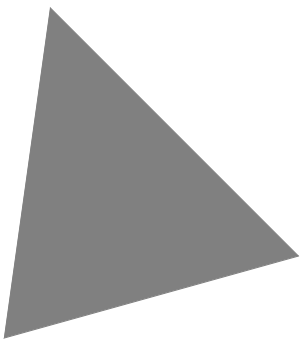
오브젝트 파스칼 랭귀지 안내서



Borland®

Object Pascal

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

Printed in the U.S.A.

HDB7010WW21000 1E0R0201
0001020304-9 8 7 6 5 4 3 2 1
D3

목 차

1 장

서문 1-1
설명서 내용 1-1
오브젝트 파스칼 사용 1-1
표기법 1-2
추가 정보 1-2
소프트웨어 등록 및 기술 지원 1-3

1 편
기본 랭귀지 설명

2 장

개요 2-1
프로그램 구성 2-1
파스칼 소스 파일 2-2
애플리케이션을 빌드하는데 사용되는 다른 파일 2- 2
컴파일러 생성 파일 2-3
예제 프로그램 2-3
간단한 콘솔 애플리케이션 2-3
좀 더 복잡한 예제 2-4
원시 애플리케이션 2-5

3 장

프로그램 및 유닛 3-1
프로그램 구조 및 구문 3-1
프로그램 헤더 3-2
프로그램 uses 절 3-2
블록 3-2
유닛 구조 및 구문 3-3
유닛 헤더 3-3
인터페이스 섹션 3-4
구현 섹션 3-4
초기화 섹션 3-4
완료 섹션 3-5
유닛 참조 및 uses 절 3-5
uses 절의 구문 3-5
다중 및 간접 유닛 참조 3-6
순환 유닛 참조 3-7

4 장

구문 요소 4-1
기본 구문 요소 4-1
특수 기호 4-2
식별자 4-2
한정된 식별자 4-2
예약어 4-3
지시어 4-3
숫자 4-4
레이블 4-4
문자열 4-4
주석 및 컴파일러 지시어 4-5
표현식 4-5
연산자 4-6
산술 연산자 4-6
부울 연산자 4-7
논리 (비트 단위) 연산자 4-8
문자열 연산자 4-9
포인터 연산자 4-9
집합 연산자 4-10
관계 연산자 4-10
클래스 연산자 4-11
@ 연산자 4-12
연산자 우선 순위 4-12
함수 호출 4-13
집합 생성자 4-13
인덱스 4-14
타입 변환 4-14
값 타입 변환 4-14
변수 타입 변환 4-15
선언과 문장 4-16
선언 4-16
문장 4-17
일반문 (Simple statements) 4-17
할당문 4-17
프로시저 및 함수 호출 4-18
Goto 문 4-18
구조문 4-19
복합문 4-20
With 문 4-20
If 문 4-22
Case 문 4-23
순환문 4-25
Repeat 문 4-25
While 문 4-25
For 문 4-26
블록과 유효 범위 4-27
블록 4-27
유효 범위 4-28
이름 충돌 4-28

5 장

데이터 타입 , 변수 및 상수 5-1
타입 정보 5-1
일반 타입 (Simple types) 5-2
순서 타입 (Oridinal types) 5-2
정수 타입 (Integer types) 5-3
문자 타입 (Character types) 5-5
부울 타입 (Boolean types) 5-5
열거 타입 (Enumerated types) 5-6
부분범위 타입 (Subrange types) . . . 5-8
실수 타입 (Real types) 5-9
문자열 타입 (String types) 5-10
짧은 문자열 (Short strings) 5-12
긴 문자열 (Long strings) 5-12
WideString 5-13
확장 문자 집합 정보 5-13
Null 종료 문자열 사용 5-13
포인터 , 배열 및 문자열 상수 사용 . . 5-14
파스칼 문자열과 Null 종료 문자열의 혼합 5-15
구조 타입 (Structured types) 5-16
집합 5-17
배열 5-18
정적 배열 5-18
동적 배열 5-19
배열 타입과 지정문 5-21
레코드 5-21
레코드의 가변 부분 5-22
파일 타입 (File types) 5-24
포인터와 포인터 타입 (Pointer types) . . . 5-25
포인터 개요 5-25
포인터 타입 (Pointer types) 5-27
문자 포인터 5-27
기타 표준 포인터 타입 5-27
프로시저 타입 (Procedural types) 5-28
문장 및 표현식의 프로시저 타입 5-29
가변 타입 (Variant types) 5-30
가변 타입 변환 5-31
표현식의 가변 타입 5-33
가변 타입 배열 5-33
OleVariant 5-34
타입 호환 및 구분 5-34
타입 구분 5-34
타입 호환성 5-35
할당 호환 5-36
타입 선언 5-36
변수 5-37
변수 선언 5-37
절대 주소 5-38
동적 변수 5-38
스레드 지역 변수 5-39
선언된 상수 5-39
True 상수 5-39
상수 표현식 5-41
리소스 문자열 5-41
타입이 지정된 상수 5-42
배열 상수 5-42
레코드 상수 5-43
프로시저 상수 5-43
포인터 상수 5-43

6 장

프로시저 및 함수 6-1
프로시저 및 함수 선언 6-1
프로시저 선언 6-2
함수 선언 6-3
호출 규칙 6-5
Forward 선언문 및 인터페이스 선언문 . . 6-6
외부 선언 6-6
객체 파일과 연결 6-7
라이브러리에서 함수 가져오기 . . . 6-7
프로시저 및 함수 오버로드 6-8
지역 선언 6-9
중첩 루틴 6-9
매개변수 6-10
매개변수 의미론 6-10
값 및 변수 매개변수 6-11
상수 매개변수 6-12
출력 매개변수 6-12
타입이 할당되지 않은 매개변수 . . . 6-13
문자열 매개변수 6-14
배열 매개변수 6-14
개방형 (open) 배열 매개변수 6-14
가변 개방형 배열 매개변수 6-15
기본 매개변수 6-16
기본 매개변수 및 오버로드된 루틴 . . 6-17
forward 선언문과 인터페이스 선언문의 기본 매개변수 6-18
프로시저 및 함수 호출 6-18
개방 타입 배열 생성자 6-18

7 장

클래스 및 객체 7-1
클래스 타입 7-2
상속 및 유효 범위 (scope) 7-3
TObject 및 TClass 7-3

클래스 타입의 호환성 7-3
객체 타입 7-4
클래스 멤버의 가시성 7-4
Private, protected 및 public 멤버. . . 7-5
Published 멤버 7-5
Automated 멤버 7-6
Forward 선언 및 상호 종속 클래스 . . . 7-6
필드. 7-7
메소드. 7-8
메소드 선언과 구현 7-8
Inherited 7-9
Self 7-9
메소드 바인딩 7-9
정적 메소드 7-10
가상 메소드 및 동적 메소드 7-10
추상 메소드 7-12
메소드 오버로드 7-12
생성자 7-13
소멸자 7-14
메시지 메소드 7-15
메시지 메소드 구현. 7-16
메시지 디스패칭 7-16
속성. 7-17
속성 액세스 7-17
배열 속성 7-19
Index 지정자. 7-20
저장소 지정자 7-21
속성 오버라이드 및 재선언 7-21
클래스 참조 7-23
클래스 참조 타입 7-23
생성자 및 클래스 참조 7-23
클래스 연산자 7-24
is 연산자 7-24
as 연산자. 7-25
클래스 메소드 7-25
예외. 7-26
예외를 사용하는 시기 7-26
예외 타입 선언 7-27
예외 발생 및 처리. 7-27
Try...except 문 7-28
예외 재발생 7-30
중첩 예외 7-31
Try...finally 문 7-31
표준 예외 클래스 및 루틴 7-32

8 장

표준 루틴 및 I/O 8-1
파일 입력 및 출력 8-1
텍스트 파일 8-3
타입이 지정되지 않은 파일 8-4
텍스트 파일 장치 드라이버. 8-4
장치 함수. 8-5
Open 함수 8-5
InOut 함수 8-5
Flush 함수 8-5
Close 함수 8-6
Null 종료 문자열 처리 8-6
와이드 문자 문자열. 8-7
기타 표준 루틴 8-7

2 편
전문적인 기능

9 장

라이브러리 및 패키지 9-1
동적으로 로드할 수 있는 라이브러리 호출 . .9-1
정적 로딩 9-1
동적 로딩 9-2
동적으로 로드할 수 있는 라이브러리 작성 . .9-3
exports 절 9-5
라이브러리 초기화 코드. 9-6
라이브러리의 전역 변수. 9-6
라이브러리 및 시스템 변수 9-7
라이브러리의 예외 및 런타임 오류 9-7
공유 메모리 관리자 (Windows 만 해당) . .9-8
패키지 9-8
패키지 선언 및 소스 파일 9-9
패키지 이름 지정 9-9
requires 절 9-10
contains 절. 9-10
패키지 컴파일 9-11
생성되는 파일 9-11
패키지 특정 컴파일러 지시어 9-11
패키지 특정 명령줄 컴파일러 스위치 . 9-12

10 장

객체 인터페이스 10-1
인터페이스 타입 10-1
IInterface 및 상속 10-2
인터페이스 식별 10-3
인터페이스의 호출 규칙. 10-3
인터페이스 속성 10-4
Forward 선언 10-4
인터페이스 구현 10-4
메소드 확인 절 10-5

상속된 구현 변경 10-6
위임 (Delegation) 으로 인터페이스 구현 . 10-6
인터페이스 타입 속성에 위임 10-7
클래스 타입 속성에 위임 10-7
인터페이스 참조 10-8
인터페이스 할당 호환 10-9
인터페이스 타입 변환 10-10
인터페이스 쿼리 10-10
Automation 객체 (Windows 만 해당) . . . 10-10
Dispatch 인터페이스 타입 (Windows 만 해당) 10-10
Dispatch 인터페이스 메소드 (Windows 만 해당) 10-11
Dispatch 인터페이스 속성 10-11
Automation 객체 액세스 (Windows 만 해당) 10-11
Automation 객체 메소드 호출 구문 . 10-12
이중 인터페이스 (Windows 만 해당) . . 10-13

11 장

메모리 관리 11-1
메모리 관리자 (Windows 만 해당) 11-1
변수 11-2
내부 데이터 형식 11-3
정수 타입 11-3
문자 타입 11-3
부울 타입 11-3
열거 타입 11-3
실수 타입 11-4
Real48 타입 11-4
Single 타입 11-4
Double 타입 11-5
Extended 타입 11-5
Comp 타입 11-5
Currency 타입 11-5
포인터 타입 11-5
짧은 문자열 타입 11-5
긴 문자열 타입 11-6
와이드 문자열 타입 11-6
집합 타입 11-7
정적 배열 타입 11-7
동적 배열 타입 11-7
레코드 타입 11-8
파일 타입 11-8
프로시저 타입 11-10
클래스 타입 11-10
클래스 참조 타입 11-11
가변 타입 11-11

12 장

프로그램 제어 12-1
매개변수 및 함수 결과 12-1
매개변수 전달 12-1
레지스터 저장 규칙 12-3
함수 결과값 12-3
메소드 호출 12-3
생성자 및 소멸자 12-4
종료 프로시저 12-4

