오브젝트 파스칼의 기초

(Fundamentals of Object Pascal)

델파이에서 사용된 오브젝트 파스칼은 표준 파스칼에 비해 많은 진보가 있은 언어이다. 파스칼은 최초에 top-down 디자인과 구조적 프로그래밍을 가르치기 위해 개발된 언어이다. 그렇기 때문에 가장 많은 수의 대학에서 프로그래밍 언어의 표준으로 이를 이용해 강의를 하곤 했다.

그러다가 볼랜드가 터보 파스칼을 발표하게 되는데, 이름 그대로 표준 파스칼에 IBM PC 에 적절한 각종 유닛과 라이브러리를 제공했으며 동시에 언어 자체의 능력도 향상시킨 진짜 '터보 파스칼' 이었다. 사실 이해하기 어려운 C 코드에 비해 직관적이면서도 깨끗한 터보 파스칼은 당시에 상당한 반향을 일으키며 터보 C 와 함께 IBM PC 시장의 양대 언어로 자리잡았다.

그 이후, 윈도우 프로그래밍 환경이 나타나면서 우리나라에서는 볼랜드 C++, 비주얼 C++ 그리고 비주얼 베이직에 밀려서 터보 파스칼의 사용자 수가 많이 줄어들었다. 그렇지만, 터보 파스칼의 인기는 유럽에서는 선풍적이었는데 인터넷이나 뉴스 그룹을 통해 알아보면 아직도 무수한 터보 파스칼에 대한 정보를 얻을 수 있다.

파스칼이 객체지향형 프로그래밍 언어의 기능을 추가하게 된 것은 터보 파스칼 5.5 버전으로, 이때부터 조금씩 진보된 환경에의 변화를 시도하다가 급기야 델파이 1.0 이 발표되면서 명실상부한 객체지향형 파스칼로서 세상에 모습을 드러내게 된다.

이번 장에서는 델파이의 언어적 토대를 이루고 있는 오브젝트 파스칼의 기본적인 문법에 대해서 알아보도록 한다. 무릇 기초가 튼튼해야 큰 건물을 지을 수 있는 법이니, 파스칼에 대한 이해의 정도가 깊을수록 좋은 델파이 프로그래머가 된다는 것은 자명한 일이다.

변수 (Variables)

파스칼에서는 변수를 사용하기 전에는 반드시 선언을 해 주어야 한다. 또한, 변수를 선언할 때에는 데이터 형을 반드시 지정해야 한다. 변수는 값은 프로그램의 수행 동안에 변경시킬 수 있으며 다양한 값을 가질 수 있다. 변수는 다음과 같이 선언한다.

```
var
  Number: Integer;
  YesNo: Boolean;
```

var 키워드는 변수를 선언하기 위해 사용한다. 위치에 따라 함수나 프로시저의 코드 시작

부분에서 지역적으로 사용되는 변수를 선언할 수도 있고, 유닛의 전역 변수를 선언할 수도 있다. 일단 변수를 선언하고 나면, 그 변수의 데이터 형이 지원하는 값을 대입할 수 있다. 간단한 대입문을 예를 들면 다음과 같다.

```
Number := 7;
YesNo := True;
```

각 문장의 끝에 ';'을 사용하는 것에 주의한다. 파스칼은 라인이 바뀌어도 문장이 종료된 것으로 취급하지 않고 ';'이 나타나야 한 문장의 끝으로 생각한다.

- 절대 주소 (Absolute addresses)

특정 메모리 주소에 있는 변수를 선언할 때에는 absolute 키워드를 사용하여 다음과 같이 선언한다.

```
var
  CrtMode: Byte absolute $0040;
```

이 방법은 평상시에는 잘 사용되지 않지만 디바이스 드라이버 등을 제작하는 등의 저수준 프로그래밍에 유용하다. 이미 존재하는 변수와 같은 주소에 새로운 변수를 생성시키고자 할 경우에는 주소를 직접 쓰지 않고, 다음과 같이 변수의 이름을 사용하여 주소를 대신하게 할 수 있다.

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

StrLen 변수는 Str 변수와 같은 주소에서 시작된다. 여기서 Str 은 32 자 길이의 ShortString 형이다. ShortString 데이터 형의 첫번째 바이트에는 문자열의 길이가 저장되므로 StrLen 변수의 값은 곧 문자열의 길이를 나타내게 된다.

- 동적 변수 (Dynamic variables)

변수를 동적으로 생성할 때에는 GetMem 또는 New 프로시저를 이용해야 한다. 이들을 이용해서 변수를 생성하게 되면 힙에 메모리가 할당되고, 자동으로 관리가 되지 않으므로 개발자가 책임지고 이들을 다 쓰고 나서 제거해 주어야 한다.

GetMem 을 통해 생성된 변수는 FreeMem 프로시저로 해제하고, New 로 생성한 변수는 Dispose 프로시저로 해제한다. 참고로 이런 동적 변수를 다루기 위해 제공되는 표준 루틴으로는 ReallocMem, Initialize, StrAlloc, StrDispose 등이 있다. 자세한 내용은 도움말을 참고하기 바란다.
델파이 4 에서 새롭게 제공되는 동적 배열에 대해서는 이 장의 후반부에서 자세히 언급하겠지만, 힙에 할당되는 동적 변수의 일종이다. 또한 긴 문자열도 마찬가지이다. 그렇지만 이들은 자동으로 관리된다는 점이 다르다.

상수 (Constants)

상수는 프로그램에 사용하는 어떤 값에 대한 이름을 붙이는 것이다. 상수는 선언할 때에 const 라는 키워드를 사용하며, 선언 예는 다음과 같다.

```
const
  Pi = 3.14159;
```

이렇게 상수를 선언하는 목적은 숫자를 기억하기 쉽게 만들어주며, 다른 사람이 코드를 관리하는 데에도 도움을 준다.

참고로 델파이 3 부터는 리소스 문자열 상수를 지원한다. 다음과 같이 선언하면 되는데, 여기에서 정의된 문자열 상수는 프로그램의 리소스안에 저장된다.

```
resourcestring
  Name = 'JiHoon Jeong';
```

기본적인 데이터 형

파스칼에서 기본적인 데이터 형은 크게 나누어 서수형(ordinal type), 실수형(real type), 문자열형(string type)의 3 가지가 있다. 그 밖에 OLE 자동화를 위한 variant 형과 subrange, enumerate, set 등의 사용자 정의 데이터 형이 제공된다. 이러한 데이터 형을 선언할 때에는 기본적으로 type 키워드를 처음에 적어준다.

- 서수형 (Ordinal type)

서수형은 순서가 있는 데이터 형으로 이해하면 된다. 즉, 이 데이터 형의 값이 있으면 이 중에서 어느 쪽이 큰지 비교할 수도 있고, 그 값이 특정 값의 앞에 있는지 뒤에 있는지도

알 수 있으며, 최소값과 최대값을 알아낼 수도 있다.
서수형 중에서 가장 흔히 사용하는 것은 Integer, Boolean, Char 의 3 가지 데이터 형이다. 이 중에서도 정수형인 Integer 형은 다음과 같이 세분할 수 있다.

