조체(structure)와 비슷)는 다양한 요소들의 집합을 표현합니다. 각 요소는 필드라고 하며, 레코드 타입의 선언에서는 각 필드의 이름과 타입을 지정합니다. 레코드 타입 선언의 문법은 다음과 같습니다.

```
type recordTypeName = record
  fieldList1: type1;
  ...
  fieldListn: typen;
end
```

여기서 *recordTypeName*은 유효한 식별자이고, 각 *type*은 타입을 의미하며, 각 *fieldList*는 유효한 식별자 하나 혹은 쉼표로 구분된 식별자 리스트입니다. 마지막의 세미콜론은 옵션입니다.
예를 들어, 다음의 선언은 TDateRec라는 레코드 타입을 만듭니다.

```
type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
      Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;
```

각 TDateRec은 세 개의 필드를 가지고 있습니다. 이들 필드는 정수인 Year 필드, 열거 타입인 Month 필드, 그리고 1에서 31까지의 정수인 Day 필드입니다. Year, Month, Day 식별자는 TDateRec에 대한 필드 지정자 이며, 이 식별자들은 변수처럼 동작합니다. 그러나 TDateRec 타입 선언으로는 Year, Month 및 Day 필드에 메모리를 할당하지 않습니다. 메모리는 다음과 같이 레코드를 인스턴스화하는 경우에 할당됩니다.

```
var Record1, Record2: TDateRec;
```

이 Variant 타입 선언은 두 개의 TDateRec 인스턴스, Record1과 Record2를 만듭니다.
레코드의 필드에 액세스하려면 필드 지정자 를 레코드의 이름으로 한정하면 됩니니다.

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

또는 with 문을 사용하여 액세스할 수 있습니다.

```
with Record1 do
begin
  Year := 1904;
  Month := Jun;
  Day := 16;
end;
```

이제 Record1 필드의 값을 Record2에 복사할 수 있습니다.

```
Record2 := Record1;
```

필드 지정자의 유효 범위(scope)는 선언된 레코드에 제한되기 때문에 필드 지정자 와 다른 변수간의 이름 충돌에 대해서는 걱정할 필요가 없습니다.
레코드 타입을 정의하는 대신 변수 선언에 직접 record ... 문법을 사용할 수 있습니다.

```
var S: record
  Name: string;
  Age: Integer;
end;
```

그러나 이와 같이 선언하면 레코드를 사용하는 목적, 즉 비슷한 변수 그룹의 반복적 코딩을 줄이려는 목적에 위배됩니다. 더욱이 이러한 방식으로 따로 선언된 레코드들은 구조가 같은 경우라도 대입 호환성이 없습니다.

■ 레코드의 가변 부분

레코드 타입은 case 문과 비슷한 형식의 가변 부분(variant part)을 가질 수 있습니다. 가변 부분은 레코드 선언의 마지막 필드에 와야 합니다.

가변 부분이 있는 레코드 타입을 선언하려면 다음과 같은 문법을 사용합니다.

```
type recordTypeName = record
  fieldList1: type1;
  ...
  fieldListn: typen;
  case tag: ordinalType of
    constantList1: (variant1);
    ...
    constantListn: (variantn);
end
```

선언의 처음에서 예약어 case의 앞까지는 다른 표준 레코드 타입과 동일합니다. 선언의 나머지 부분(case에서 옵션인 마지막 세미콜론까지)을 가변 부분이라고 합니다. 가변 부분에서,

- *tag*는 옵션이며 유효한 식별자입니다. *tag*를 생략하는 경우 그 다음의 콜론(:)도 같이 생략해야 합니다.
- *ordinalType*은 서수(ordinal) 타입을 나타냅니다.
- 각 *constantList*는 *ordinalType* 타입의 값을 표시하는 상수이거나 상수를 쉼표로 구분한 리스트입니다. *constantList*들에서 같은 값이 두 번 이상 나타날 수 없습니다.
- 각각의 *variant*는 레코드 타입의 메인 부분에서의 *fieldList: type* 문법과 비슷한 선언들이 쉼표로 구분된 리스트입니다. 즉, 가변 부분은 다음 형식을 가집니다.

```
fieldList1: type1;
...
fieldListn: typem;
```

여기서 각 *fieldList*는 유효한 식별자 또는 쉼표로 구분된 식별자의 리스트이며, 각 *type*은 타입을 나타내며, 마지막 세미콜론은 옵션입니다. *type*은 긴 문자열, 동적 배열, Variant 타입 또는 인터페이스일 수 없으며, 또한 이들 타입들을 포함하는 구조 타입일 수 없습니다. 하지만 이들은 이들 타입에 대한 포인터는 가능합니다.

가변 부분이 있는 레코드는 문법적으로는 복잡하지만 의미상으로는 매우 간단합니다. 레코

드의 가변 부분에 있는 가변 필드들은 메모리에서 같은 공간을 공유합니다. 언제든지 모든 가변 필드를 읽거나 쓸 수 있습니다. 하지만 한 가변 필드에 쓰고 나서 또 다른 가변 필드에 쓰려고 하면, 데이터를 겹쳐쓰게 될 수 있습니다. *tag*가 있을 경우 레코드의 비 가변 부분에 있는 또 하나의 필드(*ordinalType* 타입의 필드)처럼 동작합니다.
가변 부분은 두 가지 목적이 있습니다. 첫째는 여러 종류의 데이터를 위한 필드들을 가지는 레코드 타입을 작성하려고 하지만, 단일 레코드 인스턴스에서 모든 필드들을 사용할 필요는 없다는 것을 알고 있는 경우입니다. 예를 들면, 다음과 같습니다.

```
type
  TEmployee = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Salaried: Boolean of
    True: (AnnualSalary: Currency);
    False: (HourlyWage: Currency);
  end;
```

여기서의 개념은, 모든 직원들은 연봉이나 시간당 임금을 받지만, 둘 다 받지는 않습니다. 그래서 TEmployee 인스턴스를 만들 때 두필드 모두에 충분한 메모리를 할당할 필요가 없습니다. 이런 경우, 두 가변 필드 사이의 유일한 차이점은 필드 이름 뿐이지만, 다른 타입들인 것처럼 쉽습니다. 이제 조금 더 복잡한 예제를 봅시다.

```
type
  TPerson = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Citizen: Boolean of
    True: (Birthplace: string[40]);
    False: (Country: string[20];
            EntryPort: string[20];
            EntryDate, ExitDate: TDate);
  end;

type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
      Triangle: (Side1, Side2, Angle: Real);
```

```
      Circle: (Radius: Real);
      Ellipse, Other: ();
    end;
```

컴파일러는 각 레코드 인스턴스에 대해 가장 큰 가변의 모든 필드들을 수용할만큼 충분한 메모리를 할당합니다. 옵션인 *Tag*와 *constantList* (위마지막 예제의 Rectangle, Triangle과 같은)는 컴파일러가 필드들을 관리하는 방식에 아무런 역할도 하지 않습니다. 이들은 단지 프로그래머의 편의를 위해서만 존재합니다.

가변 부분을 사용하는 두 번째 이유는 컴파일러가 타입 캐스트를 허용하지 않는 경우라도 동일한 데이터를 여러 다른 타입에 속하는 것처럼 다룰 수 있기 때문입니다. 예를 들어, 한 가변의 첫 필드로서 64비트의 실수를 사용하고, 다른 가변의 첫 필드에는 32비트의 정수를 사용하는 경우, 값을 실수 필드에 할당한 다음 이를 정수 필드의 값인 것처럼 정수 파라미터를 필요로 하는 함수에 값을 전달하여 이 값의 처음 32비트를 다시 읽어낼 수 있습니다.