13 장

인라인 어셈블러 코드 13-1
asm 문 13-1
레지스터 사용 13-2
어셈블러 문 구문 13-2
레이블 13-2
명령 연산 코드 13-2
RET 명령 크기 조정 13-3
자동 점프 크기 조정 13-3
어셈블러 지시어 13-3
피연산자 13-5
표현식 13-6
오브젝트 파스칼과 어셈블러 표현식의 차이 13-6
표현식 요소 13-7
상수 13-7
레지스터 13-8
기호 13-9
표현식 분류 13-11
표현식 타입 13-12
표현식 연산자 13-13
어셈블러 프로시저 및 함수 13-14

부록 A

오브젝트 파스칼 문법 A-1
색인 I-1

표

4.1 예약어 4-3
4.2 지시어 4-3
4.3 이항 산술 연산자 4-6
4.4 단항 산술 연산자 4-7
4.5 부울 연산자 4-7
4.6 논리 (비트 단위) 연산자 4-8
4.7 문자열 연산자 4-9
4.8 문자 – 포인터 연산자 4-9
4.9 집합 연산자 4-10
4.10 관계 연산자 4-11
4.11 연산자 우선 순위 4-12
5.1 오브젝트 파스칼 32 비트 구현의 일반적인 정수 타입 5-3
5.2 기본 정수 타입 5-4
5.3 기본 실수 타입 5-9
5.4 일반적인 실수 타입 5-10
5.5 문자열 타입 5-10
5.6 시스템과 SysUtils 에서 선언된 선택된포인터 타입 5-27
5.7 가변 타입 변환 규칙 5-32
5.8 정수 상수의 타입 5-40
6.1 호출 규칙 6-5
8.1 입력과 출력 프로시저 및 함수 8-1
8.2 Null 종료 문자열 함수 8-6
8.3 기타 표준 루틴 8-7
9.1 컴파일된 패키지 파일 9-11
9.2 패키지 특정 컴파일러 지시어 9-11
9.3 패키지 특정 명령줄 컴파일러 스위치 . 9-12
11.1 긴 문자열 동적 메모리 레이아웃 . . . 11-6
11.2 와이드 문자열 동적 메모리 레이아웃 (Windows 만 해당) 11-6
11.3 동적 배열 메모리 레이아웃 11-7
11.4 타입 정렬 마스크 11-8
11.5 가상 메소드 테이블 레이아웃 11-10
13.1 기본 제공 어셈블러 예약어 13-5
13.2 문자열 예제 및 문자열 값 13-8
13.3 CPU 레지스터 13-8
13.4 기본 제공 어셈블러가 인식하는 기호 . 13-9
13.5 이미 정의된 타입 기호 13-12
13.6 기본 제공 어셈블러 표현식 연산자의 우선 순위 13-13
13.7 기본 제공 어셈블러 표현식 연산자의 정의 13-13

1

서문

이 설명서에서는 Borland 개발 도구에서 사용되는 오브젝트 파스칼 프로그래밍 랭귀지에 대해 설명합니다.

설명서 내용

7장까지는 일반적인 프로그래밍에 사용되는 랭귀지 요소에 대해 설명합니다. 8장에서는 파일 I/O 및 문자열을 처리하는 표준 루틴에 대해 요약 설명합니다.

9장에서는 DLL과 패키지에 대한 랭귀지 확장 및 제한 사항에 대해 설명하고 10장에서는 객체 인터페이스에 대한 랭귀지 확장 및 제한 사항에 대해 설명합니다. 마지막 세 장에서는 다음과 같은 보다 구체적인 주제들을 다루고 있습니다. 즉, 11장에서는 메모리 관리, 12장에서는 프로그램 제어, 13장에서는 오브젝트 파스칼 프로그램 내의 어셈블리어에 대해 설명합니다.

오브젝트 파스칼 사용

*오브젝트 파스칼 랭귀지 안내서*에서는 Linux나 Windows 운영 체제에서 사용하는 오브젝트 파스칼 랭귀지에 대해 설명합니다. 랭귀지에서 특정 플랫폼에 따른 차이점은 필요할 때마다 언급할 것입니다.

대부분의 Delphi/Kylix 애플리케이션 개발자들은 통합 개발 환경(IDE)에서 오브젝트 파스칼 코드를 작성하고 컴파일합니다. IDE에서 작업하면 유닛 간 종속 정보 유지 관리와 같은 프로젝트 및 소스 파일 설정에 대한 많은 세부 사항을 처리할 수 있습니다. Borland 제품은 엄밀히 말해서 오브젝트 파스칼 랭귀지 사양의 일부가 아닌 프로그램 구성에 대해 특정 제약 조건을 강제로 적용할 수 있습니다. 예를 들어, IDE 외부에서 프로그램을 작성하고 명령 프롬프트에서 컴파일하는 경우, 특정 파일 및 프로그램의 이름 지정 규칙을 따르지 않을 수 있습니다.

이 설명서에서는 일반적으로 사용자가 IDE에서 작업하고 있으며 Visual Component Library (VCL) 또는 크로스 플랫폼용 컴포넌트 라이브러리(CLX)를 사용하는 애플리케이션을 구축하고 있다고 가정합니다. 하지만, 경우에 따라서는 Borland 특정 규칙이 모든 오브젝트 파스칼 프로그래밍에 적용되는 규칙과 다를 수도 있습니다.

표기법

상수, 변수, 타입, 필드, 속성, 프로시저, 함수, 프로그램, 유닛, 라이브러리 및 패키지의 이름 같은 식별자는 텍스트에서 *이탤릭체*로 표시됩니다. 오브젝트 파스칼 연산자, 예약어 및 지시어는 **굵게** 표시됩니다. 파일이나 명령 프롬프트에서 입력한 예제 코드 및 텍스트는 고정 폭으로 표시됩니다.

다음에 표시된 프로그램 목록에서는 예약어와 지시어가 마치 텍스트에서 표시되는 것처럼 **굵게** 표시됩니다.

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

Syntax Highlight 옵션이 선택된 경우 코드 에디터에서 예약어와 지시어를 이와 같이 나타냅니다.
위의 예제처럼 일부 프로그램에는 생략 기호 (... 또는 :) 가 포함되어 있습니다 . 생략 기호는 실제 파일에 코드가 추가될 수 있는 부분임을 나타냅니다 . 이 기호를 문자 그대로 복사하지 마십시오 .

구문 설명에서 *이탤릭체*로 표시된 부분은 실제 코드에서 개발자가 유효한 구문으로 대체 해야 합니다. 예를 들어, 위의 함수 선언의 헤더는 다음과 같이 나타낼 수 있습니다.

```
function functionName(argumentList): returnType,
```

또한 구문 설명은 다음과 같이 생략 기호(...)와 아래 첨자를 포함할 수도 있습니다.

```
function functionName(arg1, ..., argn): ReturnType;
```

추가 정보

개발 도구의 온라인 도움말 시스템은 VCL 또는 CLX의 최신 참조 정보 뿐만 아니라 IDE와 사용자 인터페이스에 대한 정보를 제공합니다. 데이터베이스 프로그램 개발 같은 많은 프로그래밍 항목들은 *개발자 안내서*에서 깊이 있게 다룹니다. 설명서의 개요에 관해서는 소프트웨어 패키지에 제공되는 *입문서*를 참조하십시오.

소프트웨어 등록 및 기술 지원

Borland Software Corporation은 각 개발자, 컨설턴트 및 기업의 요구를 만족시키기 위하여 일정한 지원 계획을 제공합니다. 이 제품과 관련해서 도움을 받으려면, 등록 카드를 보내시고 사용자의 필요에 따라 적절한 계획을 세우십시오. 기술 지원 및 기타 Borland 서비스에 대한 추가 정보는 각국 영업 부서에 문의하거나 http://www.borland.com/에 방문하십시오.

I

기본 랭귀지 설명

1부에서는 대부분의 프로그래밍 작업에 필요한 기본적인 랭귀지 요소를 다룹니다. 각 장에는 다음과 같은 내용이 포함되어 있습니다.

- 2장 "개요"
- 3장 "프로그램 및 유닛"
- 4장 "구문 요소"
- 5장 "데이터 타입, 변수 및 상수"
- 6장 "프로시저 및 함수"
- 7장 "클래스 및 객체"
- 8장 "표준 루틴 및 I/O"

2

개요

오브젝트 파스칼은 고급 랭귀지이며, 컴파일 할 수 있고, 타입이 엄격하게 지정된 랭귀지로, 구조적 설계(Structured Design)와 객체 지향 설계(Object-oriented Design)를 지원합니다. 오브젝트 파스칼의 코드는 읽기 쉽고, 컴파일이 빠르고, 모듈 프로그래밍을 위해 유닛 파일을 여러 개 사용할 수 있는 장점이 있습니다.

오브젝트 파스칼은 Borland의 컴포넌트 프레임워크와 RAD 환경을 지원하는 특수한 기능이 있습니다. 대부분의 경우 이 설명서의 설명과 예제는 Delphi나 Kylix 같은 Borland 개발 도구를 사용하여 애플리케이션을 개발하는데 오브젝트 파스칼을 사용한다고 가정합니다.

프로그램 구성

프로그램은 일반적으로 *유닛*이라는 여러 개의 소스 코드 모듈로 나뉘어져 있습니다. 각 프로그램은 프로그램의 이름을 지정하는 헤더로 시작합니다. 헤더 다음에는 옵션인 **uses** 절이 오고, 그 뒤에 선언과 문장 블록이 옵니다. **uses** 절은 이 프로그램에 연결되어 있는 유닛을 나열하는데, 이 유닛은 다른 프로그램과 공유하며, 자신을 **uses** 절에 지정하는 경우도 있습니다.

uses 절은 컴파일러에게 모듈 간의 종속 관계에 관한 정보를 알려줍니다. 이러한 종속 관계에 대한 정보가 모듈 자체에 저장되기 때문에 오브젝트 파스칼 프로그램은 makefile, 헤더 파일, "include" 지시어가 필요없습니다. Project Manager는 프로젝트가 IDE에 로드될 때마다 makefile을 만들지만, 두 개 이상의 프로젝트가 포함된 프로젝트 그룹일 경우에만 이 파일을 저장합니다.

프로그램 구조와 종속 관계에 대한 자세한 내용은 3장 "프로그램 및 유닛"을 참조하십시오.

파스칼 소스 파일

컴파일러는 다음과 같은 세 종류의 파일에서 파스칼 소스 코드를 찾습니다.

- *유닛* 소스 파일 – 확장자가 .pas인 파일
- *프로젝트* 파일 – 확장자가 .dpr인 파일
- *패키지* 소스 파일 – 확장자가 .dpk인 파일

유닛 소스 파일에는 애플리케이션에 있는 대부분의 코드가 포함되어 있습니다. 각 애플리케이션은 프로젝트 파일 한 개와 유닛 파일 여러 개를 갖는데, 전통적인 파스칼에서 "메인" 프로그램 파일에 해당하는 프로젝트 파일은 여러 개의 유닛 파일을 하나의 애플리케이션으로 만듭니다. Borland 개발 도구는 자동으로 각 애플리케이션에 대해 프로젝트 파일을 하나만 유지합니다.

명령줄에서 프로그램을 컴파일할 경우, 모든 소스 코드를 유닛(.pas) 파일에 넣을 수 있습니다. 하지만 IDE를 사용하여 애플리케이션을 빌드할 경우에는 반드시 프로젝트(.dpr) 파일이 있어야 합니다.