메모리 크기	Signed	Unsigned
8 비트	ShortInt	Byte
16 비트	SmallInt	Word
32 비트	LongInt	LongWord
16/32 비트 (1.0/2.0 이후 버전)	Integer	Cardinal
64 비트	Int64	

이 중에서 32 비트 unsigned 정수형인 LongWord, 64 비트 signed 정수형인 Int64 는 델파이 4 에서 새롭게 제공되는 데이터 형이다.
Boolean 데이터 형도 윈도우 API 함수에 사용하기 위해 ByteBool, WordBool, LongBool 등을 사용하기도 한다. 델파이 3 버전 부터는 ByteBool, WordBool, LongBool 데이터 형의 True 값을 –1, False 값을 0 으로 가리키게 하였다. 이것은 비주얼 베이직과 OLE 자동화의 데이터 형과의 호환을 위한 것이다. Boolean 형은 True 가 1, False 가 0 을 가리킨다.
문자형은 Char 에도 크게 나누어 ANSIChar, WideChar 의 2 가지 데이터 형이 있다. 전통적으로 사용한 문자형은 8 비트의 ANSI 문자 세트이며, 특별한 언급이 없는 한 Char 문자형은 ANSIChar 를 가리킨다. WideChar 는 유니코드 문자를 지원하기 위한 것으로 16 비트 문자 세트이다. OLE 자동화와 유니코드를 지원하기 위해서 사용한다.
이런 서수형에는 공통적으로 다음과 같은 루틴을 이용할 수 있다. 이들은 매우 유용하게 사용되는 것들이므로 익혀두는 것이 좋을 것이다.

루 틴	설 명	루 틴	설 명
Dec	파라미터로서 전달되는 변수를 하나씩, 또는 특정 값만큼 감소	Inc	파라미터로서 전달되는 변수를 하나씩 또는 특정 값만큼씩 증가
Pred	주어진 데이터 형에 해당되는 변수 앞의 값을 리턴한다.	Succ	주어진 데이터 형에 해당되는 변수 뒤의 값을 리턴한다.
Low	파라미터로서 전달되는 서수형의 범위 내의 가장 낮은 값을 리턴한다.	High	파라미터로서 전달되는 서수형의 범위 내에서 가장 높은 값을 리턴한다.
Ord	주어진 데이터 형의 값세트 안에 있는 인자들의 순서를 나타내는 수를 리턴함	Odd	변수가 홀수이면 True 를 반환한다.

- 실수형 (Real type)

실수 데이터 형은 소수 부분을 갖는 숫자 데이터를 저장할 수 있다. 정수형과 마찬가지로 실수형 역시 다음과 같이 여러 가지로 세분된다.

메모리 크기	데이터 형	범 위
32 비트	Single	1.5E-45 ~ 3.4E38
64 비트	Real/Double	5.0E-324 ~ 1.7E308
64 비트	Comp	1.0 ~ 9.2E18
64 비트	Currency	0.0001 ~ 9.2E14
80 비트	Extended	3.4E-4932 ~ 1.1E4932

델파이 3 까지는 Real 데이터 형이 48 비트 였으나, 델파이 4.0 에서는 Real 데이터 형이 Double 과 같은 64 비트로 변경되었다.
Comp 데이터 형은 실 수가 아니라 매우 큰 정수형이다. 그렇지만 이 데이터 형은 실수형과 같은 방식으로 실행된다. Currency 데이터 형은 큰 숫자들을 저장할 때 매우 높은 정확도를 가지고 있으며, 금액을 나타내는 데이터베이스 형식과 호환성이 있는 이점이 있다.

- 문자열형 (String type)

델파이에서는 문자열을 여러가지 방법으로 다룰 수 있다. 다음 표는 델파이에서 사용할 수 있는 4 가지 문자열 형식을 보여주고 있다.

데이터 형	길 이	구성 문자	Null 종료
ShortString	255	ANSIChar	No
AnsiString	3GB	ANSIChar	Yes
String	255 ~ 3GB	ANSIChar	Yes or No
WideString	1.5GB	WideChar	Yes

String 데이터 형의 경우 기본적으로는 AnsiString 과 같지만, 컴파일러 옵션에 따라서는 ShortString 으로 간주될 수도 있다. 즉, {$H-} 옵션을 주면 ShortString 으로 사용된다.
AnsiString 은 Null 에 의해 문자열이 종료되기 때문에 그 길이를 매우 동적으로 할당할 수 있다. 좀더 긴 문자열을 변수에 대입할 경우 델파이에서는 문자열을 저장하기 위해 메모리를 다시 할당하게 된다. Null 종료가 장점이 되는 이유로는 Win32 API 와 같은 대부분의 시스템 루틴의 호출에서는 C 와의 호환성을 위해 Null 종료 문자열을 사용하게 된다. Null 종료 문자열을 사용하게 되면 PChar 형으로 타입 캐스팅하여 직접 C 의 문자열 형과 호환되게 할 수 있다. 델파이의 긴 문자열 들은 참조 계수(refernce counting) 기법에 기초하

고 있다. 다시 말해 메모리의 같은 문자열을 몇 개의 문자열 변수가 참조하고 있는 지를 계속 추적하여, 문자열이 더 이상 쓰이지 않을 때 (참조 계수가 0) 메모리를 비우게 된다.

- WideString

WideString 데이터 형은 동적으로 할당되는 16 비트 유니코드 문자의 문자열이다.
WideString 데이터 형은 COM 의 BSTR 데이터 형과 호환이 된다. 델파이는 COM 을 지원하기 위해 AnsiString 값을 WideString 데이터 형으로 변환하는 루틴을 제공하고 있으며, COM API 를 호출할 때에는 반드시 문자열을 WideString 데이터 형으로 명시적인 형변환을 해 주어야 한다.
윈도우는 현재 유니코드의 지원을 위해 기본적인 SBCS(single-byte character set)과 더불어 MBCS(multibyte character set)를 지원한다. SBCS 에서는 각 바이트가 한 문자를 나타내며, ANSI 문자를 사용하는 대부분의 영어 문자권의 나라들이 이를 사용하게 된다. MBCS 는 어떤 문자는 한 바이트로 표현하고, 어떤 문자는 한 바이트 이상으로 표현한다. 보통 아시아권 언어의 경우 2 바이트를 사용하기 때문에 흔히 DBCS 로 표현한다.
유니코드 문자 세트에서는 모든 문자들이 2 바이트로 표현된다. 이러한 유니코드 문자는 델파이의 WideChar 데이터 형과 호환되며, 유니코드 문자열은 WideString 데이터 형과 호환된다.

- Null 종료 문자열의 이해

Null 종료 문자열은 Null(#0) 값으로 끝나는 문자의 배열이다. 이 배열에는 길이를 나타내는 부분이 없기 때문에, 처음으로 나타나는 Null 문자를 문자열의 끝으로 간주하는 것이다. 예를 들어, 다음의 type 선언문이 Null 종료 문자열을 저장하기 위해 사용될 수 있다.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..255] of Char;
  TMemoText = array[0..1023] of WideChar;
```

이렇게 선언한 문자 배열에 기본적으로 Null 문자를 채워넣고, ShortString 문자의 값을 대입하는 식으로 문자를 변환하는 루틴을 곳곳에서 발견할 수 있을 것이다.

- 파스칼 문자열과 Null 종료 문자열의 혼용

긴 문자열인 AnsiString 값과 Null 종료 문자열인 PChar 값은 쉽게 섞어서 사용할 수 있다.

PChar 값은 긴 문자열 변수에 직접 대입이 가능하다. 즉, S 가 AnsiString 이고 P 가 PChar 형이라면 S := P 와 같은 표현식이 사용될 수 있다.
그렇지만, 경우에 따라서는 문자열의 형변환이 필요한 경우가 있다. 예를 들어, 2 개의 PChar 문자열을 붙여서 하나의 AnsiString 으로 저장하려면 다음과 같이 사용하여야 한다.

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

긴 문자열을 Null 종료 문자열로 변환시킬 때에도 마찬가지 형태의 형변환을 해주면 된다. 예를 들어 Str1, Str2 가 AnsiString 문자열일 때 파라미터로 Null 종료 문자열을 요구하는 MessageBox API 함수는 다음과 같이 호출한다.