레코드 (클래스 기능)

전통적인 레코드 타입에 더하여, 델파이 언어는 더 복잡하고 '클래스와 비슷한' 레코드 타입을 지원합니다. 레코드에는 필드 외에도 속성과 메소드(생성자도 포함), 클래스 속성, 클래스 필드, 중첩된 타입을 포함할 수 있습니다. 이들 주제에 대한 더 자세한 정보를 찾아보려면 6장 "클래스와 객체"를 참고하십시오. 아래의 예제는 예제 레코드 타입 정의입니다.

```
type
  TMyRecord = record
    type
      TInnerColorType = Integer;
    var
      Red: Integer;
    class var
      Blue: Integer;
    procedure printRed();
    constructor Create(val: Integer);
    property RedProperty: TInnerColorType read Red write Red;
    class property BlueProp: TInnerColorType read Blue write Blue;
  end;
constructor TMyRecord.Create(val: Integer);
begin
```

```
  Red := val;
end;

procedure TMyRecord.printRed;
begin
  writeln('Red: ', Red);
end;
```

레코드는 이제 클래스의 많은 기능들을 가질 수 있게 되었지만, 클래스와 레코드 사이에는 몇가지 중요한 차이점들이 있습니다.

- 레코드는 상속을 지원하지 않습니다.
- 레코드는 가변 부분을 포함할 수 있지만 클래스는 안됩니다.
- 레코드는 값 타입이기 때문에 대입을 통해 복사가 가능하고, 값으로 전달되며, 전역으로 선언되거나 New 및 Dispose 함수로 명시적으로 할당되지 않은 한 스택(stack)에 할당됩니다. 클래스는 참조 타입이므로 대입으로 복사할 수 없고, 참조로 전달되며, 힙(heap)에 할당됩니다.
- 레코드는 연산자 오버로딩을 지원하지만, 클래스는 연산자 오버로딩을 지원하지 않습니다.
- 레코드는 인자가 없는 기본 생성자를 통해 자동으로 생성됩니다. 반면 클래스는 명시적으로 생성되어야만 합니다. 레코드는 인자가 없는 기본 생성자가 있으므로, 사용자가 정의하는 레코드 생성자는 반드시 하나 이상의 파라미터를 가져야만 합니다.
- 레코드 타입은 파괴자를 가질 수 없습니다.
- 가상 메소드 (virtual, dynamic, message 키워드로 선언된 메소드)는 레코드 타입에서 사용할 수 없습니다.
- 클래스와 달리 레코드 타입은 인터페이스를 구현할 수 없습니다.

파일 타입

파일 타입(File type)은 Win32에서만 사용 가능하며, 같은 타입 요소들의 연속입니다. 표준 I/O 루틴은 줄(line) 로 구성되는 문자들을 가진 파일을 나타내는 이미 정의된 타입 TextFile이나 Text를 사용합니다. 파일 입출력에 대한 자세한 내용은 9장 "표준 루틴과 I/O"를 참조하십시오.

파일 타입을 선언하려면 다음 문법을 사용합니다.

```
type fileTypeName = file of type
```

여기서 *fileTypeName*은 유효한 식별자이고, *type*은 고정 크기를 가진 타입입니다. 암시적이든 또는 명시적이든 포인터 타입은 허용되지 않습니다. 따라서 파일 타입에는 동적 배열, 긴 문자열, 클래스, 객체, 포인터, Variant 타입, 다른 파일, 그리고 이런 타입들을 포함하는 구조 타입이 포함될 수 없습니다. 예를 들면,

```
type
  PhoneEntry = record
    FirstName, LastName: string[20];
    PhoneNumber: string[15];
    Listed: Boolean;
  end;
PhoneList = file of PhoneEntry;
```

위 코드는 이름과 전화 번호를 기록하기 위해 파일 타입을 선언합니다.
또한 file of ... 문법을 변수 선언에 직접 사용할 수도 있습니다. 예를 들면, 다음과 같습니다.

```
var List1: file of PhoneEntry;
```

다음과 같이 file 단어만 사용되었을 때는 타입 미지정 파일(untyped file)을 나타냅니다.

```
var DataFile: file;
```

자세한 내용은 9장의 "타입 미지정 파일"을 참조하십시오.
파일은 배열이나 레코드에서는 허용되지 않습니다.

포인터와 포인터 타입

포인터(pointer)는 메모리 주소를 나타내는 변수입니다. 포인터가 다른 변수의 주소를 가질 때는 이를 메모리의 해당 변수의 위치 또는 이 위치에 저장된 데이터를 가리킨다고 합니다. 배열이나 다른 구조 타입의 경우, 포인터는 그 구조의 첫 번째 요소의 주소를 가리킵니다.
포인터는 타입이 지정되는데, 이것은 포인터가 가리키는 주소에 저장된 데이터의 종류를 알려주기 위해서입니다. 범용인 Pointer 타입은 어떤 종류의 데이터도 가리킬 수 있는 포인터를 나타내며, 그 외의 포인터 타입들은 특정 타입의 데이터만 참조할 수 있습니다. 포인터는

4바이트의 메모리를 사용합니다.

포인터의 개요

포인터의 동작 원리를 이해하기 위해 다음 예제를 살펴봅시다.

```
1 var
2   X, Y: Integer; // X와 Y는 정수 변수입니다
3   P: ^Integer;   // P는 정수를 가리키는 포인터입니다
4 begin
5   X := 17;       // X에 값을 대입합니다
6   P := @X;       // X의 주소를 P에 대입합니다
7   Y := P^;       // P를 역참조하고 그 결과를 Y에 대입합니다
8 end;
```

2 행에서 정수 타입의 변수 X와 Y를 선언합니다. 3 행에서는 P를 정수 값에 대한 포인터로 선언합니다. 이것은 P가 X나 Y의 위치를 가리킬 수 있다는 것을 의미합니다. 5 행에서 X에 값을 대입하고, 6 행에서는 P에 X의 주소를 대입합니다(@X로 표시). 마지막으로 7번 행에서는 P가 가리키는 위치에 있는 값(^P로 표시)을 Y에 지정합니다. 이 코드를 실행하고 나면 X와 Y는 같은 값 17을 갖게 됩니다.

@ 연산자는 위 예제에서는 변수의 주소를 얻기 위해 사용되었지만, 함수와 프로시저에서도 사용할 수 있습니다. 자세한 내용은 3장의 "@ 연산자" 및 4장의 "문장 및 표현식의 프로시저 타입"을 참조하십시오.

^ 기호는 두 가지의 목적을 갖고 있으며, 위의 예제에서 이 두 가지 모두 보였습니다. ^ 기호가 타입 식별자 앞에 오는 경우,

```
^typeName
```

*typeName*이라는 타입의 변수에 대한 포인터를 나타냅니다. ^ 기호가 포인터 변수 뒤에 오는 경우,

```
Pointer^
```

이 경우는 포인터를 역참조(dereference)합니다. 즉, 포인터가 가리키는 메모리 주소에 저장된 값을 리턴합니다.

위 예제는 단순한 대입문으로 할 수 있는 작업, 즉 한 변수의 값을 다른 변수로 복사하는 작업을 복잡하게 둘러서 한 것처럼 보일 것입니다. 하지만 포인터는 여러 가지 이유로 유용합니다. 첫째, 포인터는 종종 코드에 명시적으로 드러나지 않고 사용되므로, 포인터를 이해하면 델파이 언어를 이해하는데 도움이 됩니다. 동적으로 할당된 큰 메모리 블럭을 필요로 하는 모든 데이터 타입은 포인터를 이용합니다. 예를 들어, 긴 문자열 변수는 클래스 변수와 마찬가지로 암시적인 포인터입니다. 또, 일부 고급 프로그래밍 기법을 적용하려면 포인터를 사용해야 합니다.

마지막으로, 때때로 포인터는 델파이의 엄격한 데이터 타입 지정을 우회할 수 있는 유일한 방법입니다. 범용인 Pointer로 변수를 참조하고, Pointer를 특정 타입으로 타입 캐스트하고, 이를 다시 역참조함으로써, 어떠한 변수에 저장된 데이터라도 원하는 타입인 것처럼 처리할 수 있습니다. 예를 들어, 다음 코드는 실수 변수에 저장된 데이터를 정수 변수에 대입합니다.