패키지 소스 파일은 프로젝트 파일과 유사하긴 하지만, *패키지*라고 하는 특수한 DLL을 구성할 경우 사용됩니다. 패키지에 대한 자세한 내용은 9장 "라이브러리 및 패키지"를 참조하십시오.

애플리케이션을 빌드하는데 사용되는 다른 파일

소스 코드 모듈 이외에 Borland 제품은 애플리케이션을 빌드하는데 파스칼이 아닌 파일을 몇 가지 사용합니다. 이러한 파일들은 자동으로 유지 관리되며 이 파일들은 다음과 같습니다.

- *폼* 파일 – 확장자가.dfm (Delphi) 또는 .xfm (Kylix)인 파일
- *리*소스 파일 – 확장자가 .res인 파일
- *프로젝트 옵션* 파일 – 확장자가 .dof (Delphi) 또는 .kof (Kylix)인 파일

폼 파일은 텍스트 파일이거나 비트맵, 문자열 등이 포함되어 있는 컴파일된 리소스 파일입니다. 각 폼 파일은 단일 폼을 나타내는데, 일반적으로 애플리케이션에서 창이나 대화 상자에 해당합니다. IDE에서는 폼 파일을 텍스트로 보고 편집할 수 있으며, 텍스트 파일이나 바이너리 파일로 저장할 수 있습니다. 기본적으로 폼 파일을 텍스트로 저장하기는 하지만 일반적으로 폼 파일은 수동으로 편집하지 않기 때문에, 편집을 하려면 Borland의 비주얼 디자인 툴을 사용하는 것이 좋습니다. 각 프로젝트는 최소 하나의 폼을 가지며, 각 폼은 기본적으로 폼 파일과 동일한 이름을 갖는 유닛(.pas) 파일에 연결되어 있습니다.

폼 파일 이외에 각 프로젝트는 리소스(.res) 파일을 사용하여 애플리케이션 아이콘의 비트맵을 저장합니다. 기본적으로 이 파일은 프로젝트(.dpr) 파일과 동일한 이름을 가집니다. 애플리케이션 아이콘을 변경하려면 Project Options 대화 상자를 사용합니다.

프로젝트 옵션(.dof 또는 .kof) 파일에는 컴파일러와 링커 설정, 검색 디렉토리, 버전 정보 등이 포함되어 있습니다. 각 프로젝트에는 프로젝트(.dpr) 파일과 동일한 이름을 가진 연결된 프로젝트 파일이 있습니다. 일반적으로 이 파일의 옵션은 Project Options 대화 상자에서 설정합니다.

IDE의 다양한 도구들은 데이터를 다른 형식의 파일에 저장합니다. 데스크탑 설정(.dsk 또는 .desk) 파일에는 창 정렬과 기타 구성 옵션이 포함되어 있는데, 데스크탑 설정을 특정 프로젝트에만 적용하거나 환경 전체에 적용할 수도 있습니다. 이 파일들은 컴파일에는 직접적인 영향을 미치지 않습니다.

컴파일러 생성 파일

애플리케이션이나 표준 DLL을 처음으로 빌드하면 컴파일러는 프로젝트에 사용된 각각의 새 유닛에 대해 컴파일된 유닛 .dcu (Windows) .dcu/.dpu (Linux) 파일을 만듭니다. 이 때 프로젝트에 있는 모든 .dcu (Windows) .dcu/.dpu (Linux) 파일이 연결되어 실행 파일 또는 공유 라이브러리 파일을 하나 만듭니다. 패키지를 처음으로 빌드하면 컴파일러는 패키지에 포함된 각각의 새 유닛에 대해 .dcu (Windows) .dpu (Linux) 파일을 만든 다음 .dcp와 패키지 파일을 만듭니다. (라이브러리와 패키지에 대한 자세한 내용은 9장을 참조하십시오.) **-GD** 스위치를 사용할 경우, 링커는 맵 파일과 .drc 파일을 만듭니다. 문자열 리소스를 포함하는 이 .drc 파일은 컴파일하여 리소스 파일로 만들 수 있습니다.

프로젝트를 다시 빌드할 때, 마지막 컴파일 이후 소스(.pas) 파일이 변경되지 않았거나 유닛의 .dcu (Windows) .dcu/.dpu (Linux) 파일이 없거나, 사용자가 명시적으로 유닛을 다시 처리하도록 지정하지 않는 한, 각각의 유닛은 다시 컴파일되지 않습니다. 실제로 컴파일러가 컴파일된 유닛 파일을 찾을 수 있는 경우에는 유닛의 소스 파일이 제공될 필요는 없습니다.

예제 프로그램

다음 예제는 오브젝트 파스칼 프로그래밍의 기본적인 특징을 보여줍니다. 이 예제는 IDE에서 컴파일 할 수 없는 간단한 오브젝트 파스칼 애플리케이션을 보여줍니다. 명령줄에서 이 코드를 컴파일할 수 있습니다.

간단한 콘솔 애플리케이션

아래의 프로그램은 명령 프롬프트에서 컴파일하고 실행할 수 있는 간단한 콘솔 애플리케이션입니다.

```
program Greeting;

{$APPTYPE CONSOLE}

var MyMessage:string;

begin
  MyMessage := 'Hello world!';
  Writeln(MyMessage);
end.
```

첫 번째 행은 *Greeting*이라는 프로그램을 선언합니다. `{$APPTYPE CONSOLE}` 지시어는 컴파일러에게 이것이 명령줄에서 실행할 콘솔 애플리케이션임을 알려줍니다. 다음 행은 *MyMessage*라는 변수를 선언하는데, 이 변수는 문자열 변수입니다. (오브젝트 파스칼

은 고유한 문자열 데이터 타입을 가집니다.) 그런 다음 *MyMessage* 변수에 "Hello world!"라는 문자열을 지정하고, *Writeln* 프로시저를 사용하여 *MyMessage*의 내용을 표준 출력으로 보냅니다. (*Writeln*은 암시적으로 *시스템* 유닛에 정의되어 있으며, 컴파일러는 모든 애플리케이션에 이 유닛을 자동으로 포함시킵니다.)

사용자는 이 프로그램을 Greeting.pas 또는 Greeting.dpr이라는 파일로 저장할 수 있고 명령줄에 다음과 같이 입력하여 파일을 컴파일할 수 있습니다.

Delphi 인 경우:
```
DCC32 Greeting
```

Kylix 인 경우:
```
dcc Greeting
```

실행 결과로 "Hello world!"라는 메시지를 인쇄합니다.

이 예제는 간단하다는 점 외에도 Borland 개발 도구를 사용하여 작성하는 프로그램과는 몇 가지 중요한 점에서 차이가 있습니다. 첫째, 이 예제는 콘솔 애플리케이션입니다. 일반적으로 Borland 개발 도구는 그래픽 인터페이스가 있는 애플리케이션을 작성하는 데 사용되기 때문에 대개의 경우 *Writeln*을 호출하지 않습니다. 게다가 전체 예제 프로그램(시스템 유닛에 포함되는 *Writeln*은 제외)은 파일 하나로 되어 있습니다. 일반적인 애플리케이션에서는 유닛 파일에 정의된 *메*소드에 대한 몇 개의 호출을 제외한 실제 애플리케이션 로직이 없는 다른 프로젝트 파일에 예제의 첫 번째 줄에 있는 프로그램 헤더가 있습니다.

좀 더 복잡한 예제

다음 예제는 *프로젝트* 파일과 *유닛* 파일로 나뉘어진 프로그램을 보여줍니다. Greeting.dpr로 저장할 프로젝트 파일은 다음과 같습니다.

```
program Greeting;

{$APPTYPE CONSOLE}

uses Unit1;

begin
  PrintMessage('Hello World!');
end.
```

첫 번째 줄은 *Greeting*, 이라는 프로그램을 선언하는데, 이것 역시 콘솔 애플리케이션입니다. `uses Unit1;` 절은 컴파일러에게 *Greeting*이 *Unit1*이라는 유닛을 포함한다는 것을 알려줍니다. 마지막으로 *PrintMessage* 프로시저를 호출하여 "Hello World!" 문자열을 전달합니다. *PrintMessage* 프로시저는 *Unit1*에 정의되어 있습니다. 다음은 *Unit1*의 소스 코드로, Unit1.pas라는 파일에 저장합니다.

```
unit Unit1;

interface

procedure PrintMessage(msg:string);

implementation

procedure PrintMessage(msg:string);
```

```
begin
  Writeln(msg);
end;

end.
```

*Unit1*은 단일 문자열을 인수로 가지고 있고 이 문자열을 표준 출력으로 보내는 *PrintMessage* 라는 프로시저를 정의합니다. 파스칼에서는 값을 반환하지 않는 루틴을 *프로시저*라고 합니다. 값을 반환하는 루틴은 *함수*라고 합니다. *PrintMessage*가 *Unit1*에서 두 번 선언된 점에 유의하십시오. **interface** 예약어 아래에 있는 첫 번째 선언은 *Greeting* 같이 *Unit1*을 사용하는 다른 모듈에서 *PrintMessage*를 호출할 수 있도록 합니다. **implementation** 예약어 아래에 있는 두 번째 선언은 실제로 *PrintMessage*를 정의합니다.

이제 명령줄에 다음과 같이 입력하여 *Greeting*을 컴파일할 수 있습니다.

```
Delphi 인 경우 :DCC32 Greeting

Kylix 인 경우 :dcc Greeting
```

명령줄 인수로 *Unit1*을 포함할 필요는 없습니다. 컴파일러가 Greeting.dpr을 처리할 때 *Greeting* 프로그램에서 필요로 하는 유닛 파일을 자동으로 찾습니다. 실행 결과는 첫 번째 예제와 같이 "Hello world!" 메시지를 출력합니다.

윈시 애플리케이션

다음 예제는 IDE에서 VCL 또는 CLX 컴포넌트를 사용하여 빌드한 애플리케이션입니다. 이 프로그램은 자동으로 생성된 폼과 리소스 파일을 사용하므로 사용자가 소스 코드에서 단독으로 컴파일할 수 없습니다. 이 프로그램은 오브젝트 파스칼의 중요한 특징을 보여줍니다. 이 프로그램은 여러 개의 유닛은 물론 7장 "클래스 및 객체"에서 설명하는 클래스와 객체도 사용합니다.

이 프로그램에는 프로젝트 파일과 두 개의 새 유닛 파일이 들어 있습니다. 프로젝트 파일은 다음과 같습니다.

```
program Greeting;  { comments are enclosed in braces }

uses
  Forms, {change the unit name to QForms on Linux}
  Unit1 in 'Unit1.pas' { the unit for Form1 },
  Unit2 in 'Unit2.pas' { the unit for Form2 };

{$R *.res}  { this directive links the project's resource file }

begin
  { calls to Application }
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

이 프로그램 이름은 *Greeting*입니다. 이 프로그램은 VCL과 CLX의 일부인 *Forms*, 애플리케이션의 메인 폼 *Form1*과 연결되어 있는 *Unit1*, 그리고 폼 *Form2*와 연결되어 있는 *Unit2* 유닛을 사용합니다.

이 프로그램은 *Forms* 유닛에 정의된 *TApplication* 클래스의 인스턴스인 *Application* 객체를 연속해서 호출합니다. (모든 프로젝트에는 자동으로 생성된 *Application* 객체가 있습니다.) *CreateForm*이라는 *TApplication* 메소드를 두 번 호출하는데, *CreateForm*에 대한 첫 번째 호출은 *Unit1*에서 정의한 *TForm1* 클래스의 인스턴스인 *Form1*을 만듭니다. *CreateForm* 에 대한 두 번째 호출은 *Unit2*에서 정의한 *TForm2* 클래스의 인스턴스인 *Form2*를 만듭니다.

*Unit1*은 다음과 같습니다.