```
MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
```

WideString 과 PWideChar 데이터 형의 관계 역시 AnsiString 과 PChar 의 관계와 동일하므로 동일한 규칙을 따르면 된다.

오브젝트 파스칼의 사용자 정의 데이터 형

표준 파스칼은 subrange, enumerated, set 등의 사용자 정의 데이터 형을 제공한다. 이들을 효과적으로 이용하면 유연한 데이터 구조를 만들어낼 수 있으며, 이해하기 쉬운 코드를 작성할 수 있다. 이 밖에도 오브젝트 파스칼에서는 배열형, 레코드 형, 포인터 형, 프로시저 형 등도 지원되는데, 이들은 이 장의 후반부에 더 자세하게 다룬다.

- Subrange 형

Subrange 데이터 형은 이름이 암시하듯이 사용자가 범위를 한정해서 사용할 수 있는 데이터 형이다. 정의는 매우 간단하게 가장 작은 값과 가장 큰 값을 양쪽 끝에 쓰고 이들 사이에 '..'을 사용하면 된다. 간단히 예를 들어보자.

```
type
  TNibble = 0..15;
```

여기에서 TNibble 데이터 형은 0~15 까지의 정수를 저장할 수 있다. 이 범위를 벗어난 값을 대입하려고 하면 에러를 발생시킨다. 여기에서 범위로 지정할 수 있는 것은 0 을 넘어선 값이나, 문자도 가능하다. 즉, 다음과 같은 코드의 사용이 가능하다.

```
type
  THiNibble = 16..255;
  TNumericChar = '0'..'9';
```

Subrange 데이터 형은 특정 범위를 제한해야 하는 변수 등을 사용할 때나 디버깅을 할 때 아주 요긴하게 쓸 수 있는 데이터 형이다.
앞에서도 언급했듯이 범위를 벗어난 값을 대입하려 하면 에러를 발생시키는데, 이때 컴파일러의 Range checking 여부에 따라서 컴파일 또는 런타임에서 에러가 발생한다. 만약에 Range checking 이 가능한 것으로 설정되어 있으면 ({$R+}), 컴파일 할 때와 런타임에서 모두 에러를 발생시킨다. 그런데, Range checking 이 가능한 것으로 설정된 경우에는 코드에 이를 위한 부분이 추가되기 때문에 다소나마 프로그램의 수행 성능을 저하시키므로, 디버깅이 충분히 된 이후에는 이 옵션을 끄고 최종적으로 컴파일해서 배포하는 것이 좋다.

- 열거(Enumerated) 형

열거형은 아마도 VCL 소스 코드에서 가장 빈번하게 찾아볼 수 있는 데이터 형일 것이다. 간단히 설명하면 괄호에 의해 둘러싸이고, 콤마를 통해 구분된 기호화된 서수(ordinal) 값이라고 생각하면 된다. 말보다 눈으로 보는 것이 더욱 알기 쉬울 것이므로 다음의 코드를 살펴 보자.

```
type
  TDayOfWeek = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

열거형으로 선언된 변수는 이들 중의 하나의 값을 대입하게 된다. 예를 들어 보면,

```
var
  DOW: TDayOfWeek;
begin
  DOW = Sunday;
end;
```

열거형을 이용한 VCL 소스 코드의 예를 들면, 다음과 같은 것들이 있다.

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields,
  dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc);
TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
```

```
TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit,
  nbPost, nbCancel, nbRefresh);
```

열거형을 이용한 프로퍼티의 경우 오브젝트 인스펙터에서 드롭-다운 리스트에서 선택할 수 있다. 열거형은 또한 subrange 데이터 형과 같이 사용하면 매우 유용하다. 예를 들어, 앞에서 선언한 TDayOfWeek 열거형을 이용해서 다음과 같이 사용할 수 있다.

```
const
  WorkDays = array[Monday..Friday] of string[3] = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri');
```

열거형은 사실상 0 에서부터 시작하는 서수형 값의 기호형태라고 생각하면 된다. 즉, TDayOfWeek 는 다음 코드와 같은 의미이다.

```
type
  TDayOfWeek = (0, 1, 2, 3, 4, 5, 6);
```

이렇게 열거형을 사용하는 이유는 코드가 읽기 편해지고, 여러모로 헷갈리기 쉬운 정수를 무차별 적으로 나열하기 보다는 이를 사용하면 훨씬 안전하고, 효율적이기 때문이다. 표준 ANSI C/C++에는 파스칼의 열거형에 해당하는 enum 형이 존재한다.

- 세트(Set) 형

세트형은 파스칼에서 가장 유용한 데이터 형이라고 말해도 과언이 아닌 중요한 데이터 형이다. 세트형을 간단하게 말하자면 값들의 목록 또는 집합이라고 설명할 수 있다. 세트를 구성하는 값들을 요소(element) 또는 세트 멤버라고 한다.
그러면, 실제로 세트를 정의하는 방법을 알아보자.

```
type
  TYesNo = set of Char;
var
  YN: TYesNo;
begin
  YN := ['Y', 'y', 'N', 'n'];
end;
```

그다지 어려운 방법은 아니다.

사실 세트 데이터 형의 가장 큰 장점은 여러가지 연산이 가능하다는 것이다. 중학교 수학에서 배우는 각종 집합 연산을 적용할 수 있으며, 이것이 아주 유용하게 쓰일 수 있다. 대표적인 연산으로는 합(union), 곱(intersection), 차(difference), 등호(equivalence), 부분집합(membership) 등이 있다. 이해하기 어렵지는 않다고 생각되지만, 직관적으로 알아보기 위해 다음 표를 살펴보자.

연산자	의 미	표현식	결 과
+	Set union	[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]
*	Set intersection	['A', 'B', 'C'] * ['B', 'C', 'D']	['B', 'C']
-	Set difference	['A', 'B', 'C'] - ['B', 'C', 'D']	['A']
=	Set equivalence	['A', 'B', 'C'] = ['A', 'B', 'C']	True
in	Set membership	'A' in ['A', 'B', 'C']	True

참고로, 세트 연산을 위해서는 같은 세트의 데이터 형은 같아야 한다. 예를 들어 다음의 표현식은 부당하다.

['Y', 'y', 'N', 'n'] + [1, 0]

세트 역시 열거형과 마찬가지로 subrange 데이터 형을 적절히 사용하면 편리하게 쓸 수 있다. 즉, 다음과 같은 선언이 가능하다.

['A'..'G']
['0'..'9', 'a'..'z', '#', '*', '-', '+', '/']

숫자에 대해서도 마찬가지로 적용할 수 있다. 세트의 연산을 이용하면 더욱 다양하게 세트의 멤버를 구성할 수 있다. 예를 들어 1 부터 30 까지의 수를 세트로 가지되 5, 15, 25 를 제외하고 싶다면 다음과 같이 정의하면 된다.

[1..30] - [5, 15, 25]

이를 앞에서 보여준 subrange 데이터 형을 이용해서 표현하면 다음과 같다.

[1..4, 6..14, 16..24, 26..30]

세트의 연산식 중에서 가장 유용한 것을 꼽으라면 필자는 'in' 을 꼽을 것이다. 판단문에서 특히 유용하게 사용할 수 있는데, 다음의 예를 살펴보자.

```
if (Number >=10) and (Number <= 100) then DoSomething;
if (Number = 1) or (Number = 5) or (Number = 9) or (Number = 10) then DoSomething;
```

이것은 다음의 코드들과 같은 의미를 가진다.