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ...
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

물론, 실수와 정수는 서로 다른 형식으로 저장됩니다. 이 대입문에서는 R에서 I로 바이너리 데이터를 데이터 변환 없이 단순히 복사합니다.

포인터에 @ 연산의 결과를 대입하는 방법 외에도, 포인터에 값을 지정하기 위해 몇 가지 표준 루틴을 사용할 수 있습니다. New와 GetMem 프로시저는 메모리 주소를 기존의 포인터에 대입하며, Addr 함수와 Ptr 함수는 지정된 주소나 변수로의 포인터를 리턴합니다.

역참조된 포인터는 한정될 수 있으며, P1^.Data^ 표현식에서처럼 한정자(qualifier)로 사용할 수 있습니다.

예약어 nil은 모든 포인터에 대입할 수 있는 특별한 상수입니다. Nil을 포인터에 대입하면 그 포인터는 아무것도 참조하지 않습니다.

포인터 타입(Pointer type)

다음 문법을 사용하여 모든 타입에 대한 포인터를 선언할 수 있습니다.

```
type pointerTypeName = ^type
```

레코드나 다른 데이터 타입을 정의할 때 해당 타입에 대한 포인터도 함께 정의해두면 유용할 수 있습니다. 이렇게 하면 대규모의 메모리 블럭을 복사하지 않고도 그 타입의 인스턴스를 간단히 처리할 수 있습니다.

Note
포인터가 가리킬 타입을 선언하기 전에 그 포인터를 먼저 선언할 수 있습니다.

표준 포인터 타입은 여러 가지 목적으로 사용됩니다. 가장 다목적으로 사용되는 것은 Pointer로서, 어떠한 종류의 데이터라도 가리킬 수 있습니다. 하지만 Pointer 변수는 역참조가 불가능하며, Pointer 변수 뒤에 ^ 기호를 사용하면 컴파일 에러가 발생합니다. 포인터 변수가 참조하는 데이터를 액세스하려면, 먼저 이를 다른 포인터 타입으로 변환한 다음 역참조합니다.

■ 문자 포인터

기본 타입 PAnsiChar와 PWideChar는 각각 AnsiChar 및 WideChar 변수에 대한 포인터를 나타냅니다. 일반적인 PChar는 Char, 즉 WideChar에 대한 포인터를 나타냅니다. 이들 문자 포인터는 Null 종료 문자열을 처리하기 위해 사용됩니다.(4장의 "Null 종료 문자열" 참조)

■ 타입 체크 포인터

$T 컴파일러 지시자는 @ 연산자에 의해 생성되는 포인터 값의 타입을 제어합니다. 이 지시자는 다음과 같은 형식을 따릅니다.

{$T+} 혹은 {$T-}

{$T-} 상태에서는 @ 연산자의 결과 타입은 타입 미지정 포인터로서, 다른 모든 포인터 타입들과 호환 가능합니다. {$T+}상태에서 @ 연산자로 변수를 참조하면 결과의 타입은 해당 변수의 타입을 가리키는 포인터와만 호환됩니다.

■ 기타 표준 포인터 타입

System과 SysUtils 유닛은 일반적으로 사용되는 많은 표준 포인터 타입을 선언합니다.

표 5.6 시스템과 SysUtils에서 선언된 선택된 포인터 타입

포인터 타입	가리키는 변수 타입
PAnsiString, PString	AnsiString
PByteArray	TbyteArray (SysUtils에 선언) 동적 할당된 메모리를 타입 캐스트하여 배열로 액세스하기 위해 사용.
PCurrency, PDouble, PExtended, PSingle	Currency, Double, Extended, Single
PInteger	Integer
POleVariant	OleVariant
PShortString	ShortString PString 타입을 사용하는 예전 코드를 포팅할 때 유용함.
PTextBuf	TTextBuf (SysUtils에 선언) TtextRec 파일 레코드의 내부 버퍼 타입.
PVarRec	TVarRec (System에 선언)
PVariant	Variant
PWideString	WideString
PWordArray	TWordArray (SysUtils에 선언). 동적 할당된 메모리를 타입 캐스트하여 2바이트 배열로 액세스하기 위해 사용.

프로시저 타입

프로시저 타입(procedural type)은 프로시저와 함수를 변수에 대입하거나 다른 프로시저나 함수에 전달할 수 있는 값으로 다룰 수 있게 해줍니다. 예를 들어, 정수 파라미터 두 개를 받고 정수를 리턴하는 Calc라는 함수를 정의한다고 가정해봅니다.

```
function Calc(X, Y: Integer): Integer;
```

이 Calc 함수를 변수 F에 대입할 수 있습니다.

```
var F: function(X, Y: Integer): Integer;
F := Calc;
```

어떤 프로시저나 함수이든 그 헤더에서 procedure 또는 function 뒤의 식별자를 제거하면

프로시저 타입의 이름이 됩니다. 이런 타입 이름을 위의 예제처럼 변수 선언에 직접 사용하거나 다음과 같이 새로운 타입을 선언하는 데 사용할 수 있습니다.

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
var
  F: TIntegerFunction;     { F는 파라미터가 없고 정수를 리턴하는 함수 }
  Proc: TProcedure;        { Proc은 파라미터가 없는 프로시저 }
  SP: TStrProc;            { SP는 문자열 파라미터를 받는 프로시저 }
  M: TMathFunc;            { M은 Double (real) 파라미터를 받고 Double을 리턴하는 함수 }
  procedure FuncProc(P: TintegerFunction);
                           { FuncProc는 유일한 파라미터로 파라미터가 없는 정수 값 함수를 갖는 프로시저 }
```

위의 변수들은 모두 프로시저 포인터입니다. 즉, 프로시저나 함수의 주소를 가리키는 포인터입니다. 인스턴스 객체의 메소드를 참조하려는 경우에는(6장 "클래스와 객체" 참조), 프로시저 타입 이름에 of object를 추가해야 합니다. 예를 들면, 다음과 같습니다.

```
type
  TMethod = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

이들 타입은 메소드 포인터를 나타냅니다. 메소드 포인터는 실제로는 두개의 포인터로 이루어지는데, 첫 번째 포인터는 메소드의 주소를 저장하고, 두 번째 포인터는 메소드가 속한 객체에 대한 참조를 저장합니다. 다음과 같이 선언했을 때,

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ...
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent;
```

다음과 같은 대입문을 만들 수 있습니다.

```
OnClick := MainForm.ButtonClick;
```

다음의 조건을 충족하는 경우 두 프로시저 타입이 서로 호환됩니다.

- 같은 호출 규칙(calling convention)을 사용
- 리턴 값이 같거나 리턴 값이 없음
- 파라미터의 수가 같고 해당 위치의 파라미터 타입이 동일 (파라미터의 이름은 무관)

프로시저 포인터 타입은 항상 메소드 포인터 타입과 호환되지 않습니다. nil 값은 모든 프로시저 타입에 지정될 수 있습니다.
중첩된 프로시저 및 함수(다른 루틴 내에 선언된 루틴)는 프로시저 값으로 사용할 수 없으며, 이미 정의된 프로시저나 함수도 프로시저 값으로 사용할 수 없습니다. 프로시저 값으로 Length처럼 이미 정의된 루틴을 사용하려면 이에 대한 래퍼를 만들면 됩니다.

```
function FLength(S: string): Integer;
begin
  Result := Length(S);
end;
```

문장 및 표현식의 프로시저 타입

프로시저 변수가 대입문의 왼쪽에 있으면, 컴파일러는 오른쪽에 프로시저 값이 있을 것으로 예상합니다. 대입을 하면 왼쪽의 변수는 오른쪽에 표시된 함수나 프로시저에 대한 포인터 값을 가지게 됩니다. 하지만 다른 문법에서 프로시저 변수를 사용하면 그 변수가 참조하는 프로시저나 함수가 호출됩니다. 프로시저 변수가 호출될 때는 파라미터도 전달할 수 있습니다.