```
unit Unit1;

interface

uses  { these units are part of the Visual Component Library (VCL) }
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

{
On Linux, the uses clause looks like this:
uses  { these units are part of CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}

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

var
  Form1: TForm1;

implementation

uses Unit2;  { this is where Form2 is defined }

{$R *.dfm}  { this directive links Unit1's form file }

procedure TForm1.Button1Click(Sender:TObject);
begin
  Form2.ShowModal;
end;

end.
```

*Unit1*은 *TForm*에서 파생된 *TForm1*이라는 클래스와 이 클래스의 인스턴스인 *Form1*을 만듭니다. *TForm1*에는 *TButton*의 인스턴스인 *Button1* 버튼과 런타임 시 사용자가 *Button1*을 클릭할 때마다 호출되는 *TForm1.Button1Click*이라는 프로시저가 포함되어 있습니다. *TForm1.Button1Click*은 *Form1*을 숨기고 *Form2.ShowModal*을 호출하여 *Form2*를 표시합니다. *Form2*는 *Unit2*에 다음과 같이 정의되어 있습니다.

```
unit Unit2;

interface

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

{
On Linux, the uses clause looks like this:
uses  { these units are part of CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}
type
  TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton:TButton;
    procedure CancelButtonClick(Sender:TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;

var
  Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender:TObject);
begin
  Form2.Close;
end;

end.
```

*Unit2*는 *TForm2*라는 클래스와 이 클래스의 인스턴스인 *Form2*를 만듭니다. *TForm2*에는 *TButton*의 인스턴스인 *CancelButton* 버튼과 *TLabel*의 인스턴스인 *Label1* 레이블이 포함되어 있습니다. 소스 코드에서 볼 수 없지만 *Label1*은 "Hello world!"라는 캡션을 표시합니다. 이 캡션은 *Form2*의 폼 파일 Unit2.dfm에 정의되어 있습니다.

*Unit2*는 프로시저를 하나 정의합니다. *TForm2.CancelButtonClick*은 런타임 시 사용자가 *CancelButton*을 클릭할 때마다 호출되어 *Form2*를 닫습니다. *Unit1*의 *TForm1.Button1Click*을 비롯한 이 프로시저는 프로그램이 실행되는 동안 발생하는 이벤트에 응답하기 때문에 이를 *이벤트 핸들러*라고 합니다. *Form1*과 *Form2*에 대한 폼(Windows인 경우 .dfm, Linux인 경우 .xfm) 파일에서 특정 이벤트에 이벤트 핸들러를 지정합니다.

Greeting 프로그램을 실행하면 *Form1*이 표시되고 *Form2*는 보이지 않습니다. 기본적으로, 런타임 시 프로젝트 파일에서 만든 첫 번째 폼만 표시됩니다. 이 폼을 프로젝트의 *메인 폼*이라고 합니다. 사용자가 *Form1*에 있는 버튼을 누르면 *Form1*이 사라지고 "Hello world!"라는 인사말이 *Form2*에 나타납니다. *CancelButton*이나 제목 표시줄의 Close 버튼을 클릭하여 *Form2*를 닫으면 *Form1*이 다시 나타납니다.

3

프로그램 및 유닛

프로그램은 *유닛*이라고 하는 소스 코드 모듈로 구성되어 있습니다. 각 유닛은 각각의 파일로 저장되고 별도로 컴파일되며, 컴파일된 유닛이 링크되어 애플리케이션이 만들어집니다. 유닛을 사용하면 다음과 같은 작업을 할 수 있습니다.

- 큰 프로그램을 별도로 편집할 수 있는 모듈로 나눌 수 있습니다.
- 프로그램 간에 공유할 수 있는 라이브러리를 만들 수 있습니다.
- 소스 코드 없이도, 다른 개발자에게 라이브러리를 배포할 수 있습니다.

종래의 파스칼 프로그래밍에서는 메인 프로그램을 비롯한 모든 소스 코드를 .pas 파일에 저장했습니다. 그런데 Borland 툴은 *프로젝트*(.dpr) 파일을 사용하여 "메인" 프로그램을 저장하고, 나머지 소스 코드 대부분은 *유닛*(.pas) 파일에 저장합니다. 각 애플리케이션 또는 *프로젝트*는 한 개의 프로젝트 파일과 하나 이상의 유닛 파일로 구성됩니다. 엄밀히 말해서 프로젝트의 어떠한 유닛을 명시적으로 사용할 필요는 없지만, 모든 프로그램은 자동적으로 *시스템* 유닛을 사용합니다. 컴파일러는 프로젝트를 빌드하기 위해 각 유닛에 대한 소스 파일이나 컴파일된 유닛 파일을 필요로 합니다.

프로그램 구조 및 구문

프로그램에는 다음과 같은 내용이 들어 있습니다.

- 프로그램 헤더
- **uses** 절(옵션)
- 선언과 문장 블록

프로그램 이름은 프로그램 헤더에 할당합니다. **uses**절은 프로그램에서 사용하는 유닛을 나열합니다. 블록에는 프로그램을 실행할 때 실행되는 선언과 문장이 들어 있습니다. IDE는 단일 프로젝트(.dpr) 파일에서 이러한 세 가지 요소를 찾습니다.

아래 예제는 Editor라는 프로그램의 프로젝트 파일입니다.

```
1     program Editor;
2
3     uses
4       Forms, {change to QForms in Linux}
5       REAbout in 'REAbout.pas' {AboutBox},
6       REMain in 'REMain.pas' {MainForm};
7
8     {$R *.res}
9
10    begin
11      Application.Title := 'Text Editor';
12      Application.CreateForm(TMainForm, MainForm);
13      Application.Run;
14    end.
```

1행에는 프로그램 헤더가 있습니다. **uses** 절은 3행~6행에 있습니다. 8행은 프로젝트의 리소스 파일을 프로그램에 연결하는 컴파일러 지시어입니다. 10행~14행에는 프로그램을 실행할 때 실행되는 문장의 블록이 있습니다. 마지막으로, 모든 소스 파일처럼 프로젝트 파일도 마침표로 끝납니다.

사실 이 파일은 아주 전형적인 프로젝트 파일입니다. 프로그램 로직 대부분을 유닛 파일에 정의하기 때문에, 프로젝트 파일은 대체로 길이가 짧습니다. 프로젝트 파일은 자동으로 생성되고 유지 관리되므로 직접 편집할 필요가 거의 없습니다.

프로그램 헤더

프로그램의 이름은 프로그램 헤더에 할당합니다. 프로그램 헤더는 예약어 **program**, 유효한 식별자, 세미콜론 순으로 구성됩니다. 식별자는 프로젝트 파일 이름과 일치해야 합니다. 위의 예제에서 프로그램 이름이 Editor이므로 프로젝트 파일 이름도 EDITOR.dpr이어야 합니다.

표준 파스칼에서 프로그램 헤더에는 프로그램 이름 뒤에 다음과 같은 매개변수가 올 수 있습니다.

```
program Calc(input, output);
```

Borland의 오브젝트 파스칼 컴파일러는 이 매개변수를 무시합니다.

프로그램 uses 절

uses 절은 프로그램에서 사용하는 유닛을 나열합니다. 이러한 유닛들은 자신의 **uses** 절을 가질 수 있습니다. **uses** 절에 대한 자세한 내용은 3-5 페이지의 "유닛 참조 및 uses 절"을 참조하십시오.

블록

블록에는 프로그램을 실행할 때 실행되는 일반문 또는 구조문이 있습니다. 대부분의 프로그램에서 블록은 예약어 **begin**과 **end** 쌍으로 묶인 복합문으로 구성됩니다. 이러한 복합문은 프로젝트의 *Application* 객체에 대한 단순한 메소드 호출를 수행합니다. 모든 프로젝트는 *TApplication*, *TWebApplication* 또는 *TServiceApplication*의 인스턴스를 유

지하는 *Application* 변수를 가지고 있습니다. 블록에는 상수, 타입, 변수, 프로시저 및 함수의 선언도 포함할 수 있습니다. 이러한 선언은 반드시 블록의 문장 부분보다 앞에 있어야 합니다.

유닛 구조 및 구문

유닛은 클래스를 포함한 타입, 상수, 변수 및 루틴(함수와 프로시저)으로 구성됩니다. 각 유닛은 자체 유닛(.pas) 파일에서 정의합니다.

유닛 파일은 유닛 헤더로 시작되고, 뒤에 *인터페이스*, *구현*, *초기화* 및 *완료* 섹션이 옵니다. 초기화 및 완료 섹션은 옵션입니다. 유닛 파일 구조는 다음과 같습니다.

```
unit Unit1;

interface

uses  { List of units goes here }

  { Interface section goes here }

implementation

uses  { List of units goes here }

  { Implementation section goes here }

initialization
  { Initialization section goes here }

finalization
  { Finalization section goes here }

end.
```

유닛은 반드시 마침표가 있는 **end**로 끝나야 합니다.

유닛 헤더

유닛의 이름은 유닛 헤더에 할당합니다. 유닛 헤더는 예약어 **unit**, 유효한 식별자, 세미콜론 순으로 구성됩니다. Borland 도구를 사용하여 개발된 애플리케이션인 경우 식별자는 반드시 유닛 파일 이름과 일치해야 합니다. 따라서 유닛 헤더는