```
if (Number in [10..100]) then DoSomething;
if (Number in [1, 5, 9, 10]) then DoSomething;
```

세트 형의 구현은 생각보다 간단하다. 즉, 각각의 요소마다 한 비트 씩을 차지하여 그 비트가 1 일 경우 해당되는 요소가 포함되며, 0 일 경우 제외된다. 비트간 연산을 통해 합, 곱, 차 등이 계산될 수 있는 것이다. 그러므로, 세트 연산은 대단히 빠르게 동작한다.
세트와 열거형, subrange 데이터 형은 같이 많이 사용되며 VCL 소스 코드를 살펴 보면 무수히 많은 예를 볼 수 있다. 보통 세트형의 경우 열거형으로 선언된 데이터 형의 이름의 제일 뒤에 's'를 붙인 이름을 많이 붙이게 된다.
예를 구체적으로 들자면, 앞에서 열거형으로 선언된 TBorderIcon 데이터 형의 세트형은 TBorderIcons 로 선언되어 있다. VCL 코드를 살펴보면 간단하게 다음과 같이 정의되어 있다.

```
TBorderIcons = set of TBorderIcon;
```

TBorderIcons 를 실제로 사용하는 예를 들어보도록 하자. 폼의 최대화, 최소화 상태를 검사하기 위해서는 다음과 같이 하면 된다.

```
if ((BorderIcons * [biMaximize, biMinimize] = [biMaximize, biMinimize]) then DoSomething;
```

다소 복잡해 보이지만 최대화, 최소화된 상태가 biSystemMenu 일 경우를 포함하기 위해서 집합곱을 이용하였다. 즉, [biSystemMenu, biMaximize, biMinimize]인 경우도 포함한다.

배열 (Arrays)

배열은 base type 이라고 하는 같은 데이터 형의 요소 들로 이루어진 색인된 컬렉션이다. 배열은 정적인 배열과 동적 배열이 있다. 동적 배열형은 델파이 4 에서부터 지원되는 새로운 데이터 형이다.

- 정적 배열 (Static arrays)

정적 배열형은 다음과 같이 선언한다.

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

여기서 인덱스로 사용되는 indexType 은 2GB 범위를 넘지 않는다면 어떤 서수형(ordinal type)도 사용할 수 있다. 보통은 정수의 subrange 를 이용한다. 1 차원 배열의 간단한 선언 예는 다음과 같다.

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

이렇게 하면, MyArray 는 100 자의 문자를 담게 되는 배열 변수가 된다. 정적 배열을 생성해도 값을 대입하지 않으면, 메모리는 할당되어 있지만, 그 값은 임의의 값이 들어 있는 초기화가 되지 않은 변수가 된다.
다차원 배열의 선언은 다음과 같이 한다.

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

또는,

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

어떤 방법으로든 TMatrix 가 선언되면, 이 배열 변수는 실제로 500 개의 실수값을 담을 수 있게 된다.
배열의 값을 검사하는 데에는 앞에서 잠시 언급한 서수형 처리 루틴인 Low, High 가 유용하게 사용된다. 이를 이용하면 배열의 첫번째 요소에서 마지막 요소까지 검사하는 루틴을 손쉽게 작성할 수 있다.

- 동적 배열 (Dynamic arrays)

델파이 4 에서는 동적 배열의 선언이 가능해 졌다. 동적 배열은 타입 정보(배열의 차원과 요소의 데이터 형)를 지정하되, 요소의 수를 지정하지 않는다. 즉, 다음과 같이 선언하면 된다.

```
var
  A: array of integer;
  B: array of array of string;
```

A 는 정수형의 1 차원 배열이며, B 는 문자열의 2 차원 배열이다.

동적 배열은 배열에 새로운 값을 대입하거나 SetLength 프로시저를 호출하면 해당되는 메모리를 재할당한다. 앞에서와 같이 선언만 한 경우에는 그에 해당되는 메모리가 할당되지 않는다. 동적 배열을 메모리에 생성하려면 SetLength 프로시저를 호출한다. 예를 들어 앞에서의 A 배열의 경우 다음과 같이 선언해보자.

```
SetLength(A, 20);
```

이 문장을 통해 20 개의 정수를 저장할 수 있는 배열이 생성된다. 동적 배열은 언제나 0 부터 시작하는 정수 인덱스를 가진다.
동적 배열 변수는 내부적으로는 포인터이며, 델파이 2 에서부터 지원되는 문자열 변수 처럼 참조 계수 테크닉을 이용하여 관리된다. 동적 배열을 해제하려면 변수에 nil 을 대입하거나 Finalize 메소드를 호출한다.
만약에 X, Y 가 같은 동적 배열 형을 가진 변수이면 X := Y 와 같은 표현식을 이용하여 배열 Y 의 크기와 같도록 배열 X 의 크기가 결정되며, X 는 Y 와 같은 배열을 가리키게 된다. 문자열과는 달리 배열은 복사된 후의 다른 변수의 변경 값이 원본 변수에 반영된다. 다음의 코드를 살펴보자.

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

이 코드를 실행하면 A[0]의 값이 2 로 설정된다.
동적 배열 변수가 비교될 때에는 참조값이 비교되는 것이지, 배열의 값들이 비교되지 않는다. 다음의 코드를 살펴보자.

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

여기에서 A = B 는 False 이지만, A[0] = B[0]는 True 를 반환한다.
Copy 함수를 이용하면 동적 배열을 잘라낼 수 있다. 예를 들어, A := Copy(A, 0, 20) 이라는 표현식은 동적 배열 A 의 처음 20 개의 요소만을 남겨두고 나머지는 제거한다.
일단 동적 배열이 할당되면 Length, High, Low 등의 표준 함수를 이용할 수 있다. Length 는 배열의 요소 수를 반환하며, High 와 Low 는 각각 동적 배열의 가장 큰 인덱스와 0 을 반환한다.
다차원 동적 배열의 선언 후 이를 사용하는 방법도 일차원 동적 배열과 마찬가지이다.
예를 들어 다음과 같이 다차원 배열을 선언해 보자.

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

이 동적 배열을 인스턴스화 하려면, 다음과 같이 하면 된다.

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

이 코드에서 IxJ 크기의 2 차원 배열이 할당된다. 또한, 이렇게 SetLength 를 이용하여 다차원 배열을 선언한 후 할당할 때 다양한 테크닉을 사용할 수 있다. 다음의 코드를 살펴보자.

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

이 코드에 의해 Ints 에 10 개의 행이 할당되지만 열은 없다. 이제 한번에 하나씩의 열에

대해 배열의 할당이 가능하다. 예를 들어, 다음의 코드를 보자.

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

이 코드는 3 번째 행에 5 개의 요소가 할당된다. 이제 이 배열에 값을 대입할 때에는 Ints[2, 4] := 6 과 같은 표현식을 이용하면 된다.
다음 예제는 IntToStr 함수와 동적 배열을 이용해서 문자열의 삼각형 모양의 매트릭스로 생성한다.

```
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;
```

레코드 형 (Record type)

레코드는 C 에서의 구조체(structure)와 비슷한 역할을 하는 것으로, 여러가지 데이터 요소의 집합을 대표할 수 있는 데이터 형이다. 각 요소를 필드라고 하며, 레코드 형의 선언은 각 필드의 이름과 데이터 형을 지정한다. 그러면 실제 선언부분을 살펴 보자.

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

TDateRec 레코드는 3 개의 필드로 구성되는데 각 필드는 정수형인 Year, 열거형인 Month, 1~31 까지의 subrange 형인 Day 이다. 이들은 마치 다른 변수처럼 접근해서 사용하면 된다. 그러면, 이런 레코드 형 변수를 선언하고 사용하는 방법을 알아보자.