```
var
  F: function(X: Integer): Integer;
  I:Integer;
function SomeFunction(X: Integer): Integer;
...
F := SomeFunction;    // F에 SomeFunction을 대입
I := F(4);            // 함수를 호출하고 그 결과를 I에 대입
```

대입문에서는 왼쪽의 변수 타입에 따라 오른쪽의 프로시저 또는 메소드의 해석이 결정됩니다. 예를 들면, 다음과 같습니다.

```
var
  F, G: function: Integer;
  I: Integer;
function SomeFunction: Integer;
...
F := SomeFunction;    // F에 SomeFunction을 대입
G := F;               // F를 G에 복사
I := G;               // 함수를 호출하고 그 결과를  I에 대입
```

첫 번째 대입문은 프로시저 값을 F에 대입합니다. 두 번째 대입문은 값을 또 다른 변수로 복사합니다. 세 번째 대입문은 참조되는 함수를 호출하고 결과를 I에 대입합니다. I는 프로시저 변수가 아닌 정수 변수이기 때문에, 마지막 대입문에서는 함수(정수를 리턴하는)가 호출됩니다.
경우에 따라서 프로시저 변수가 어떻게 해석될지가 명확하지 않을 수도 있습니다. 다음 문법을 봅시다.

```
if F = MyFunction then ...;
```

이 경우에 F로 인해 함수가 호출됩니다. 컴파일러는 F가 가리키는 함수를 호출한 다음 MyFunction 함수를 호출하고, 결과를 비교합니다. 표현식 내에 프로시저 변수가 나타날 때마다 그 변수가 참조하는 프로시저나 함수가 호출된다는 것이 규칙입니다. F가 프로시저(값을 리턴하지 않음)를 참조하거나 F가 파라미터가 있는 함수를 참조하는 경우에는 위의 문법은 컴파일 에러가 발생합니다. F의 프로시저 값을 MyFunction과 비교하려면 다음과 같은 코드를 사용합니다.

```
if @F = @MyFunction then ...;
```

@F는 F를 타입 미지정 포인터 변수로 변환하여 주소를 가지게 되고, @MyFunction은 MyFunction의 주소를 리턴합니다.
프로시저 변수 안에 저장된 주소가 아닌 프로시저 변수 자체의 메모리 주소를 얻으려면 @@를 사용합니다. 예를 들어, @@F는 F의 주소를 리턴합니다.

@ 연산자를 사용하면 타입 미지정 포인터 값을 프로시저 변수에 대입할 수도 있습니다. 예를 들면,

```
var StrComp: function(Str1, Str2: PChar): Integer;
...
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

위 코드는 GetProcAddress 함수를 호출하고 StrComp 변수가 GetProcAddress 함수의 결과를 가리키도록 합니다.
모든 프로시저 변수는 nil 값을 가질 수 있는데, 이것은 아무것도 가리키지 않는다는 의미입니다. 하지만 nil 값의 프로시저 변수를 호출하려고 시도하면 에러가 발생합니다. 프로시저 변수가 대입되어 있는지 확인하려면 표준 함수 Assigned를 사용합니다.

```
if Assigned(OnClick) then OnClick(X);
```

Variant 타입

경우에 따라서는, 타입이 달라질 수 있거나 컴파일 시에 결정할 수 없는 데이터를 처리해야 하는 경우가 있습니다. 이런 경우, 해결 방법들 중 하나는 Variant 타입의 변수나 파라미터를 사용하는 것입니다. Variant 타입이란 런타임 시 변경될 수 있는 값을 나타냅니다. variant 타입은 더 유연하지만, 일반적인 변수보다 많은 메모리를 사용하며, 정적인 타입보다 연산도 느립니다. 뿐만 아니라, 일반 변수를 사용할 경우에는 컴파일 시에 발견할 수 있는 실수가 variant 타입에서는 암시적인 연산으로 인해 런타임 에러로 나타날 수 있습니다. 사용자 정의 variant 타입을 만들 수도 있습니다.
variant 타입은 레코드, 집합, 정적 배열, 파일, 클래스, 클래스 참조, 포인터를 제외한 모든 타입의 값을 저장할 수 있습니다. 다시 말해 variant 타입은 구조 타입과 포인터를 제외한 모든 데이터를 저장할 수 있다는 뜻입니다. variant 타입은 인터페이스를 저장할 수 있으며, 그 메소드와 속성을 variant 타입을 통해 액세스할 수 있습니다. (11장 "인터페이스" 참조). variant 타입은 동적 배열을 저장할 수 있으며, variant 배열이라고 하는 특수한 종류의 정적 배열을 수 있습니다("variant 타입 배열" 참조). variant 타입은 표현식 및 대입문

에서 다른 variant 타입, 정수, 실수, 문자열, 부울 값과 혼합될 수 있습니다. 컴파일러는 자동으로 타입 캐스트를 수행합니다.

문자열을 데이터로 가진 variant 타입은 인덱싱할 수 없습니다. 즉, V가 문자열 값을 갖는 variant 타입인 경우, V[1] 문법에서 런타임 에러가 발생합니다

독자적인 값들을 저장하기 위해 variant 타입을 확장하여 사용자 정의 variant 타입을 정의할 수 있습니다. 예를 들어, 인덱싱이 가능하거나 특정 클래스 참조, 레코드 타입, 정적 배열을 저장하는 variant 문자열 타입을 정의할 수 있습니다. 사용자 정의 variant 타입은 TCustomVariantType 클래스로부터 파생하여 정의할 수 있습니다.

Note

이 기능을 포함한 거의 모든 variant 타입의 기능들은 Variants 유닛에서 구현되어 있습니다.

variant 타입은 16바이트의 메모리를 사용하며, 코드에서 지정하는 타입의 타입 코드와 값, 또는 값에 대한 포인터로 구성됩니다. 모든 variant 타입은 생성 시 특수 값 Unassigned로 초기화됩니다. 특수 값 Null은 알려지지 않거나 누락된 데이터를 나타냅니다.

표준 함수 VarType은 variant 타입 코드를 리턴합니다. varTypeMask 상수는 VarType의 리턴 값에서 코드를 추출하기 위해 사용되는 비트 마스크입니다. 예를 들면 다음과 같습니다.

```
VarType(V) and varTypeMask = varDouble
```

이 문법은 V에 Double 타입이나 Double 타입 배열이 있는 경우 True를 리턴합니다. (마스크는 variant 타입에 배열이 있는지를 나타내는 첫 번째 비트를 단순히 숨깁니다.) System 유닛에서 정의된 TVarData 레코드 타입은 variant 타입을 타입 캐스트하기 위해 사용될 수 있으며 내부 표현 방식에 액세스할 수 있게 해줍니다. 코드 리스트는 VarType에 대한 온라인 헬프를 참조하십시오. 새로운 타입 코드는 오브젝트 파스칼 다음 버전에 추가될 수도 있습니다.