```
unit MainForm;
```

MAINFORM.pas라는 이름의 소스 파일에서 발생하며, 컴파일된 유닛이 있는 파일 이름은 MAINFORM.dcu가 됩니다.

유닛 이름은 반드시 프로젝트 내에서 유일해야 합니다. 유닛 파일이 다른 디렉토리에 있더라도 동일한 이름을 가진 두 개의 유닛을 단일 프로그램에서 사용할 수 없습니다.

인터페이스 섹션

유닛의 인터페이스 섹션은 **interface** 예약어로 시작되고 구현 섹션이 시작될 때까지 계속됩니다. 인터페이스 섹션에는 *클라이언트* 즉, 선언된 위치에서 이 유닛을 사용하는 다른 유닛이나 프로그램이 사용할 수 있는 상수, 타입, 변수, 프로시저 및 함수를 선언합니다. 클라이언트가 마치 클라이언트 내에서 선언된 것처럼 이러한 엔티티에 액세스할 수 있기 때문에 이들 엔티티를 *public*이라고 합니다.

프로시저나 함수의 인터페이스 선언은 루틴의 헤더만 포함합니다. 프로시저나 함수의 블록은 구현 섹션에 옵니다. 그러므로 인터페이스 섹션의 프로시저와 함수 선언은 **forward** 지시어가 사용되지 않았음에도 불구하고 forward 선언처럼 사용합니다.

클래스의 인터페이스 선언은 모든 클래스 멤버의 선언을 포함해야 합니다.

인터페이스 섹션은 자체의 **uses** 절을 포함할 수 있는데, 반드시 **interface** 단어 바로 뒤에 표시해야 합니다. **uses** 절에 대한 자세한 내용은 3-5 페이지의 "유닛 참조 및 uses 절"을 참조하십시오.

구현 섹션

유닛의 구현 섹션은 **implementation** 예약어로 시작하고 초기화 섹션이 시작할 때까지 또는 초기화 섹션이 없는 경우에는 유닛의 끝까지 계속됩니다. 구현 섹션은 인터페이스 섹션에 선언된 프로시저와 함수를 정의합니다. 구현 섹션 내에서 이러한 프로시저와 함수는 순서에 상관없이 정의하고 호출할 수 있습니다. 구현 섹션에 public 프로시저와 함수 헤더를 정의할 때 매개변수 목록을 생략할 수 있습니다. 그러나 매개변수 목록을 포함할 경우에는 인터페이스 섹션의 선언과 반드시 일치해야 합니다.

구현 섹션은 public 프로시저와 함수의 정의 외에도 클라이언트에서 액세스할 수 없는 유닛에 대해 *private*으로 상수, 클래스를 포함한 타입, 변수, 프로시저 및 함수를 선언할 수 있습니다.

구현 섹션은 자체의 **uses** 절을 포함할 수 있는데, 반드시 **implementation** 바로 뒤에 표시해야 합니다. **uses** 절에 대한 자세한 내용은 3-5 페이지의 "유닛 참조 및 uses 절"을 참조하십시오.

초기화 섹션

초기화 섹션은 옵션입니다. 초기화 섹션은 **initialization** 예약어로 시작하고 완료 섹션이 시작될 때까지 또는 완료 섹션이 없는 경우에는 유닛의 끝까지 계속됩니다. 초기화 섹션에는 프로그램 시작 시 나타나는 순서대로 실행될 문장이 포함되어 있습니다. 그러므로 예를 들면 초기화해야 할 정의된 데이터 구조를 갖고 있을 경우 초기화 섹션에서 정의할 수 있습니다.

클라이언트에서 사용하는 유닛의 초기화 섹션은 유닛이 클라이언트의 **uses** 절에 표시되는 순서대로 실행됩니다.

완료 섹션

완료 섹션은 옵션이며 초기화 섹션이 있는 유닛에서만 사용할 수 있습니다. 완료 섹션은 **finalization** 예약어로 시작하고 유닛의 끝까지 계속됩니다. 완료 섹션에는 메인 프로그램이 종료될 때 실행될 문장이 들어 있습니다. 완료 섹션을 사용하면 초기화 섹션에 할당되어 있는 리소스를 해제할 수 있습니다.

완료 섹션은 초기화 섹션과 반대로 실행됩니다. 예를 들어 애플리케이션에서 유닛을 *A*, *B*, *C* 순으로 초기화했다면 *C*, *B*, *A* 순으로 유닛을 해제합니다.

일단 어떤 유닛의 초기화 코드가 실행되기 시작하면, 애플리케이션 종료 시에 해당 종료 섹션이 실행됩니다. 런타임 오류가 발생하면 초기화 코드가 완전하게 실행되지 않을 수도 있기 때문에, 완료 섹션은 불완전하게 초기화된 데이터도 처리할 수 있어야 합니다.

유닛 참조 및 uses 절

uses 절은 이 절이 들어 있는 프로그램, 라이브러리 또는 유닛에서 사용하는 유닛을 나열합니다. (라이브러리에 대한 자세한 내용은 9장 "라이브러리 및 패키지"를 참조하십시오.) **uses** 절을 사용할 수 있는 곳은 다음과 같습니다.

- 프로그램이나 라이브러리의 프로젝트 파일.
- 유닛의 인터페이스 섹션
- 유닛의 구현 섹션

유닛의 인터페이스 섹션이 대부분 그러한 것처럼, 대부분의 프로젝트 파일도 **uses** 절을 갖고 있습니다. 유닛의 구현 섹션 또한 자신의 **uses** 절을 포함할 수 있습니다.

시스템 유닛은 모든 애플리케이션이 자동으로 사용하며 **uses** 절에 명시적으로 나열할 수 없습니다. (*시스템*은 파일 I/O, 문자열 처리, 부동 소수점 연산, 동적 메모리 할당 등에 대한 루틴을 구현합니다.) *SysUtils*와 같은 다른 표준 라이브러리 유닛은 반드시 **uses** 절에 포함해야 합니다. 대부분의 경우 프로젝트에서 소스 파일을 생성하고 유지할 때 필요한 모든 유닛은 **uses** 절에 있습니다.

uses 절의 위치와 내용에 대한 자세한 내용은 3-6 페이지의 "다중 및 간접 유닛 참조"와 3-7 페이지의 "순환 유닛 참조"를 참조하십시오.

uses 절의 구문

uses 절은 예약어 **uses**, 하나 이상의 쉼표로 구분된 유닛 이름, 세미콜론 순으로 구성됩니다.예를 들면, 다음과 같습니다.

```
uses Forms, Main;
uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;
uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

프로그램이나 라이브러리의 **uses** 절에서 유닛 이름 뒤에 예약어 **in**이 오고, 디렉토리 경로의 유무와 상관없이 소스 파일 이름이 작은 따옴표로 묶여 있습니다. 디렉토리 경로를 표시할 경우에는 절대 경로나 상대 경로를 표시할 수 있습니다. 예를 들면, 다음과 같습니다.

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;
```

```
uses
  QForms,
  Main,
  Extra in '../extra/extra.pas';
```

유닛의 소스 파일을 할당할 필요가 있을 경우, 유닛 이름 뒤에 **in** ...을 사용합니다. IDE에서는 유닛 이름이 유닛이 있는 소스 파일의 이름과 일치해야 하므로 이렇게 할 필요는 없습니다. **in**을 사용하는 것은 소스 파일의 위치가 명확하지 않을 경우에만 필요합니다. 예를 들면 다음과 같은 경우입니다

- 프로젝트 파일과 다른 디렉토리에 있는 소스 파일을 사용했는데, 그 디렉토리가 컴파일러의 검색 경로나 일반 라이브러리 검색 경로에 없는 경우.
- 컴파일러의 검색 경로에 있는 다른 디렉토리에 이름이 같은 유닛이 있을 경우.
- 명령줄에서 콘솔 애플리케이션을 컴파일할 때 유닛의 소스 파일 이름과 일치하지 않는 식별자로 유닛 이름을 할당했을 경우.

또한 컴파일러는 어떤 유닛이 프로젝트를 구성하는 지 결정할 때 **in**...을 사용합니다. 프로젝트(.dpr) 파일의 **uses** 절에서 **in**과 파일 이름이 뒤에 표시되는 유닛만 프로젝트를 구성하는 유닛으로 간주되고, **uses** 절의 다른 유닛은 프로젝트에 속하지 않는 유닛으로 간주합니다. 이러한 차이가 컴파일에는 아무런 영향을 주지 않지만 Project Manager와 Project Browser 같은 IDE 툴에 영향을 줍니다.

유닛의 **uses** 절에서는 **in**을 사용하여 컴파일러에게 소스 파일의 위치를 알려 줄 수 없습니다. 모든 유닛은 반드시 컴파일러의 검색 경로, 일반적인 라이브러리 검색 경로 또는 유닛을 사용하는 유닛과 동일한 디렉토리에 있어야 합니다. 뿐만 아니라 유닛 이름이 유닛의 소스 파일 이름과 일치해야 합니다.

다중 및 간접 유닛 참조

uses 절에 표시되는 유닛의 순서는 유닛의 초기화 순서를 결정하고 (3-4 페이지의 "초기화 섹션" 참조) 컴파일러가 식별자를 인식하는 방법에 영향을 미칩니다. 두 개의 유닛이 동일한 이름을 가진 변수, 상수, 타입, 프로시저 또는 함수를 선언할 경우, 컴파일러는 **uses** 절의 마지막에 나열된 유닛의 식별자를 사용합니다. (다른 유닛의 식별자에 액세스하려면 *UnitName.Identifier* 처럼 한정자를 추가해야 합니다.)

uses 절에는 절이 표시되는 프로그램이나 유닛이 직접 사용하는 유닛만 포함시켜야 합니다. 다시 말해, 유닛 *A*가 유닛 *B*에 선언된 상수, 타입, 변수, 프로시저 또는 함수를 참조할 경우 *A*는 명시적으로 *B*를 사용해야 합니다. *B*가 유닛 *C*의 식별자를 참조할 경우에 *A*는 간접적으로 *C*에 종속됩니다. 이 경우 *C*를 *A*의 **uses** 절에 포함할 필요는 없지만 컴파일러가 *A*를 처리하려면 여전히 *B*와 *C*를 모두 찾을 수 있어야 합니다.

아래 예제는 간접 종속 관계를 보여 줍니다.

```
program Prog;
uses Unit2;
const a = b;
⋮
unit Unit2;
```

```
interface
uses Unit1;
const b = c;
⋮

unit Unit1;
interface
const c = 1;
⋮
```

이 예제에서 *Prog*는 *Unit2*에 직접적으로 종속되고, Unit2는 *Unit1*에 직접적으로 종속됩니다. 그러므로 *Prog*는 *Unit1*에 간접적으로 종속됩니다. *Unit1*은 *Prog*의 **uses** 절에 없기 때문에 *Unit1*에서 선언된 식별자는 *Prog*에서 사용할 수 없습니다.

컴파일러가 클라이언트 모듈을 컴파일하기 위해서는 클라이언트가 직접 또는 간접적으로 종속되어 있는 모든 유닛이 필요합니다. 그러나 이러한 유닛의 소스 코드를 변경하지 않았다면 컴파일러는 소스(.pas) 파일이 아닌 .dcu (Windows) 또는 .dcu/.dpu (Linux) 파일만 있으면 됩니다.

유닛의 인터페이스 섹션을 변경할 경우 이 유닛에 종속된 다른 유닛도 반드시 다시 컴파일해야 합니다. 하지만 유닛의 구현 섹션이나 다른 섹션만 변경할 경우에는 종속된 유닛을 다시 컴파일할 필요가 없습니다. 컴파일러가 이러한 종속 관계를 추적해서 필요할 때에만 유닛을 자동으로 다시 컴파일하기 때문입니다.

순환 유닛 참조

유닛이 서로 직접적으로나 간접적으로 참조할 경우 이러한 유닛을 상호 종속이라고 합니다. 상호 종속 관계는 어떤 인터페이스 섹션의 **uses** 절과 또 다른 인터페이스 섹션의 **uses** 절을 연결하는 순환 경로가 없을 경우에만 가능합니다. 다시 말해, 어떤 유닛의 인터페이스 섹션에서 시작하여, 다른 유닛의 인터페이스 섹션을 통해 참조를 따라 시작한 유닛으로 돌아오게 해서는 절대로 안됩니다. 상호 종속 패턴이 유효하려면 각 순환 참조 경로는 최소 하나 이상의 구현 섹션에서 **uses** 절을 사용해야 합니다.

가장 간단한 예로 두 개의 상호 종속 유닛이 있을 경우 유닛의 **uses** 절에는 두 개의 유닛을 서로 나열할 수 없다는 것을 의미합니다. 그러므로 다음 예제에서는 컴파일 오류가 발생합니다.