```
var
  Record1, Record2: TDateRec;

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

보통, 레코드 형과 같이 중복되는 부분이 있는 경우에는 다음과 같이 with 문을 사용하면 유용하다.

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

레코드 형에는 이런 표준적인 데이터 형 이외에도 가변 파트를 포함한 필드를 사용할 수 있다. 레코드 형에 가변 파트를 추가하고 사용할 때에는 다음과 같은 문법을 따른다.

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

  case tag: ordinalType of
   constantList1: (variant1);
   ...
   constantListn: (variantn);
  end;
```

기본적인 선언 부분은 표준 레코드 형과 동일하다. 그런데, 특기할 것은 case 문을 이용해서 가변 파트를 지정하는 부분이다.
여기서의 tag 는 옵션으로 사용할 수 있는 identifier 로 유효한 identifier 는 어느 것이나 사용할 수 있다. 이것이 빠지면, ':'도 같이 빼 주어야 한다. ordinalType 은 서수형이면 어는 것이나 사용할 수 있다는 의미이다.
여기에 대해 자세히 설명하는 이유는 델파이의 VCL 소스 코드를 보면 언뜻 처음 봐서는 도저히 이해가 가지 않는 코드들이 있는데, 이들 중 가변 파트를 이용한 레코드인 경우가 많다. 필자도 델파이 사용자 모임에 갔을 때, 이런 소스 코드에 대해서 질문을 많이 받았는데 문법적으로 제대로 설명된 적이 없어서 다소 이해가 힘들었던 기억이 있다.
각각의 필드 리스트에는 여러가지 데이터 형을 사용할 수 있지만, 긴 문자열이나 동적 배열, 가변형, 인터페이스나 이들이 포함된 레코드 형은 사용하면 안된다. 그렇지만, 이런 데이터 형이 필요한 경우에는 포인터를 이용하면 된다.
가변 파트를 포함한 레코드는 문법적으로 사용하기가 다소 복잡하지만, 상당히 유용한 데이터 형이다. 이런 레코드에는 같은 메모리 공간을 공유하는 여러 종류의 가변 데이터를 포함할 수 있다. 즉, 데이터의 종류가 다르지만 이를 모두 독립된 필드로 구성하기 싫은 경우에 사용할 수 있다.
그러면, 실제로 사용되는 예를 살펴보자.

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

여기에서 앞에서도 잠시 설명했지만, 'case Salaried: Boolean of' 문장은 'case Boolean of' 와 같은 의미가 된다. 아마도 델파이 소스에서 'case Integer of' 와 같은 문장을 보면서 어떤 의미인지 몰랐던 사람들은 같은 의미로 생각하면 된다. Integer, Boolean 데이터 형이 둘 다 일종의 서수형이기 때문에 사용에 무리가 없다.
앞의 코드에서 모든 employee 는 보수를 지급받을 때, 어떤 경우에는 연봉으로 받을 수도 있고, 시간당으로 받을 수도 있을 것이다. 그렇지만, 이 두 가지를 동시에 사용하는 경우는 없다. 참고로 여기서는 2 가지 경우를 생각했기 때문에 Boolean 을 사용했지만, 월급과 같은 경우를 추가한다면 Integer 등을 사용해야 할 것이다. 이런 경우에 TEmployee 레코드 형의 인스턴스를 생성할 때 두 필드를 모두 포함시키는 것은 다소 낭비이다. 그러므로, 가

변 파트를 추가해서, 서로 다른 데이터 형을 하나의 필드에 사용하게 하면 효율적이다.
조금 더 복잡한 예를 알아 보자.

```
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;
```

이와 같이 미리 열거형을 이용해서 정의한 경우에는 보다 편리하게 사용할 수 있다. 각각의 레코드 인스턴스에 대해 컴파일러가 가장 큰 데이터 형을 기준으로 충분한 메모리를 할당하게 되고, Rectangle, Triangle 과 같은 상수 들은 사실상 컴파일러가 필드를 관리하는데 아무런 영향을 미치지는 않는다. 다만, 이렇게 선언할 경우 프로그래머가 코드를 읽거나 관리하는데 도움이 된다.
가변 파트를 이용하는 또 하나의 예로는, 컴파일러가 형변환을 허용하지 않는 경우 서로 다른 데이터 형의 변환 효과를 노릴 수 있다는 것이다. 예를 들어, 64 비트 실수와 32 비트 정수를 하나의 필드에 선언했을 때, 이들 간의 자연스런 변환이 가능하다.
가변 파트를 이용한 레코드 형의 가장 대표적이 사용 예라고 할 수 있는 TMessage 의 선언부를 살펴보자

```
TMessage = record
    Msg: Cardinal;
    case Integer of
      0: (WParam: Longint; LParam: Longint; Result: Longint);
      1: (WParamLo: Word; WParamHi: Word; LParamLo: Word; LParamHi: Word;
        ResultLo: Word; ResultHi: Word);
end;
```

즉, 여기서는 WParam, LParam, Result 로 접근해도 되고 WParamLo, WParamHi 와 같이 접근해도 된다. 그렇지만, 기본적으로 LongInt 데이터 형이 Word 데이터 형 2 개와 크기가 같기 때문에 결국에는 어떻게 접근해도 데이터의 일관성이 유지된다. 아마도 TMessage 의 경우를 살펴 보면 가변 파트를 포함한 레코드 형의 장점을 이해할 수 있을 것이다.

포인터 형 (Pointer type)

포인터는 메모리 주소를 가리키는 것이다. 포인터가 특정 변수나 데이터의 주소를 가지고 있을 때에, 이를 가리켜 포인터가 변수나 데이터를 참조(reference)하고 있다고 한다. 간단한 용어 정의 같지만, 이 개념은 무척 중요한 것이므로 꼭 알아두기 바란다. 만약, 포인터가 가리키는 것이 레코드나 배열 처럼 여러 개의 요소로 이루어진 경우라면, 포인터는 첫번째 요소의 주소를 참조한다.
포인터가 특정 데이터 형을 가리키는 경우에는 이를 typed 포인터라고 하며, 여러가지 데이터 형을 모두 가질 수 있는 포인터 변수를 untyped 포인터 변수라고 한다.
다음의 코드를 살펴 보자/