variant 타입의 변환

variant 타입은 모든 정수 타입, 실수 타입, 문자열 타입, 문자 타입, 부울 타입과 대입 호환이 됩니다. 표현식은 명시적으로 variant 타입으로 타입 캐스트될 수 있습니다. 그리고 VarAsType과 VarCast 표준 루틴은 variant의 내부 표현 방식을 변경하기 위해 사용될 수 있습니다. 다음 코드는 variant 타입이 사용되는 방식과 variant 타입이 다른 타입과 혼합될 때 수행되는 자동 변환의 일부를 보여줍니다.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string
begin
  V1 := 1;                    { 정수 값 }
  V2 := 1234.5678;            { 실수 값 }
  V3 := 'Hello world!';       { 문자열 값 }
  V4 := '1000';               { 문자열 값 }
  V5 := V1 + V2 + V4;         { 실수 값 2235.5678 }
  I := V1;                    { I = 1 (정수 값) }
  D := V2;                    { D = 1234.5678 (실수 값) }
  S := V3;                    { S = 'Hello world!' (π문자열 값 }
  I := V4;                    { I = 1000 (정수 값) }
  S := V5;                    { S = '2235.5678' (π문자열 값 }
end;
```

컴파일러는 다음의 규칙에 따라 변환을 수행합니다.

표 4.7 variant 타입 변환 규칙

대상소스	정수	실수	문자열	부울
정수	정수 포맷으로변환	실수로 변환	문자 표현으로 변환	0이면 False 리턴, 아니면 True
실수	가장 가까운 정수로 반올림	실수 포맷으로 변환	지역 설정에 따라 문자열 표현으로 변환	0이면 False 리턴, 아니면 True
문자열	정수로 변환, 필요할 경우 자름. 문자열이 숫자가 아닐 경우 예외 발생	지역 설정에 따라 실수로 변환. 문자열이 숫자가 아닐 경우 예외 발생	문자열/ 문자 포맷을 변환	문자열이 'false'(대소문자 무시) 이거나 0이 되는 숫자 문자열이면 False 리턴. 문자열이 'true'이거나 0이 아닌 숫자 문자열이면 True 리턴. 그 외에는 예외 발생.
문자	위 문자열과 같음.	위 문자열과 같음.	위 문자열과 같음.	위 문자열과 같음.
부울	False = 0, True: 모든 비트가 1로 설정됨. (정수이면 –1, Byte이면 255 등)	False = 0, True = 1	기본적으로는 False = 'False', True = 'True'. 전역 변수Boolean ToStringRule에 따름	False = False, True = True
Unassigned	0 리턴	0 리턴	빈 문자열 리턴	False 리턴
Null	전역변수 NullStrictConvert에 따름 (기본은 예외 발생)	전역변수 NullStrictConvert에 따름 (기본은 예외 발생)	전역변수 NullStrictConvert와 NullAsStringValue에 따름 (기본은 예외 발생)	전역변수 NullStrictConvert에 따름(기본은 예외 발생)

범위를 벗어난 대입의 경우, 대상 변수가 가질 수 있는 범위에서 최고값을 얻게 되는 경우가 많습니다. 잘못된 variant 조작, 대입문이나 타입 캐스트를 하게 되면 EVariantError 예외나 EVariantError로부터 파생된 예외가 발생합니다.

System 유닛에서 선언된 TDateTime 실수 타입에는 특수한 변환 규칙이 적용됩니다. TDateTime이 다른 타입으로 변환될 때는 보통 Double 타입으로 취급됩니다. 정수, 실수 또는 부울이 TDateTime으로 변환될 때는 먼저 Double로 변환된 다음 날짜/시간 값으로 읽습니다. 문자열이 TDateTime으로 변환될 때는 지역 설정에 따라 날짜/시간 값으로 해석합니다. Unassigned 값이 TDateTime으로 변환될 때는 실수나 정수 0처럼 취급합니다. Null 값을 TDateTime으로 변환하면 예외가 발생합니다.

variant 타입이 COM 인터페이스를 참조하는 경우, 이를 변환하려고 하면 객체의 default 속성을 읽어서 해당 값을 요청된 타입으로 변환합니다. 객체에 default 속성이 없는 경우에는 예외가 발생합니다.

표현식 내의 variant 타입

^, is, in 연산자를 제외한 모든 연산자는 variant 타입 피연산자를 받습니다. 항상 부울 값을 리턴하는 비교 연산을 제외하면 variant 타입에 대한 모든 연산은 variant 타입 값을 리턴합니다. variant 타입을 다른 일반적인 타입의 값과 함께 쓰는 표현식에서는 일반 타입의 값이 variant 타입으로 변환됩니다.

이와 달리, 비교 연산에서는 Null variant에 대한 연산은 Null variant 값을 리턴합니다. 예를 들어 다음 코드는,

```
V := Null + 3;
```

V에 Null variant를 대입하게 됩니다. 기본적으로, 비교에서는 Null variant를 다른 모든 값들보다도 작은 특이한 값으로 취급합니다. 예를 들어,

```
if Null > -3 then ... else ...;
```

위의 예에서, if 문의 else 부분이 실행될 것입니다. NullEqualityRule과 NullMagnitudeRule 글로벌 변수를 설정하면 이런 동작 방식을 바꿀 수 있습니다.

variant 타입 배열

일반적인 정적 배열은 variant 타입으로 변환할 수 없습니다. 대신, 표준 함수 VarArrayCreate 또는 VarArrayOf를 호출하여 variant 타입 배열을 생성할 수 있습니다. 예를 들면,

```
V: Variant
...
V := VarArrayCreate([0,9], varInteger);
```

위 코드는 정수의 variant 타입 배열(길이 10)을 생성하고 이를 variant 타입 변수 V에 대입합니다. 이 배열은 V[0], V[1] 등을 사용하여 각 인덱스 요소들을 액세스할 수 있습니다. 하지만 variant 타입 배열 요소를 var 파라미터로서 전달하는 것은 불가능합니다. variant 타입 배열은 항상 정수로 인덱싱 됩니다.

VarArrayCreate 호출의 두 번째 파라미터는 배열의 기본 타입에 대한 타입 코드입니다. 이들 코드의 리스트는 VarType에 대한 온라인 헬프를 참조하십시오. 절대로 varString 코드를 VarArrayCreate로 전달하지 마십시오. 문자열의 variant 타입 배열을 생성하려면 varOleStr을 사용합니다.

variant 타입은 크기, 차원 및 기본 타입이 다른 variant 타입 배열을 가질 수 있습니다. variant 타입 배열의 요소는 ShortString 및 AnsiString을 제외한 variant에서 허용되는 모든 타입일 수 있으며, 배열의 기본 타입이 Variant인 경우는 요소가 이질적일 수도 있습니다. variant 타입 배열의 크기를 조정하려면 VarArrayRedim 함수를 사용합니다. variant 타입 배열에서 작동되는 다른 표준 루틴에는 VarArrayDimCount, VarArrayLowBound, VarArrayHighBound, VarArrayRef, VarArrayLock 및 VarArrayUnlock이 있습니다.

variant 타입 배열을 포함하는 variant이 다른 variant에 대입되거나 값 파라미터로서 전달되는 경우에는 전체 배열이 복사됩니다. 이 연산은 메모리를 비효율적으로 사용하기 때문에 꼭 필요하지 않다면 이 연산을 수행하지 마십시오.

Note

사용자 정의 variant 타입의 variant 타입 배열은 지원되지 않습니다. 사용자 정의 variant 타입의 인스턴스는 VarVariant variant 타입 배열에 추가될 수 있기 때문입니다.

OleVariant

Variant와 OleVariant의 주요 차이점은 Variant는 어떻게 처리해야 할지 현재 애플리케이션만이 알고 있는 데이터 타입만을 포함할 수 있다는 것입니다. OleVariant는 OLE 오토메이션과 호환되도록 정의된 데이터 타입만 포함할 수 있습니다. 다시 말해, 프로그램 사이나 네트워크를 통해 상대가 데이터 처리 방법을 알고 있는지에 대해 걱정할 필요 없이 전달될 수 있는 데이터 타입입니다.

사용자 정의 데이터(델파이 문자열이나 새로운 사용자 정의 variant 타입 등)를 가진 Variant를 OleVariant에 대입하면, 런타임 라이브러리는 Variant를 OleVariant 표준 데이터 타입(델파이 문자열을 OLE BSTR 문자열로 변환하는 등) 중 하나로 변환하려고 시도합니다. 예를 들어, AnsiString을 포함하는 variant 변수가 OleVariant에 대입되면, AnsiString은 WideString이 됩니다. 그리고 Variant를 OleVariant 함수 파라미터로 전달하는 경우에도 동일합니다.

타입 호환 및 동등성

어떤 표현식에서 어떤 연산이 수행될 수 있는지를 이해하려면, 타입과 값들 사이의 몇가지 호환성을 구별할 수 있어야 합니다. 여기에는 다음과 같은 세가지가 있습니다.

- 타입 동등성
- 타입 호환성
- 대입 호환성

타입 동등성

하나의 타입 식별자를 다른 타입 식별자를 사용하여 한정자 없이 선언하는 경우에는 같은 타입을 의미하게 됩니다. 따라서, 다음과 같이 선언한 경우,

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

T1, T2, T3, T4 및 Integer는 모두 같은 타입을 표시합니다. 새로 선언하는 타입이 이전 타입과 구분되도록 하려면 선언에서 type을 반복해야 합니다. 다음의 예에서,