```
unit Unit1;
interface
uses Unit2;
⋮

unit Unit2;
interface
uses Unit1;
⋮
```

하지만 참조 중 하나를 구현 섹션으로 옮기면 두 유닛은 서로 참조할 수 있습니다.

```
unit Unit1;
interface
uses Unit2;
⋮
```

```
unit Unit2;
interface
⋮
implementation
uses Unit1;
⋮
```

순환 참조를 줄이려면 가능한 한 구현 **uses** 절에 유닛을 사용하는 것이 좋습니다. 다른 유닛의 식별자가 인터페이스 섹션에서 사용될 경우에만 인터페이스 **uses** 절에 유닛을 사용해야 합니다.

4

구문 요소

오브젝트 파스칼은 *A*부터 *Z*까지와 *a*부터 *z*까지의 영문자, *0*부터 *9*까지의 숫자 및 기타 표준 문자를 포함하는 ASCII 문자 집합을 사용합니다. 대/소문자는 구분하지 *않습니다*. 공백 문자(ASCII 32)와 제어 문자(리턴 또는 줄 끝 문자인 ASCII 13을 비롯한 ASCII 0부터 31까지)는 *공백(blank)*이라고 합니다.

기본 구문 요소들은 *토큰*이라고 부르며, 조합하여 표현식, 선언 및 문장을 만듭니다. *문장*은 프로그램 내에서 실행할 수 있는 알고리즘 동작을 나타냅니다. *표현식*은 문장에서 나타나는 구문 단위로서 값을 나타냅니다. *선언*은 표현식과 문장에서 사용할 수 있는 함수나 변수의 이름과 같은 식별자를 정의하고, 적절한 곳에 식별자에 대한 메모리를 할당합니다.

기본 구문 요소

가장 간단한 프로그램은 구분자에 의해 경계가 정해진 토큰을 연속적으로 배열한 것입니다. *토큰*은 프로그램 텍스트의 의미를 가진 가장 작은 단위입니다. *구분자*는 공백이거나 주석입니다. 엄밀히 말하자면, 두 개의 토큰 사이에 구분자를 항상 표시할 필요는 없습니다. 예를 들어,

```
Size:=20;Price:=10;
```

이 문장은 적합한 문장입니다. 그러나 편의성과 가독성을 높이기 위해서는 다음과 같이 코드를 작성해야 합니다.

```
Size:=20;
Price:=10;
```

토큰은 *특수 기호*, *식별자*, *예약어* , *지시어*, *숫자*, *레이블* 및 *문자열*의 범주로 나뉩니다. 토큰이 문자열인 경우에만 구분자가 토큰의 일부가 될 수 있습니다. 인접한 식별자, 예약어, 숫자 및 레이블 사이에 하나 이상의 구분자가 올 수 있습니다.

특수 기호

특수 기호는 영문자나 숫자를 제외한 고정된 의미를 가진 문자이거나 그런 문자들의 쌍입니다. 다음 단일 문자들은 특수 기호입니다.

 # $ & ' ( ) * + , - . / : ; < = > @ [ ] ^ { }

다음 문자 쌍 또한 특수 기호입니다.

(* (. *) .) .. // := <= >= <>

왼쪽 대괄호(**[**)는 왼쪽 괄호와 마침표 쌍(**(.**)과 동일하고, 오른쪽 대괄호(**]**)는 마침표와 오른쪽 괄호 쌍(**.)**)과 동일합니다. 왼쪽 괄호, 별표, 별표, 오른쪽 괄호(**(* *)**)는 왼쪽 및 오른쪽 중괄호(**{ }**)와 동일합니다.

!, **"** (큰 따옴표), **%**, **?**, ****, **_** (밑줄), **|** (파이프) 및 **~** (물결 표시)는 특수 기호가 아닙니다.

식별자

식별자는 상수, 변수, 필드, 타입, 속성, 프로시저, 함수, 프로그램, 유닛, 라이브러리 및 패키지를 나타냅니다. 식별자는 길이에 상관 없지만 처음 255개의 문자만 의미를 가집니다. 식별자는 문자나 밑줄(_)로 시작해야 하고 공백을 포함할 수 없습니다. 두 번째 문자부터는 문자, 숫자 및 밑줄이 올 수 있습니다. 예약어는 식별자로 사용할 수 없습니다.

오브젝트 파스칼에서는 대/소문자를 구분하지 않기 때문에 *CalculateValue*와 같은 식별자는 다음 중 어떤 방식으로도 쓸 수 있습니다.

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Linux에서 대/소문자를 구분하는 유일한 식별자는 유닛 이름입니다. 유닛 이름은 파일 이름에 해당하기 때문에 대/소문자가 일치하지 않으면 경우에 따라 컴파일에 영향을 줄 수도 있습니다.

한정된 식별자

둘 이상의 유닛에서 선언된 식별자를 사용할 때, 경우에 따라 식별자를 *한정*해야 할 수도 있습니다. 한정된 식별자에 대한 구문이 다음과 같은 경우

$identifier_1.identifier_2$

여기서, $identifier_1$은 $identifier_2$를 한정합니다. 예를 들어, 두 개의 유닛 각각이 *CurrentValue*라는 변수를 선언하는 경우 사용자는 다음과 같이 *Unit2*의 *CurrentValue*에 액세스를 지정할 수 있습니다.

```
Unit2.CurrentValue
```

한정자를 반복해서 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Form1.Button1.Click
```

이 문장은 *Form1*의 *Button1*에 있는 *Click* 메소드를 호출합니다.

식별자를 한정하지 않으면 4-27 페이지의 "블록과 유효 범위"에 설명한 범위 규칙에 따라 식별자를 해석합니다.

예약어

다음 예약어는 재정의하거나 식별자로 사용할 수 없습니다.

표 4.1 예약어

and	downto	in	or	string
array	else	inherited	out	then
as	end	initialization	packed	threadvar
asm	except	inline	procedure	to
begin	exports	interface	program	try
case	file	is	property	type
class	finalization	label	raise	unit
const	finally	library	record	until
constructor	for	mod	repeat	uses
destructor	function	nil	resourcestring	var
dispinterface	goto	not	set	while
div	if	object	shl	with
do	implementation	of	shr	xor

표 4.1에 있는 예약어 이외에 private, protected, public, published 및 automated는 객체 타입 선언 내에서 예약어로 사용되지만, 그 이외의 경우에는 지시어로 간주됩니다. at과 on도 특별한 의미를 가집니다.

지시어

지시어는 소스 코드 내의 특정 위치에서 의미를 가진 단어입니다. 지시어는 오브젝트 파스칼에서 특별한 의미를 갖지만 예약어와 달리 사용자 정의된 식별자가 나타날 수 없는 문맥에만 나타납니다. 따라서 권장할 만한 방법은 아니지만 지시어처럼 보이는 식별자를 사용자가 정의할 수 있습니다.

표 4.2 지시어

absolute	dynamic	message	private	resident
abstract	export	name	protected	safecall
assembler	external	near	public	stdcall
automated	far	nodefault	published	stored
cdecl	forward	overload	read	varargs
contains	implements	override	readonly	virtual
default	index	package	register	write
deprecated	library	pascal	reintroduce	writeonly
dispid	local	platform	requires	

숫자

정수 상수와 실수 상수는 쉼표나 공백이 없는 숫자로서 진법에 따라 나타낼 수 있습니다. 부호를 표시하려면 숫자 앞에 + 또는 − 연산자를 붙입니다. 값은 양수를 기본으로 하고 (예를 들어, 67258은 +67258과 동일), 선언된 타입에 따라 이미 정의된 값의 범위를 넘지 않아야 합니다.

소수점이나 지수가 있는 숫자는 실수를 나타내며 실수가 아닌 숫자는 정수를 나타냅니다. *E* 또는 *e* 문자가 실수에서 사용되면, "10의 거듭제곱"을 의미합니다. 예를 들어, 7E-2는 7×10^{-2}를 의미하고, 12.25e+6과 12.25e6은 모두 12.25×10^{6}을 의미합니다.

달러 기호가 접두사로 쓰이면 16진수를 나타냅니다. 예를 들면, $8F입니다. 16진수의 부호는 바이너리 데이터의 가장 왼쪽에 있는 비트(최상위 비트)에 의해서 결정됩니다.

실수 타입과 정수 타입에 대한 자세한 내용은 5장 "데이터 타입, 변수 및 상수"를 참조하십시오. 숫자의 데이터 타입에 대한 내용은 5−39 페이지의 "True 상수"를 참조하십시오.

레이블

레이블은 4자리 이하 숫자입니다. 즉, 0과 9999 사이의 숫자를 사용할 수 있습니다. 앞의 0은 의미가 없습니다. 식별자도 레이블로 사용할 수 있습니다.

레이블은 **goto** 문에서 사용됩니다. **goto** 문과 레이블에 대한 자세한 내용은 4−18 페이지의 "Goto 문"을 참조하십시오.

문자열

문자열 리터럴 또는 *문자열 상수*라고도 하는 문자열은 *인용 문자열*, *제어 문자열*, 또는 인용 문자열과 제어 문자열의 조합으로 구성됩니다. 구분자는 인용 문자열 내에서만 사용할 수 있습니다.

인용 문자열은 확장된 ASCII 문자 집합에서 최대 255개의 문자들로 구성되어 있고, 한 줄 길이이며, '로 묶여 있습니다. ' ' 사이에 아무것도 없는 인용 문자열은 *Null 문자열*입니다. 인용 문자열에 있는 두 개의 연속적인 '는 생략 기호라고 부르는 단일 문자를 나타냅니다. 예를 들면, 다음과 같습니다.

```
'BORLAND'          { BORLAND }
'You''ll see'      { You'll see }
''''               { ' }
''                 { Null string }
' '                { a space }
```

제어 문자열은 하나 이상의 *제어 문자*로 되어 있습니다. 각각의 제어 문자는 0에서부터 255까지의 부호없는 정수 상수(10진수 또는 16진수)가 뒤에 붙는 **#** 기호로 구성되어 있고, 해당되는 ASCII 문자를 나타냅니다. 제어 문자열의 예제는 다음과 같습니다.

```
#89#111#117
```

이 문장은 다음 인용 문자열과 동일합니다.

```
'You'
```

제어 문자열과 인용 문자열을 조합하면 더 큰 문자열을 만들 수 있습니다. 예를 들어,

```
'Line 1'#13#10'Line 2'
```

"Line 1"과 "Line 2" 사이에 캐리지 리턴 라인 피드가 삽입됩니다. 그러나 한 쌍의 연속적인 '는 단일 문자로 해석되기 때문에 이러한 방법으로 두 개의 인용 문자열을 연결할 수는 없습니다. 인용 문자열을 연결하려면 4-9 페이지의 "문자열 연산자"에서 설명된 + 연산자를 사용하거나, 하나의 인용 문자열로 만드십시오.

문자열의 *길이*는 문자열에 있는 문자의 수를 나타냅니다. 길이에 상관 없이 문자열은 문자열타입 및 *PChar* 타입과 호환됩니다. 길이가 1인 문자열은 모든 문자 타입과 호환되고, 확장 구문이 활성화(**{$X+}**)되면, 길이 $n \geq 1$인 문자열은 첨자가 0부터 시작하는 배열 및 *n*자의 압축 배열과 호환될 수 있습니다. 문자열 타입에 대한 자세한 내용은 5장 "데이터 타입, 변수 및 상수"를 참조하십시오.