```
var
  X, Y: Integer;
  P: ^Integer;
begin
  X := 17;
  P := @X;
  Y := P^;
end;
```

여기에서 P 는 정수값에 대한 포인터로 선언된다. 이는 P 변수가 X 나 Y 변수의 위치를 가리킬 수 있다는 것을 의미한다. 실제로 포인터인 P 에 X 의 주소를 대입하였으며, 그 다음 줄에서 P 의 값을 Y 에 대입하였다. 그러므로, 결과적으로 이 코드는 X 의 값을 Y 에 대입하게 되므로 X, Y 모두 17 을 값으로 가지게 된다.
여기에서 '@' 연산자는 변수나 함수, 프로시저의 주소를 나타내는데 사용되며, '^' 기호는 데이터 형 identifier 앞에 사용될 경우에는 그 데이터 형에 대한 포인터 임을 나타내고, 포인터 변수의 뒤에 쓰일 때에는 그 주소에 들어 있는 값을 나타내게 된다.
델파이에서 포인터에 대해 잘 익혀 두어야 하는 이유는 다음과 같다.

1. 포인터에 대한 이해는 오브젝트 파스칼을 이해하는데 큰 도움이 된다. 이는 오브젝트 파스칼에서 명시적으로는 보이지 않지만, 내부적으로 작동하는 핵심적인 부분이다.
2. 데이터 형이 크고, 동적으로 할당될 필요가 있을 때에는 포인터를 사용한다. 또한, 긴 문자열의 경우 내부적으로 포인터에 의해 동작하며, 클래스 변수 들도 본질적으로 포인터이다.
3. 포인터는 오브젝트 파스칼의 엄격한 데이터 형 검사를 비켜갈 수 있는 해결책을 제시한

다. 예를 들어 다음의 코드를 살펴보자.

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

이런 코드에 의해 R 에 저장된 실수형을 정수형 변수인 I 에 저장하는 것이 가능하다. 물론, 실수와 정수는 다른 형태로 저장된다. 그러므로, 이런 형태의 복사는 단순히 이진 데이터를 전달하는 것이지, 적절한 형태의 형변환은 아니다.
포인터 변수가 아무 것도 참조하지 않고 있을 때에는 nil 이라는 예약된 상수를 사용한다.
포인터를 선언할 때에 기본적인 문법은 데이터 형 앞에 '^' 기호를 사용하면 된다. 포인터를 가장 유용하게 사용하는 예는 역시 레코드와 같이 다소 메모리를 차지하는 데이터 형에 포인터를 선언하여, 이들 데이터 블록이 포함된 메모리를 직접 처리하지 않고 포인터를 이용해 처리하는 것이다.
또한, 포인터 중에서는 어떤 데이터 형도 참조할 수 있는 untyped 포인터 형도 존재하는데, 이것은 단순히 Pointer 로 선언하며 직접 값을 알아내지는 못하고 일단 특정 데이터 형의 포인터로 형변환한 후에 '^' 기호를 사용하여 값을 알아낸다.

프로시저 형 (Procedural types)

프로시저 형은 프로시저나 함수를 변수나 다른 프로시저나 함수에 파라미터로 전달할 수 있게 해준다. 예를 들어, 다음과 같이 두개의 정수 파라미터를 받아서 정수값을 반환하는 Calc 라는 함수가 있다고 하자.

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

Calc 함수를 F 라는 변수에 다음과 같이 지정할 수 있다.

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

F := Calc;
```

이때, 함수나 프로시저의 파라미터와 리턴 값에 대한 것을 지정한 부분이 바로 프로시저 형이다. 앞의 코드와 같이 직접 변수 선언 부분에 사용할 수도 있고, 새로운 type 으로 선언하고 이를 사용하는 경우가 있는데 후자의 경우가 더 자주 쓰인다.
다음의 코드를 살펴보자.

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

이런 데이터 형을 메소드 포인터라고 한다. 메소드 포인터는 메소드의 주소와 메소드가 속해있는 객체의 참조값(reference)을 저장한다. 메소드 포인터는 다음과 같이 이벤트 선언에서 매우 자주 사용된다.

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

델파이로 프로그래밍 하다 보면 자주 있게 되는 일인데, 다음의 대입문으로 버튼의 OnClick 이벤트 핸들러를 OnClick 이라는 메소드 포인터 변수에 저장할 수 있다.

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

프로시저 형이 서로 호환되는 것은 호출 규칙(calling convention)이 같고, 리턴 값과 파라

미터의 수와 데이터 형이 같으면 된다. 파라미터의 이름은 상관 없다.
참고로, 전역 프로시저 포인터 형은 메소드 포인터 형과는 호환되지 않는다는 것을 알아두기 바란다. 그리고, 프로시저 형 변수에는 nil 을 대입할 수 있다.
대입문에서 프로시저 형의 경우라면 같은 프로시저 형이 아니라 리턴 값의 데이터 형과 같은 변수에 대입하는 문장이 있을 수 있다. 이럴 때에는 함수의 연산 결과가 대입된다. 다음의 코드를 살펴 보자.

```
var
  F, G: function: Integer;
  I: Integer;
function SomeFunction: Integer;
 ...
F := SomeFunction;          //SomeFunction 함수를 F 에 대입
G := F;                     //F 를 G 에 복사
I := G;                     //함수를 실행한 결과 값을 I 에 대입
```

처음 F, G 에 대한 대입문은 프로시저 형의 값을 대입하는 것이지만, I 에 대한 대입문은 함수의 실행 결과인 정수값을 대입하게 된다.
프로시저 형을 사용할 때에 판단문의 비교 대상을 구별하는 것도 혼돈하기 쉬운 부분이다.

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

이 문장에서 F 와 MyFunction 이 모두 함수 이름이라고 할 때 이 판단문의 결과는 F 함수 호출의 결과값과 MyFunction 함수 호출의 결과값을 서로 비교하는 것이다. 프로시저 형의 변수가 표현식에 있을 경우에 참조되고 있는 프로시저나 함수를 호출하는 것이 규칙이다. 만약에 앞의 문장에서 F 가 리턴 값이 없는 프로시저를 참조하고 있거나, 파라미터를 필요로 하는 함수를 참조하는 경우에는 컴파일 에러가 발생한다. 그러므로, 프로시저 값을 서로 비교할 때에는 '@' 연산자를 사용하여 다음과 같이 비교해야 한다.

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

@F 는 F 를 untyped 포인터 변수로 변환하게 되므로 주소를 가지게 되며, @MyFunction 은 MyFunction 의 주소를 반환한다. 그러므로, 프로시저 변수 자체의 메모리 주소를 얻고자 할 때에는 @@F 와 같이 사용해야 한다.
프로시저 변수의 값은 nil 일 수도 있는데, 이 경우에는 아무것도 참조하지 않게 된다. 그렇지만 nil 포인터를 직접 사용하면 에러가 발생하기 때문에, 변수의 값이 할당되어 있는지 알

아볼 필요가 있는데 이럴 때 사용되는 것이 Assigned 함수이다. 이 함수는 다음과 같이 주로 컴포넌트에서 이벤트 핸들러가 있는지 확인할 때 많이 사용된다. :

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

가변형 (Variant types)

어쩔 때에는 여러가지 데이터 형을 다룰 필요가 있다. 그런데, 컴파일하기 전에 이런 변수를 결정하기 어려울 때에는 가변형 변수를 사용한다. 가변형은 다소 처리 속도가 느리고, 메모리를 많이 사용하지만 유연한 처리를 반드시 필요로 할 때에는 유용한 해결 방안이 되기도 한다.
가변형에 담을 수 없는 데이터 형은 레코드, 세트, 정적 배열, 파일, 클래스, 클래스 레퍼런스와 포인터이다. 즉, 가변형 변수에는 포인터와 구조체의 형태를 가지는 데이터 형을 제외한 다른 데이터 형은 모두 담을 수 있다. 가변형 변수가 중요하게 여겨지는 또 하나의 목적은 COM 객체를 이용할 수 있다는 것이다.
모든 가변형 변수는 초기화될 때 Unassigned 라는 특별한 값으로 초기화 된다. 그리고, Null 이라는 값을 가질 수 있는데 이 값은 데이터가 빠져 있거나, 알 수 없는 경우이다.
정적 배열을 가변형 데이터로 활용하려면 가변형 배열을 선언해서 사용하여야 한다. 이럴 때에는 VarArrayCreate, VarArrayOf 와 같은 표준 함수를 사용한다. 다음의 코드를 살펴보자.