```
type TMyInteger = type Integer;
```

Integer와는 다른 새로운 TMyInteger라는 타입을 생성합니다.

타입 이름과 같은 기능을 하는 코드는 선언될 때마다 다른 타입이 됩니다. 그러므로 다음 선언은,

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

두 개의 서로 다른 타입인 TS1과 TS2를 생성합니다. 이와 비슷하게 다음의 변수 선언,

```
var
  S1: string[10];
  S2: string[10];
```

이 문법은 두 개의 서로 다른 타입의 변수를 생성합니다. 같은 타입의 변수들을 생성하려면 다음을 사용합니다.

```
var S1, S2: string[10];
```

또는

```
type MyString = string[10];
var
  S1: MyString;
  S2: MyString;
```

타입 호환성

모든 타입은 동일한 자체 타입과 호환됩니다. 두개의 서로 다른 타입은 다음 조건들 중 최소 하나 이상을 만족할 경우에만 호환됩니다.

- 양쪽 모두 실수 타입인 경우
- 양쪽 모두 정수 타입인 경우
- 한 타입이 다른 타입의 부분범위 타입인 경우

- 양쪽 모두 같은 타입의 부분범위 타입인 경우
- 양쪽 모두 호환 가능한 기반(base) 타입을 가진 집합 타입인 경우
- 양쪽 모두 문자 수가 같은 packed 문자열 타입인 경우
- 하나는 문자열 타입이고 다른 하나는 문자열, packed 문자열 또는 Char 타입인 경우
- 하나는 Variant이고 다른 하나는 정수, 실수, 문자열, 문자 또는 부울 타입인 경우
- 양쪽 모두 클래스, 클래스 참조 또는 인터페이스 타입이고, 하나가 다른 하나로부터 파생된 경우
- 하나가 PChar 또는 PWideChar이고 다른 하나는 array[0..n] of Char 형식의 인덱스가 0부터 시작하는 문자 배열인 경우
- 하나는 Pointer(타입이 지정되지 않은 포인터)이고 다른 하나는 임의의 포인터 타입인 경우
- 양쪽 모두 같은 타입에 대한 (타입이 지정된) 포인터이고 {$T+} 컴파일러 지시어가 켜졌을 경우
- 양쪽 모두 같은 결과 타입, 같은 파라미터 수, 해당 위치의 파라미터들 사이에 타입 동등성이 있는 프로시저 타입인 경우

대입 호환성

대입 호환성(assignment-compatibility)은 대칭적인 관계가 아닙니다. T2 타입의 표현식은 T1 타입의 변수에 대입될 수 있으려면, 표현식의 값이 T1의 범위 내에 있고 다음 조건 중 하나 이상이 만족되어야 합니다.

- T1과 T2가 같은 타입이고, 어떤 수준에서든 파일 타입이거나 파일을 포함하는 구조 타입이 아닌 경우
- T1과 T2가 호환 가능한 서수(ordinal) 타입인 경우
- T1과 T2가 모두 실수 타입인 경우
- T1은 실수 타입이고 T2는 정수 타입인 경우
- T1이 PChar, PWideChar이거나 임의의 문자열 타입이고 표현식이 문자열 상수인 경우
- T1과 T2가 모두 문자열 타입인 경우
- T1이 문자열 타입이고 T2는 Char 또는 압축 문자열 타입인 경우
- T1이 긴 문자열이고 T2는 PChar이거나 PWideChar인 경우
- T1과 T2가 호환 가능한 압축 문자열 타입인 경우
- T1과 T2가 호환 가능한 집합 타입인 경우
- T1과 T2가 호환 가능한 포인터 타입인 경우
- T1과 T2가 모두 클래스, 클래스 참조 또는 인터페이스 타입이고, T2가 T1에서 파생된 경우
- T1이 인터페이스 타입이고 T2가 T1을 구현하는 클래스 타입인 경우

- T1이 PChar 또는 PWideChar이고 T2가 array[0..n] of Char 형식의 인덱스가 0부터 시작하는 문자 배열인 경우
- T1과 T2가 호환 가능한 프로시저 타입인 경우 (함수나 프로시저 식별자는 특정 대입문에서 프로시저 타입의 표현식으로 취급됩니다. 4장의 "문장 및 표현식의 프로시저 타입"을 참조하십시오.)
- T1은 variant 타입이고 T2는 정수, 실수, 문자열, 문자, 부울, 인터페이스 타입 또는 OleVariant인 경우
- T1은 OleVariant이고 T2는 정수, 실수, 문자열, 문자, 부울, 인터페이스 타입 또는 Variant인 경우
- T1은 정수 타입, 실수 타입, 문자열 타입, 문자 타입 또는 부울 타입이고 T2는 Variant 혹은 OleVariant 타입인 경우
- T1이 IUnknown 또는 IDispatch 인터페이스 타입이고 T2는 Variant 혹은 OleVariant 타입인 경우 (Variant의 타입 코드는 T1이 IUnknown인 경우에는 varEmpty, varUnknown 또는 varDispatch여야 하며, T1이 IDispatch인 경우에는 varEmpty 또는 varDispatch여야 합니다.)

타입 선언

타입 선언은 타입을 표시하는 식별자를 지정합니다. 타입 선언 문법은 다음과 같습니다.

```
type newTypeName = type
```

여기서 *newTypeName*은 유효한 식별자입니다. 예를 들어, 다음과 같이 타입을 선언한 경우,

```
type TMyString = string;
```

다음과 같이 변수를 선언을 할 수 있습니다.

```
var S: TMyString;
```

타입 식별자의 유효 범위에 타입 선언 자체는 포함되지 않습니다(포인터 타입들은 제외). 따라서 예를 들어 그 자체를 재귀적으로 사용하는 레코드 타입은 정의할 수 없습니다.
기존의 타입과 같은 타입을 선언할 때는 컴파일러가 새로운 타입 식별자를 기존의 식별자에 대한 별명(alias)으로 간주합니다. 따라서, 다음과 같이 선언했다고 가정하면,

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

X와 Y는 같은 타입입니다. 런타임 시 TValue와 Real은 구별이 되지 않습니다. 이것은 일반적으로 별로 쓸모가 없지만, 새 타입을 정의하는 목적이 런타임 타입 정보(RTTI)를 활용하려는 것이라면, 예를 들어 특정 타입의 속성에 프로퍼티 에디터를 연결시키려고 한다면 "다른 이름"과 "다른 타입" 사이의 구별이 중요해집니다. 이런 경우에는 다음 문법을 사용합니다.

```
type newTypeName = type type
```

예를 들면, 다음 코드는,

```
type TValue = type Real;
```

컴파일러에게 TValue라는 새로운 구별된 타입을 만들도록 합니다.
var 파라미터의 경우에는 형식 타입과 실제 타입은 반드시 일치해야 합니다. 다음의 예를 보십시오.