주석 및 컴파일러 지시어

인접한 토큰의 경계를 정하는 구분자 또는 컴파일러 지시어로서의 기능을 하는 경우를 제외하고 주석은 컴파일러에 의해서 무시됩니다.

주석문을 만드는 방법은 다음과 같습니다.

```
{ Text between a left brace and a right brace constitutes a comment. }

(* Text between a left-parenthesis-plus-asterisk and an
   asterisk-plus-right-parenthesis also constitutes a comment.

// Any text between a double-slash and the end of the line constitutes a comment.
```

열린 **{** 또는 **(*** 바로 뒤에 달러 기호(**$**)가 있는 주석문은 컴파일러 지시어입니다. 예를 들면, 다음과 같습니다.

```
{$WARNINGS OFF}
```

이 구문은 컴파일러가 경고 메시지를 만들지 않도록 알려줍니다.

표현식

*표현식*은 값을 반환합니다. 예를 들면, 다음과 같습니다.

```
X                  { variable }
@X                 { address of a variable }
15                 { integer constant}
InterestRate       { variable }
Calc(X,Y)          { function call }
X *  Y              { product of X and Y }
Z / (1 - Z)        { quotient of Z and (1 - Z) }
X = 1.5            { Boolean }
C in Range1        { Boolean }
not Done           { negation of a Boolean }
['a','b','c']      { set }
Char(48)           { value typecast }
```

가장 간단한 표현식은 5장 "데이터 타입, 변수 및 상수"에서 설명하는 변수와 상수입니다. *연산자*, *함수 호출*, *집합 생성자*, *인덱스* 및 *타입 변환*을 사용하면 간단한 표현식에서 복잡한 표현식을 만들 수 있습니다.

연산자

연산자는 오브젝트 파스칼 언어의 일부분인 이미 정의된 함수와 비슷한 역할을 합니다. 예를 들어, 표현식 (X + Y)는 *피연산자*라고 부르는 변수 X와 Y로부터 + 연산자를 사용하여 만듭니다. 즉, X와 Y가 정수 또는 실수를 나타내는 경우 (X + Y)는 이 둘의 합을 반환합니다. 연산자에는 **@**, **not**, **^**, *****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, **−**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in** 및 **is**가 있습니다.

연산자 **@**, **not** 및 **^**는 피연산자가 하나인 *단항 연산자*입니다. **+**와 **−**를 단항 연산자와 이항 연산자로 모두 사용할 수 있다는 것을 제외하고, 나머지 연산자는 모두 피연산자가 두 개인 *이항 연산자*입니다. 피연산자 뒤에 오는 **^**(예를 들어 P^)를 제외한 단항 연산자는 -B 같이 항상 피연산자의 앞에 옵니다. 이항 연산자는 피연산자들 사이에 둡니다(예를 들어, A = 7).

일부 연산자는 전달된 데이터 타입에 따라 다르게 행동합니다. 예를 들어, **not**은 정수 피연산자에 대해서는 비트 단위(bitwise) 부정을 수행하고, 부울 피연산자에 대해서는 논리 부정을 수행합니다. 이런 연산자들은 아래와 같이 다양한 범주에서 나타납니다.

^, **is** 및 **in**을 제외한 모든 연산자는 *가변* 타입의 피연산자를 사용할 수 있습니다. 자세한 내용은 5−30 페이지의 "가변 타입(Variant types)"을 참조하십시오.

다음에 이어지는 단원들은 오브젝트 파스칼 데이터 타입과 다소 유사할 것입니다. 데이터 타입에 대한 내용은 5장 "데이터 타입, 변수 및 상수"를 참조하십시오.

복잡한 표현식의 연산자 우선 순위에 대한 내용은 4−12 페이지의 "연산자 우선 순위"를 참조하십시오.

산술 연산자

실수 또는 정수를 가지는 산술 연산자에는 **+**, **−**, *****, **/**, **div** 및 **mod**가 있습니다.

표 4.3 이항 산술 연산자

연산자	연산	피연산자 타입	결과 타입	예제
+	더하기	정수, 실수	정수, 실수	X + Y
−	빼기	정수, 실수	정수, 실수	Result - 1
*****	곱하기	정수, 실수	정수, 실수	P * InterestRate
/	실수 나누기	정수, 실수	정수	X / 2
div	정수 나누기	정수	정수	Total **div** UnitSize
mod	나머지	정수	정수	Y **mod** 6

표 4.4 단항 산술 연산자

연산자	연산	피연산자 타입	결과 타입	예제
+	부호 동일	정수, 실수	정수, 실수	+7
−	부호 부정	정수, 실수	정수, 실수	-X

다음 규칙은 산술 연산자에 대해 적용됩니다.

- *x*/*y*의 값은 *x*와 *y*의 타입에 관계 없이 *Extended* 타입을 가집니다. 다른 산술 연산자에서, 피연산자 중 하나가 실수이면, 결과는*Extended* 타입입니다. 반면에 피연산자 중 하나가 *Int64*타입이면, 결과는 *Int64* 타입입니다. 그 외에는 *정수* 타입입니다. 피연산자의 타입이 정수 타입의 부분범위인 경우 정수 타입인 것처럼 처리됩니다.
- *x* **div** *y*값은 *x*/*y*를 가장 가까운 정수로 내림한 값, 즉 몫입니다.
- **mod** 연산자는 피연산자 나눗셈의 나머지를 반환합니다. 다시 말하면 다음과 같습니다. *x* **mod** *y* = *x* − (*x* **div** *y*) * *y*.
- *x*/*y*, *x* **div** *y* 또는 *x* **mod** *y* 같은 표현식에서 *y*가 0이면 런타임 오류가 발생합니다.

부울 연산자

부울 연산자 **not**, **and**, **or** 및 **xor**은 부울 타입의 피연산자를 사용하고 *부울* 타입의 값을 반환합니다.

표 4.5 부울 연산자

연산자	연산	피연산자 타입	결과 타입	예제
not	부정	부울	*부울*	**not** (C **in** MySet)
and	논리곱	부울	*부울*	Done **and** (Total > 0)
or	논리합	부울	*부울*	**or** B
xor	배타적인 논리합	부울	*부울*	A **xor** B

이러한 연산들은 부울 로직의 표준 규칙에 따릅니다. 예를 들어, *x*와 *y* 모두가 *True*인 경우에만 *x* **and** *y* 표현식이 *True*입니다.

완전한 부울 계산과 부분 부울 계산 비교

컴파일러는 **and** 및 **or** 연산자에 대한 계산을 완전한 계산과 부분 계산 두 가지 모드로 지원합니다. *완전한 계산*은 전체 표현식의 결과가 이미 정해지더라도, 각 논리곱이나 논리합이 계산된다는 것을 의미합니다. *부분 계산*은 전체 표현식의 결과값이 정해지면 곧바로 중단되는 엄격한 왼쪽에서 오른쪽으로(left−to−right) 계산을 수행합니다. 예를 들어, *A*가 *False*이고, `A and B` 표현식이 부분 모드로 계산되는 경우, 컴파일러가 *B*를 계산하지 않습니다. 컴파일러는 *A*를 계산하자마자 전체 표현식이 *False*라는 것을 압니다.

부분 계산이 실행 시간을 최소화하고 또 대개의 경우 코드 크기를 최소화하기 때문에 일반적으로 더 많이 사용됩니다. 어떤 피연산자가 프로그램의 실행을 바꿀 수 있는 부작용이 있는 함수일 경우에는 완전한 계산이 더 편리합니다.

부분 계산은 유효하지 않는 런타임 연산을 피할 수 있게 해 줍니다. 예를 들어, 다음 코드는 문자열 *S*에서 첫 번째 쉼표를 만날 때까지 반복됩니다.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  f
  Inc(I);
end;
```

*S*에 쉼표가 없으면, 마지막 반복에서 *I* 값이 증가하여 *S*의 길이보다 큰 값을 가지게 됩니다. **while** 조건문이 다음 조건을 테스트하려 하면 완전한 계산은 *S[I]*를 읽으려고 시도하게 되어 런타임 오류를 야기할 수 있습니다. 반대로, 부분 계산에서는 **while**의 첫 번째 조건이 실패하므로 두 번째 조건 즉,(`S[I] <> ','`)는 계산되지 않습니다.

$B** 컴파일러 지시어를 사용하면 제어 계산 모드를 제어할 수 있습니다. 기본 상태는 **{$B−}로서 부분 계산을 활성화할 수 있습니다. 완전한 계산을 부분적으로만 활성화하려면, 사용자의 코드에 **{$B+}** 지시어를 추가하십시오. Compiler Option 대화 상자에서 Compiler Boolean Evaluation을 선택하면 프로젝트 전체를 완전한 계산 모드로 바꿀 수도 있습니다.

참고 피연산자가 가변과 관련된 경우에는 컴파일러가 **{$B−}** 모드에서도 항상 완전한 계산을 수행합니다.

논리 (비트 단위) 연산자

다음 논리 연산자는 정수 피연산자를 비트 단위(bitwise)로 조작합니다. 예를 들어, *X*에 저장된 값이 001101이고 *Y*에 저장된 값이 100001인 경우 문장은 다음과 같습니다.