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

이 코드의 결과로 10 개의 요소를 가지는 정수형의 배열이 생성되며, 이 값이 가변형 변수인 V 에 저장된다. 이렇게 가변형 배열로 선언된 가변형 변수는 인덱스를 사용해서 V[0], V[1]과 같은 형태로 배열의 요소에 접근할 수 있다. VarArrayCreate 함수의 두번째 파라미터에는 배열의 데이터 형을 지정한다. varInteger 는 정수형을 나타내며, 그 밖에 여러가지 데이터 형 코드를 사용할 수 있지만 varString 은 사용할 수 없다. 문자열의 배열을 이용해야 한다면 varOleStr 코드를 사용한다.
가변형 배열의 기초 데이터 형을 가변형으로 설정하면 더욱 유연하게 사용할 수도 있다. 가변형에 관련된 함수는 상당히 많고, 도움말도 비교적 자세하므로 더 깊은 내용에 대해서는 이를 참고하기 바란다.
가변형에서 델파이 4 에 새롭게 추가된 데이터 형이 있는데, Any 라는 데이터 형이 그것이다. 이 데이터 형은 CORBA 와 호환되는 가변형 변수로 사용된다. COM 에 의해 사용되는

OleVariant 와 비슷한 목적으로 사용된다고 이해하면 된다.

연산자 (Operatror)

델파이에서는 비교와 평가 등에 연산자를 사용하게 된다. 연산자에는 산술 연산자와 논리 연산자, 관계 연산자가 있다. 이들 각각에 대한 설명은 도움말을 참고하기 바라며, 각 연산자의 우선 순위에 대해서 알아보자. 같은 순위에 있는 하나 이상의 연산자를 함께 사용할 때에는 기본적으로 왼쪽에서 오른쪽으로 연산을 수행해 나간다. 연산 순위를 높은 곳에서 낮은 곳을 정리해보면 다음과 같다.

연산자	설 명
. ^	필드, 포인터 dereferencing
@ not	단일 연산자
* / div mod as and shl shr	곱셈, 형변환 관련 연산자
+ - or xor	덧셈 관련 연산자
= <> < > <= >= in is	관계형 연산자

2 개의 연산자들 사이의 피연산자는 높은 우선 순위의 연산자에 우선적으로 적용된다. 동등 연산자들 사이의 피연산자는 왼쪽에 있는 것을 먼저 적용하며, 괄호 안의 연산식은 하나의 피연산자로 다루기 때문에 모든 연산식에 우선하게 된다.

언어 문장 (Statements)

오브젝트 파스칼의 문장(statement)은 개발자의 표현식을 실제로 판단하고 실행하는 기본 단위가 된다. 문장의 종류를 크게 둘로 나누면 단일문(simple statement)과 구조문(structured statement)으로 나눌 수 있다. 파스칼의 각 문장의 끝은 ‘;’ 기호로 구별된다.

- 단일문 (Simple statement)

단일문은 메모리를 변경하거나 값을 대입하고, 프로시저나 함수를 활성화하는 등의 문장이다. 단일문으로 대표적인 것은 대입(assignment), 프로시저, goto 문 등이 있다. Goto 문장은 많이 사용하지 않는 것이 좋으며, 오브젝트 파스칼에서 거의 사용할 일은 없다. 대입문과 프로시저 호출은 프로그램에서 가장 핵심적으로 사용되는 문장이다.

1. 대입문 (Assignment)

대입문은 프로그램 내에서 메모리의 내용을 변경하는 문장이라고 생각하면 된다. 대입문에 사용되는 대입 연산자(:=)의 우측에는 대입할 값을, 좌측에는 대입의 대상을 지정한다.
앞에서도 자주 코드가 등장하였으므로 다들 이해하고 있을 것으로 믿는다.
대입문에서 중요한 것은 대입 호환성(assignment compatibility)이 지켜지고 있는지 여부이다. 대입 호환성 규칙은 다소 복잡하지만, 기본적으로 좌우의 데이터 형을 같은 것을 사용한다고 생각하기 바라며 더 자세한 내용은 도움말을 참고하면 된다.

2. 프로시저문 (procedure statement)

프로시저 호출문은 메소드나 프로서저를 실행하기 위해 호출하는 문장이다. 전형적인 예는 다음과 같다.

```
procedurename(parameter1, parameter2);
```

여기서 중요한 것은 파라미터의 수와 데이터 형을 일치시키는 것이다. 이때 파라미터가 넘어가는 방식에 몇 가지가 존재하는데 여기에 대해서는 이 장의 후반부에 다루게 된다.

- 구조문 (Structured statement)

구조문은 순서대로 실행되어야 할 문장 들로 구성된 문장이다. 구조문은 주로 문장 들이 실행되는 방법을 조절하는데 이용된다. 오브젝트 파스칼이 지원하는 구조문은 복합문(compound statement), 조건문(conditional statement), 순환문(loops), with 문의 4 가지로 구분할 수 있다.

1. 복합문 (compound statement)

복합문이란 실행될 문장들을 하나의 블록으로 묶는 것으로 begin~end 키워드 사이에 수록된 문장들을 하나의 복합문으로 생각한다. 문법적인 표현은 다음과 같다.

```
begin
  statement1;
  statement2;
end;
```

2. 조건문 (conditional statement)

조건문은 특정 문장이 실행될 것인지 여부를 결정하는 문장으로 오브젝트 파스칼에는 if 와 case 의 2 가지 조건문이 있다.
if...then 조건문은 boolean 표현식 여부에 따라서 조건이 결정되는 경우이다. 즉, if 뒤에 따라 나오는 표현식이 True 이면 then 이후의 문장이 실행되고, False 이면 else 이후의 문장이 실행된다. Boolean 표현식에는 논리 연산에 사용되는 not, and, or, xor 등의 연산자를 사용할 수 있다.

```
if boolean_expression then
  statement1
else
  statement2;
```

여기에서 주의할 것은 if ... then ... else 문이 하나의 문장이기 때문에 else 문의 문장 뒤에 ‘;’을 사용한다는 것이다. 초보자들이 자주 틀리는 부분 중 하나가 else 문이 있는데, then 이후의 문장 뒤에 ‘;’을 사용하는 것이다.
case 문은 if 문과 달리 조건부에 여러가지 경우가 나타날 수 있는 경우에 사용된다. case 문의 조건 표현식에는 서수형 표현식이 사용된다.

```
case ordinal_expression of
  1: statement1;
  2: statement2;
  3: statement3;
  else otherstatement;
end;
```

3. 순환문 (loops)

순환문은 문장들을 여러 차례 실행할 수 있게 해주는 문장으로, 오브젝트 파스칼에서는 for...to/downto...do, while...do, repeat...until 의 3 가지 순환문을 지원한다.
이들 순환문에서 빠져 나갈 때에는 break, exit 의 2 가지를 사용할 수 있는데 break 는 중첩된 루프가 있을 경우 그 이전의 루프로 빠져나가며, exit 는 모든 루프에서 빠져나간다.
for...do 루프는 가장 간단한 순환문으로, 루프를 몇 번이나 실행시킬 것인지를 알고 있을 때에 사용한다.
간단한 사용 예는 다음과 같다.

```
for Count := 1 to 10 do i := i + Count;
```

이 문장은 Count 변수의 값이 1~10 까지 증가하면서, 10 번 루프를 돌게 된다. 만약 값을 감소시키면서 루프를 돌게 하려면 to 대신 downto 를 사용한다.
while...do 순환문은 루프에 들어가기 전에 조건문을 검사하게 된다. 조건이 거짓이면, 루프가 한번도 실행되지 않는다. 참고로 다음 문장은 무한 루프를 돌게 된다.

```
while True do Something;
```

repeat...until 문장은 repeat 와 until 사이의 문장들을 until 뒤에 지정한 조건이 참이 될 때까지 계속해서 반복 실행하게 되는 것이다. while...do 문장과는 달리 조건과 상관없이 한번은 실행된다. 문법은 다음과 같다.

```
repeat Something until Condition;
```

4. with...do 문

with 문장은 중복되는 코드가 있을 때에 유용하다. 사용 예는 이번 장의 앞부분에서 소개한 바 있으므로 생략하겠다.