```
type
  TMyType = type Integer
  procedure p(var t: TMyType);
  begin
  end;

procedure x;
var
  m: TMyType;
  i: Integer;
begin
  p(m); // 동작함
  p(i); // 에러! 형식 타입과 실제 타입은 반드시 일치해야 함.
end;
```

Note
이것은 var 파라미터에만 해당하며, const 파라미터와 값 파라미터에는 해당하지 않습니다.

변수

변수는 런타임 시에 그 값이 변경될 수 있는 식별자입니다. 달리 말하면, 변수는 메모리의 위치에 대한 이름입니다. 이 이름을 사용하면 해당 메모리 위치를 읽거나 쓸 수 있습니다. 변수는 데이터를 위한 컨테이너와 비슷하면서 타입이 지정되기 때문에 컴파일러에게 변수가 갖고 있는 데이터를 해석하는 방법을 알려주게 됩니다.

변수의 선언

변수 선언문의 기본 문법은 다음과 같습니다.

```
var identifierList: type;
```

여기서 *identifierList*는 쉼표로 구분된 유효한 식별자의 리스트이고, type은 임의의 유효한 타입입니다. 예를 들면,

```
var I: Integer;
```

위 코드는 정수 타입 변수 I를 선언하며,

```
var X, Y: Real;
```

위 코드는 두 변수 X와 Y를 실수 타입으로 선언합니다.
연속적인 변수 선언문들에서는 예약어 var를 여러 번 사용하지 않아도 됩니다.

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

프로시저나 함수 내에서 선언된 변수는 지역(local) 변수라고 하고, 그 외의 변수들은 전역(global) 변수라고 합니다. 전역 변수는 선언과 동시에 초기화할 수 있으며, 다음과 같은 문법을 따릅니다.

```
var identifier: type = constantExpression;
```

여기서 *constantExpression*은 *type* 타입의 값을 나타내는 임의의 상수 표현식입니다. 상수 표현식에 대한 자세한 내용은 4장의 "상수 표현식"을 참조하십시오. 따라서, 다음의 선언은

```
var I: Integer = 7;
```

다음 선언 및 문장과 동일합니다.

```
var I: Integer;
  ...
  I := 7;
```

지역 변수는 선언에서 초기화할 수 없습니다. 복수의 변수를 선언하는 선언문(`var X, Y, Z: Real;`처럼)에서는 초기화를 할 수 없으며, variant 및 파일 타입의 변수 선언도 초기화를 할 수 없습니다.

전역 변수를 명시적으로 초기화하지 않으면 컴파일러가 전역변수들을 0으로 초기화합니다. 객체 인스턴스 데이터(필드)도 0으로 초기화됩니다. 지역 변수는 변수에 값이 대입되기 전까지는 값이 정의되지 않습니다.

변수를 선언하면 메모리가 할당되며 변수가 더 이상 사용되지 않으면 자동적으로 해제됩니다. 특히, 지역 변수는 이들이 선언된 함수나 프로시저로부터 빠져나가기 전까지만 존재합니다. 변수와 메모리 관리에 대한 자세한 내용은 12장 "메모리 관리"를 참조하십시오.

■ 절대 주소

다른 변수와 같은 주소에 위치하는 새로운 변수를 생성할 수 있습니다. 이렇게 하려면, absolute 지시어를 새 변수의 선언문 타입 이름 다음에 추가하고 그 뒤에는 이전에 선언된 기존 변수의 이름을 덧붙여야 합니다. 예를 들면,

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

위 코드는 StrLen 변수가 Str과 같은 주소에서 시작한다는 것을 지정합니다. 짧은 문자열의 첫 바이트에는 문자열의 길이가 있기 때문에 StrLen의 값은 Str의 길이가 될 것입니다. absolute 선언에서 변수를 초기화할 수 없으며, absolute 지시어를 다른 지시어와 조합할 수도 없습니다.

■ 동적 변수

GetMem이나 New 프로시저를 호출하여 동적 변수를 생성할 수 있습니다. 이러한 변수는 힙(heap)에 할당되며 자동으로 관리되지 않습니다. 변수를 생성한 이후에는 변수의 메모리를 해제하는 것은 궁극적으로 개발자의 책임입니다. GetMem으로 생성된 변수는 FreeMem으로 해제하고, New로 생성된 변수는 Dispose로 해제합니다. 동적 변수를 다루는 다른 표준 루틴으로는 ReallocMem, AllocMem, Initialize, Finalize, StrAlloc 및 StrDispose가 있습니다.

UnicodeString, AnsiString, WideString, 동적 배열, variant, 인터페이스도 힙에 할당된 동적 변수이지만, 이들 메모리는 자동으로 관리됩니다.

■ 쓰레드 지역 변수

쓰레드 지역 변수(thread local variable) 또는 쓰레드 변수는 멀티 쓰레드 애플리케이션에서 사용됩니다. 쓰레드 지역 변수는 전역 변수와 비슷하지만, 각 쓰레드를 실행하면 자체 복사본을 갖게 되며 이 복사본은 다른 쓰레드에서는 액세스할 수 없습니다. 쓰레드 지역 변수는 var 대신 threadvar로 선언됩니다. 예를 들면 다음과 같습니다.

```
threadvar X: Integer;
```

쓰레드 변수 선언문은 다음과 같은 제약이 있습니다.

- 프로시저나 함수 내에 올 수 없습니다.
- 초기화를 포함할 수 없습니다.
- absolute 지시어를 지정할 수 없습니다.

일반적으로 컴파일러에 의해 관리되는 동적 변수(UnicodeString, AnsiString, WideString, 동적 배열, variant, 인터페이스)도 threadvar로 선언될 수는 있지만, 컴파일러는 각 실행 쓰레드마다 생성된 힙 할당 메모리를 자동으로 해제하지 않습니다. 이들 데이

터 타입을 쓰레드 변수로 사용하는 경우에는 사용자가 직접 이들 메모리를 해제해야 합니다. 예를 들면,

Note
이런 코드는 권장되지 않습니다.

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
...
S := ''; // S에 할당된 메모리를 해제
```

variant 타입의 메모리를 해제하려면 Unassigned로 설정하면 되고, 인터페이스나 동적 배열의 메모리를 해제하려면 nil로 설정하면 됩니다.

선언된 상수

몇가지 서로 다른 언어 구조들이 "상수"라고 불립니다. 상수에는 17과 같은 숫자 상수(숫자, 뉴머럴 (numeral)이라고도 함)도 있고, 'Hello world!' 와 같은 문자열 상수(문자 문자열 또는 문자열 리터럴이라고도 함)도 있습니다. 숫자 및 문자열 상수에 대한 내용은 3장 "문법 요소"를 참조하십시오.
모든 열거 타입은 해당 타입의 값들을 나타내는 상수들을 정의합니다. 이미 정의된 상수들로 True, False, nil과 같은 것들이 있습니다. 변수와 비슷하게 선언문에 의해 개별적으로 생성된 상수도 있습니다.
선언된 상수는 식별자를 가지고 선언된 상수를 말합니다. 선언된 상수에는 순수 상수와 타입 지정 상수가 있습니다. 이 두 종류의 상수는 외견상으로 비슷하지만, 서로 다른 규칙이 적용되며 다른 목적으로 사용됩니다.

순수 상수

순수 상수(true constant)는 선언된 식별자로서 값이 변경될 수 없습니다. 예를 들면,

```
const MaxValue = 237;
```

정수 237을 리턴하는 MaxValue라는 상수를 선언합니다. 순수 상수 선언의 문법은 다음과 같습니다.

```
const identifier = constantExpression
```

여기서 *identifier*는 임의의 유효한 식별자이고 *constantExpression*은 프로그램의 실행 전에 컴파일러가 계산할 수 있는 표현식입니다. 자세한 내용은 4장의 "상수 표현식"을 참조하십시오.

*constantExpression*이 서수(ordinal) 값을 리턴하는 경우에는 값 타입 캐스트를 사용하여 선언된 상수의 타입을 지정할 수 있습니다. 예를 들면,

```
const MyNumber = Int64(17);
```

위 코드는 정수 17을 리턴하는 Int64 타입의 MyNumber 상수를 선언합니다. 이런 경우를 제외하면, 선언된 상수의 타입은 *constantExpression*의 타입을 따르게 됩니다.

- *constantExpression*이 문자열인 경우, 선언된 상수는 모든 문자열 타입과 호환됩니다. 문자열의 길이가 1인 경우에는 모든 문자 타입과도 호환됩니다.
- *constantExpression*이 실수인 경우, 상수의 타입은 Extended입니다. 정수인 경우에는 다음의 표에 따라 타입이 지정됩니다.

표 4.8 정수 상수의 타입

상수의 범위 (16진수)	상수의 범위 (10진수)	타입
-$8000000000000000..-$80000001	-2^63..-2147483649	Int64
-$80000000..-$8001	-2147483648..-32769	Integer
-$8000..-$81	-32768..-129	Smallint
-$80..-1	-128..-1	Shortint
0..$7F	0..127	0..127
$80..$FF	128..255	Byte
$0100..$7FFF	256..32767	0..32767
$8000..$FFFF	32768..65535	Word
$10000..$7FFFFFFF	65536..2147483647	0..2147483647
$80000000..$FFFFFFFF	2147483648..4294967295	Cardinal
$100000000..$7FFFFFFFFFFFFFFF	4294967296..2^63?1	Int64

다음은 상수 선언의 예입니다.

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

■ 상수 표현식

상수 표현식은 프로그램의 실행 전에 컴파일러가 계산할 수 있는 표현식입니다. 상수 표현식에는 숫자 상수, 문자열, 순수 상수, 열거 타입의 값들, 특수 상수인 True, False, nil, 그리고 이들 상수들과 연산자, 타입 캐스트, 집합 생성자로만 구성된 표현식이 있습니다. 상수 표현식에는 변수, 포인터, 함수 호출을 포함될 수 없습니다. 다만 다음의 함수 호출은 예외적으로 상수 표현식에서 사용될 수 있습니다.

Abs	High	Low	Pred	Succ
Chr	Length	Odd	Round	Swap
Hi	Lo	Ord	Sizeof	Trune

상수 표현식을 이렇게 정의하는 것은 델파이의 문법 표준의 일부 위치에서 사용됩니다. 상수 표현식은 전역 변수의 초기화, 부분범위 타입의 정의, 열거 타입의 값에 순서값 지정, 기본 파라미터(default parameter) 값 지정, case 문, 순수 상수와 타입 지정 상수 등에 사용됩니다. 다음은 상수 표현식의 예입니다.

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'CodeGear' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

■ 리소스 문자열

리소스 문자열(resource string)은 리소스로 저장되며, 실행 파일이나 프로그램을 다시 컴파일하지 않아도 되도록 라이브러리로 링크됩니다.

const가 resourcestring으로 대체된 것을 제외하면 리소스 문자열의 선언은 다른 순수 상수들과 유사합니다. = 기호 오른쪽의 표현식은 상수 표현식이어야 하고 문자열 값을 리턴해야 합니다. 예를 들면, 다음과 같습니다.

```
resourcestring
  CreateError = 'Cannot create file %s';
  OpenError = 'Cannot open file %s';
  LineTooLong = 'Line too long';
  ProductName = 'CodeGear Rocks';
  SomeResourceString = SomeTrueConstant;
```

타입 지정 상수

타입 지정 상수(typed constant)는 순수 상수와는 달리 배열, 레코드, 프로시저, 포인터 타입의 값을 가질 수 있습니다. 타입 지정 상수는 상수 표현식에서 발생할 수 없습니다.

다음과 같이 타입 지정 상수를 선언합니다.

```
const identifier: type = value
```

여기서 *identifier*는 임의의 유효한 식별자이고, *type*은 파일 타입과 variant 타입을 제외한 임의의 타입이며, *value*는 *type* 타입의 표현식입니다. 예를 들면, 다음과 같습니다.

```
const Max: Integer = 100;
```

대부분의 경우 value는 상수 표현식이어야 합니다. 하지만 type이 배열, 레코드, 프로시저, 포인터 타입인 경우에는 특수 규칙이 적용됩니다.

노트: 기본 설정인 {$J-} 컴파일러 상태에서 타입 지정 상수는 새 값이 지정될 수 없습니다. 타입 지정 상수는 사실 읽기 전용 변수입니다. 하지만 {$J+} 컴파일러 지시문을 사용하면 타입 지정 상수에 새로운 값을 대입할 수 있습니다. 이들은 초기화된 변수처럼 행동합니다.

■배열 상수

배열 상수를 선언하려면, 선언의 뒤에 각 배열 요소 값들을 쉼표로 구분하고 괄호로 둘러쌉니다. 이 각각의 값들은 상수 표현식으로 표기해야 합니다. 예를 들면,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

위 코드는 문자 배열을 갖는 Digits라는 타입 지정 상수를 선언합니다.
0 기반(zero based) 문자 배열은 종종 Null 종료 문자열을 나타냅니다. 따라서문자 배열을 초기화하기 위해 문자열 상수를 사용할 수 있습니다. 그러므로 위의 선언은 다음과 같이 더 간편하게 나타낼 수 있습니다.

```
const Digits: array[0..9] of Char = '0123456789';
```

다차원 배열 상수를 정의하려면 각 차원의 값들에 괄호를 사용하고 이를 쉼표로 구분합니다. 예를 들면,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

위의 코드는 다음과 같은 Maze 배열을 만듭니다.

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

배열 상수에는 어느 수준에서든 파일 타입 값이 포함될 수 없습니다.

■ 레코드 상수

레코드 상수를 선언하려면 선언 뒤에 fieldName: value와 같이 세미콜론으로 구분하여 각

필드의 값을 지정하고 괄호로 감쌉니다. 값은 상수 표현식으로 나타내야 합니다. 필드는 레코드 타입 선언에 나타난 순서대로 나열되어야 합니다. 그리고 태그 필드가 있는 경우, 태그 필드는 지정된 값을 가져야 합니다. 레코드에 variant 부분이 있으면 태그 필드에서 선택된 variant만 값이 지정될 수 있습니다.
예를 들면, 다음과 같습니다.

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X:0.0; Y: 0.0);
  Line: TVector = ((X:-3.1; Y: 1.5), (X:5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

레코드 상수는 어느 수준에서든 파일 타입 값을 포함할 수 없습니다.

■ 프로시저 상수
프로시저 상수를 선언하려면, 상수의 선언된 타입과 호환 가능한 함수나 프로시저의 이름을 지정합니다. 예를 들면,

```
function Calc(X, Y: Integer): Integer;
begin
...
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

위와 같이 선언을 한 후에는 다음과 같이 MyFunction 프로시저 상수로 함수를 호출할 수 있습니다.

```
I := MyFunction(5, 7)
```

또한 프로시저 상수에 nil 값을 지정할 수도 있습니다.

■ 포인터 상수

포인터 상수를 선언할 때는 적어도 컴파일 시 상대 주소로라도 해석할 수 있는 값으로 초기화해야 합니다. 그러려면 세가지 방법이 있는데, @연산자를 사용하거나, nil을 사용하거나, 문자열 리터럴(상수가 PChar나 PWideChar 타입인 경우)을 사용하는 방법이 있습니다. 예를 들어, I가 Integer 타입 전역 변수인 경우, 다음과 같이 상수를 선언할 수 있습니다.

```
const PI: ^Integer = @I;
```

컴파일러는 이를 해석할 수 있는 것은 전역 변수가 코드 세그먼트에 포함되기 때문입니다. 함수와 전역 상수의 경우도 마찬가지입니다.

```
const PF: Pointer = @MyFunction;
```

문자열 리터럴은 전역 상수로서 할당되기 때문에 문자열 리터럴로 PChar 상수를 초기화할 수 있습니다.

```
const WarningStr: PChar = 'Warning!';
```