프로시저와 함수 (Procedures and Functions)

파스칼에서는 루틴이라는 개념을 중요시 한다. 파스칼의 루틴에는 프로시저와 함수의 2 가지 종류가 있다. 프로시저와 함수의 유일한 실제 차이점은 함수는 리턴값을 가지며, 프로시저는 그렇지 않다는 것이다. 이들은 모두 선언부에는 루틴의 이름과 파라미터, 리턴값에 대한 선언을 해야 하며, 구현부분에서는 begin~end 블록에 둘러싸인 부분에 실제 루틴의 몸체를 구성한다.
가장 단순한 형태의 프로시저와 함수의 코드 예제를 살펴 보자.

```
procedure Hello;
begin
  ShowMessage('Hello !');
end;

function Hello: Boolean;
begin
  ShowMessage('Hello !');
```

```
  Result := True;
end;
```

이 두가지 프로시저와 함수는 본질적으로 똑같은 동작을 한다. 다만 후자의 경우에는 True 를 리턴한다. 이렇게 리턴값을 지정할 때에는 앞의 코드와 같이 Result 변수에 값을 대입할 수도 있고, 다른 방법으로는 함수의 이름에 대입할 수 있다. 즉, 'Hello := True;'가 같은 의미가 된다.

- 파라미터의 전달 방법

파스칼 언어는 파라미터 전달을 값에 의해서도 할 수 있고(call by value), 참조(reference)에 의해서도 할 수 있다(call by reference). 파라미터를 참조에 의해 전달한다는 것은 그 값이 루틴의 형식상의 파라미터 내의 스택에 복사되지 않는다는 뜻이다. 그 대신 프로그램은 루틴의 코드 안에 있는 원래의 값을 참조하게 된다. 이렇게 하면 프로시저나 함수가 그 파라미터의 값을 바꾸게 할 수 있다. 이렇게 참조에 의해 파라미터를 전달하려면 var 키워드를 사용한다. 값에 의한 전달 방식은 프로시저 루틴에서 넘어온 파라미터의 값을 복사해서 사용하게 되므로 파라미터에 전달된 변수의 값을 변화시키지 않는다.
참조에 의한 파라미터의 전달은 미리 정의된 데이터 형, 구형의 문자열, 그릭 큰 레코드 들에 대해서도 적용된다. 실제로 델파이의 객체 들은 모두 참조에 의해 전달된다.
참고로 델파이 3 부터는 out 이라는 새로운 종류의 파라미터가 제공된다. out 파라미터는 초기값이 없고 단지 값을 반환하기 위해서만 사용된다. 이러한 파라미터 들은 오직 COM 프로시저나 함수들에 대해서만 사용된다. 초기값이 없다는 점을 빼고는 out 파라미터는 var 파라미터와 똑같이 작용한다.
그 밖에 상수 파라미터와 개방형 배열 파라미터, 가변 데이터형 개방형 배열 파라미터 등이 있는데 여기에 대해서 조금 더 알아보도록 하자.

1. 상수 파라미터 (const parameter)

상수 파라미터는 const 키워드를 이용해서 사용하는데, 루틴 내부에서 상수 파라미터에 값을 대입할 수 없다는 특징이 있다. 그렇기 때문에 컴파일러는 파라미터의 전달을 최적화시킬 수 있다. 다음의 코드를 살펴보자.

```
procedure Test(const Parameter1: Integer; var Parameter2: Integer);
begin
  Parameter2 := 10;          //참조에 의한 전달이므로 변수의 값을 변경할 수 있다.
  Parameter1 := 10;          //상수 파라미터에 값을 대입하려 했으므로 컴파일 에러가 발생한다.
```

```
end;
```

2. 개방형 배열 파라미터

파스칼에서는 파라미터의 수를 고정적으로 선언해야 하는 약점이 있는데, 이를 극복하기 위한 방법으로 개방형 배열을 사용하면 된다. 이 때 개방형 배열은 특정 데이터 형만 지정하며, 요소의 수가 몇 개가 될 지는 지정하지 않는 것을 말하며, 다음과 같이 정의한다.

```
var
  parameter1: array of atype;
```

그러면 실제로 이를 이용하는 예를 살펴보자. 가장 쉽게 생각할 수 있는 것은 합을 내는 함수이다. 즉, 몇 개의 수가 될지는 모르지만 개방형 배열에 저장된 요소를 모두 더하는 것이다.

```
function Sum(const Parameter1: array of Integer): Integer;
var
  i: integer;
begin
  Result := 0;
  for i := Low(Parameter1) to High(Parameter1) do Result := Result + Parameter1[i];
end;
```

여기에서 High 와 Low 라는 서수형 루틴을 유용하게 사용한다는 것을 잘 알 수 있을 것이다. 이 함수는 다음과 같이 호출할 수 있다.

```
var
  Test1: array[1..5] of Integer;
  i : Integer;
begin
  for i := 1 to 5 do Test1[i] := i;
  i := Sum(Test1);
end;
```

3. 가변 데이터형 개방형 배열 파라미터

형이 지정되지 않은 개방형 배열을 사용하여, 데이터 형을 지정하지 않고도 파라미터를 사용하는 방법도 있다. 이럴 때 사용하는 것이 array of const 라는 것으로 서로 다른 데이터 형을 지닌 요소들을 배열에 넣어서 한 번에 전달하는 것이다. 선언부는 다음과 같다.

```
const Parameter1: array of const;
```

가장 전형적인 사용 예는 문자열을 처리할 때 이용하는 Format 함수를 들 수 있다.

- 디폴트 파라미터 (Default parameters)

델파이 4 에서는 프로시저와 함수에 디폴트 파라미터를 사용할 수 있게 되었다. 디폴트 파라미터는 다음과 같은 형태로 지정한다.

```
Name: Type = value
```

프로시저나 함수를 호출할 때 선언부에 디폴트 파라미터를 포함하면, 이를 호출할 때 디폴트 값과 같다면 파라미터를 빼도 된다. 예를 들어 다음과 같이 선언한 프로시저가 있다고 하자.

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

이때, 다음의 두 문장은 같은 기능을 한다.

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

- 호출 규칙 (Calling conventions)

호출 규칙이란 파라미터를 전달하는 방법을 말한다. 델파이 2.0 부터는 fastcall 이라는 호출 규칙을 디폴트로 사용하는데, 가능하면 최대 3 개까지의 파라미터를 CPU 레지스터에 전달해서 이를 처리하기 때문에 처리 속도가 빠르다. fastcall 호출 규칙을 이용하려면 register 키워드를 사용한다. 그렇지만 디폴트가 fastcall 이기 때문에 보통의 경우 특별히 지정해줄 필요는 없다.
윈도우와의 호환성을 위해서는 stdcall 호출 규칙을 사용한다. 여기에 대해 더욱 자세한 사항은 DLL 에 대해서 다루는 24 장을 참고하기 바란다.

정 리 (Summary)

이번 장에서는 오브젝트 파스칼의 기본적인 데이터 형과 문법에 대해서 알아 보았다. 오브젝트 파스칼의 OOP 적인 특성에 대해서는 다음 장에서 다루게 될 것이다.
무슨 일을 하든 기초가 중요하다는 것은 당연한 일이다. 오브젝트 파스칼의 문법에 대한 이해가 충분하지 않은 델파이 프로그래머는 그 발전의 정도에 한계가 있기 마련이다. 이 책에서는 분량의 문제로 충분히 자세하게 문법에 대해서 다루지는 못하고 있지만, 델파이에서 제공되는 도움말이나 다른 서적을 이용해서 충분히 문법적인 지식을 익혀두도록 권하고 싶다.