```
Z := X or Y;
```

*Z*에 값 101101을 지정합니다.

표 4.6 논리 (비트 단위) 연산자

연산자	연산	피연산자 타입	결과 타입	예제
not	비트 단위 부정	정수	정수	**not** `X`
and	비트 단위 and	정수	정수	`X` **and** `Y`
or	비트 단위 or	정수	정수	`X` **or** `Y`
xor	비트 단위 xor	정수	정수	`X` **xor** `Y`
shl	왼쪽으로 비트 시프트	정수	정수	`X` **shl** `2`
shr	오른쪽으로 비트 시프트	정수	정수	`Y` **shr** `I`

비트 단위 연산자에는 다음과 같은 규칙이 적용됩니다.

- **not** 연산의 결과는 피연산자와 동일한 형식을 가집니다.
- **and**, **or** 또는 **xor** 연산의 피연산자가 둘 다 정수인 경우 그 결과는 두 타입의 가능한 값을 모두 포함하는 가장 작은 범위의 이미 정의된 정수 타입을 가집니다.

- *x* **shl** *y*와 *x* **shr** *y* 연산은 *y* 비트에 의해서 *x*의 값을 왼쪽이나 오른쪽으로 시프트하는데 결과 값은 *x*를 2^y로 곱하거나 나눈 것과 동일합니다. 그 결과는 *x*와 동일한 형을 가집니다. 예를 들어, *N*이 01101 (십진수 13) 값을 저장하고 있는 경우, `N shl 1`은 11010 (십진수 26)을 반환합니다.

문자열 연산자

관계 연산자 **=**, **<>**, **<**, **>**, **<=** 및 **>=**는 문자열 피연산자를 가집니다(4-10 페이지의 "관계 연산자" 참조). **+** 연산자는 두 개의 문자열을 연결합니다.

표 4.7 문자열 연산자

연산자	연산	피연산자 타입	결과 타입	예제
+	연결	문자열, 압축 문자열, 문자	문자열	`S + '. '`

문자열 연결에는 다음과 같은 규칙이 적용됩니다.

- **+**의 피연산자는 문자열, 압축 문자열(*Char* 타입의 압축 배열) 또는 문자일 수 있습니다. 그러나, 하나의 피연산자가 *WideChar* 타입인 경우, 다른 피연산자는 긴 문자열이어야만 합니다.
- **+** 연산의 결과는 모든 문자열 타입과 호환됩니다. 그러나, 피연산자가 모두 짧은 문자열이나 문자이고 그 연결한 길이가 255보다 클 경우 결과는 처음 255개 문자 이후를 제외한 값입니다.

포인터 연산자

관계 연산자 **<**, **>**, **<=** 및 **>=**는 *PChar*타입의 피연산자를 가질 수 있습니다(4-10 페이지의 "관계 연산자" 참조). 다음 연산자도 피연산자로서 포인터를 가질 수 있습니다.포인터에 대한 자세한 내용은 5-25 페이지의 "포인터와 포인터 타입(Pointer types)"을 참조하십시오.

표 4.8 문자 - 포인터 연산자

연산자	연산	피연산자 타입	결과 타입	예제
+	포인터 더하기	문자 포인터, 정수	문자 포인터	`P + I`
-	포인터 빼기	문자 포인터, 정수	문자 포인터, 정수	`P - Q`
^	포인터 역참조	포인터	포인터의 기본타입	`P^`
=	동등	포인터	*부울*	`P = Q`
<>	부등	포인터	*부울*	`P <> Q`

^ 연산자는 포인터를 역참조합니다. 피연산자는 일반 *포인터*를 제외한 모든 타입의 포인터가 될 수 있는데 역참조하기 전에 타입 변환되어야 합니다.

P **=** *Q*는 *P*와 *Q*가 동일한 주소를 가리키는 경우에만 *True*입니다. 그렇지 않은 경우에 *P* **<>** *Q*가 *True*입니다.

+와 **-** 연산자를 사용하여 문자 포인터의 오프셋을 증가시키고 감소시킬 수 있습니다. 사용자는 또한 **-**를 사용하여 두 문자 포인터의 오프셋 차이를 계산할 수 있습니다. 다음과 같은 규칙이 적용됩니다.

- *I*가 정수이고 *P*가 문자 포인터인 경우, *P* **+** *I*는 *P*의 주소에 *I*를 더합니다. 즉, *P* 뒤로 주소 *I* 문자만큼 오프셋한 포인터를 반환합니다. 표현식 *I* **+** *P*는 *P* **+** *I*와 동일합니다. *P* **−** *I*는 *P* 주소에서 *I*를 뺍니다. 즉, *P* 앞으로 주소 *I* 문자만큼 오프셋한 포인터를 반환합니다.
- *P*와 *Q*가 모두 문자 포인터인 경우 *P* − *Q*는 *P* 주소(상위 주소)와 *Q* 주소(하위 주소) 간의 차이를 계산합니다. 즉, *P*와 *Q* 사이의 문자 수를 나타내는 정수를 반환합니다. *P* + *Q*는 정의하지 않습니다.

집합 연산자

다음 연산자는 피연산자로서 집합을 가집니다.

표 4.9 집합 연산자

연산자	연산	피연산자 타입	결과 타입	예제
+	합집합	집합	집합	`Set1 + Set2`
−	차집합	집합	집합	`S - T`
*****	교집합	집합	집합	`S * T`
<=	서브셋	집합	*부울*	`Q <= MySet`
>=	수퍼셋	집합	*부울*	`S1 >= S2`
=	동등	집합	*부울*	`S2 = MySet`
<>	부등	집합	*부울*	`MySet <> S1`
in	속한다	순서, 집합	*부울*	`A in Set1`

+, **−** 및 ***에는 다음과 같은 규칙이 적용됩니다.

- *O*가 *X*나 *Y*(또는 둘다)에 속하면, 순서 *O*는 *X* **+** *Y*에 속합니다. *O*가 *X*에 속하고 *Y*에 속하지 않는 경우, *O*는 *X* **−** *Y*에 속합니다. *O*가 *X*와 *Y*에 있는 속하는 경우, *O*는 *X* ***** *Y*에 속합니다.
- **+**, **−** 또는 ***** 연산의 결과는 **set of** *A..B* 타입인데 *A*는 결과 집합에서 가장 작은 순서값이고 *B*는 가장 큰 순서값입니다.

다음 규칙은 **<=**, **>=**, **=**, **<>** 및 **in**에 적용됩니다.

- *X*가 *Y*에 속한 경우에만 *X* **<=** *Y*는 *True*입니다. 즉, *Z* **>=** *W*는 *W* **<=** *Z*와 동일합니다. *U*와 *V*의 요소가 정확히 동일한 경우에만 *U* **=** *V*는 *True*입니다. 그렇지 않은 경우 *U* **<>** *V*가 *True*입니다.
- 순서 *O*와 집합 *S*에서는 *O*가 *S*에 속하는 경우에만 *O* **in** *S*가 *True*입니다.

관계 연산자

관계 연산자는 두 개의 피연산자를 비교하는 데 사용됩니다. 연산자 **=**, **<>**, **<=** 및 **>=**는 집합에도 적용됩니다(4−10 페이지의 "집합 연산자" 참조). 즉, **= <>**는 포인터에도 적용됩니다(4−9 페이지의 "포인터 연산자" 참조).

표 4.10 관계 연산자

연산자	연산	피연산자 타입	결과 타입	예제
=	동등	일반, 클래스, 클래스 참조, 인터페이스, 문자열, 압축 문자열	*부울*	`I = Max`
<>	부등	일반, 클래스, 클래스 참조, 인터페이스, 문자열, 압축 문자열	*부울*	`X <> Y`
<	작다	일반, 문자열, 압축 문자열, *PChar*	*부울*	`X < Y`
>	크다	일반, 문자열, 압축 문자열, *PChar*	*부울*	`Len > 0`
<=	작거나 같다	일반, 문자열, 압축 문자열, *PChar*	*부울*	`Cnt <= I`
>=	크거나 같다	일반, 문자열, 압축 문자열, *PChar*	*부울*	`I >= 1`

가장 일반적인 타입에서는 비교가 간단합니다. 예를 들어, *I*와 *J*가 동일한 값을 가지는 경우에만 *I* = *J*가 *True*이고, 그렇지 않은 경우 *I* <> *J*가 *True*입니다. 관계 연산자에는 다음과 같은 규칙이 적용됩니다.

- 피연산자는 실수와 정수가 비교될 수 있다는 점 외에도 호환 가능한 타입이어야 합니다.
- 문자열은 확장 ASCII 문자 집합의 순서에 따라 비교됩니다. 문자 타입은 길이가 1인 문자열로 처리됩니다.
- 두 압축 문자열을 비교하려면 컴포넌트 수가 같아야 합니다. *n*개의 컴포넌트가 있는 압축 문자열을 문자열과 비교할 때 압축 문자열은 길이가 *n*인 문자열로 처리됩니다.
- 두 포인터가 동일한 문자 배열 내를 가리키는 경우에만 연산자 **<**, **>**, **<=** 및 **>=**가 *PChar* 피연산자에 적용됩니다.
- 연산자 **=**와 **<>**는 클래스 타입과 클래스 참조 타입의 피연산자를 가집니다. 클래스 타입의 피연산자가 있는 경우 **=**와 **<>**는 포인터에 적용되는 다음 규칙에 따라 계산됩니다. *C*와 *D*가 동일한 인스턴스 객체를 가리키는 경우에만 *C* = *D*가 *True*입니다. 그렇지 않은 경우 *C* <> *D*가 *True*입니다. 클래스 참조 타입의 연산자로는, *C*와 *D*가 동일한 클래스를 나타내는 경우에만 *C* = *D*는 *True*입니다. 그렇지 않은 경우 *C* <> *D*가 *True*입니다. 포인터에 대한 자세한 내용은 7장 "클래스 및 객체"를 참조하십시오.

클래스 연산자

연산자 **as**와 **is**는 클래스와 인스턴스 객체를 피연산자로 가집니다. 즉, **as**는 인터페이스에 대해서도 연산합니다. 자세한 내용은 7장 "클래스 및 객체" 및 10장 "객체 인터페이스"를 참조하십시오.

관계 연산자 **=**와 **<>**도 클래스에 대해 연산합니다. 4-10 페이지의 "관계 연산자"를 참조하십시오.

@ 연산자

@ 연산자는 변수, 함수, 프로시저, 메소드의 주소를 반환합니다. 즉, @는 피연산자에 대한 포인터를 생성합니다. 포인터에 대한 자세한 내용은 5-25 페이지의 "포인터와 포인터 타입(Pointer types)"을 참조하십시오. 다음 규칙은 @에 적용됩니다.

- *X*가 변수인 경우 @*X*는 *X*의 주소를 반환합니다. *X*가 프로시저 변수일 때에는 특별한 규칙이 적용됩니다. 5-29 페이지의 "문장 및 표현식의 프로시저 타입"을 참조하십시오. 기본값 {$T-} 컴파일러 지시어가 유효한 경우 @*X*의 타입은 *포인터*입니다. {$T+} 상태에서 @*X*는 ^*T* 타입을 가지는데 *T*는 *X*의 타입입니다.
- *F*가 루틴(함수 또는 프로시저)인 경우, @*F*는 *F*의 엔트리 포인트 주소 값을 반환합니다. @*F*의 타입은 항상 *포인터*입니다.
- @가 클래스에서 정의된 메소드에 적용될 때 메소드 식별자는 클래스 이름으로 한정되어야 합니다. 예를 들면, 다음과 같습니다.

  ```
  @TMyClass.DoSomething
  ```

  이 문장은 *TMyClass* 의 *DoSomething* 메소드를 가리킵니다 . 클래스와 메소드에 대한 자세한 내용은 7 장 " 클래스 및 객체 " 를 참조하십시오 .

연산자 우선 순위

복잡한 표현식에서 연산자 우선 순위는 연산이 수행되는 순서를 정합니다.

표 4.11 연산자 우선 순위

연산자	우선 순위
@, not	첫 번째(가장 높음)
*, /, div, mod, and, shl, shr, as	두 번째
+, -, or, xor	세 번째
=, <>, <, >, <=, >=, in, is	네 번째(가장 낮음)

우선 순위가 높은 연산자는 우선 순위가 낮은 연산자보다 먼저 계산되는 반면, 우선 순위가 같은 연산자는 왼쪽에서 오른쪽 순으로 계산됩니다. 따라서 표현식이 다음과 같은 경우,

```
X + Y * Z
```

여기서, *Y*와 *Z*를 곱한 후 그 결과에 *X*를 더합니다. 즉, *가 +보다 우선 순위가 높기 때문에 먼저 수행됩니다. 그러나 다음의 경우,

```
X - Y + Z
```

먼저 *X*에서 *Y*를 뺀 후 그 결과에 *Z*를 더합니다. 즉, -와 + 는 우선 순위가 같기 때문에 왼쪽에 있는 연산이 먼저 수행됩니다.

사용자는 괄호를 사용하여 이러한 우선 순위보다 우선하여 처리되도록 할 수 있습니다. 괄호 내에 있는 표현식을 먼저 계산하고, 계산된 결과는 하나의 피연산자로 처리됩니다. 예를 들면, 다음과 같습니다.

```
(X + Y) * Z
```

이 문장은 *X*와 *Y*의 합에 *Z*를 곱합니다.

경우에 따라서는 굳이 사용하지 않아도 될 것처럼 보이는 곳에 괄호가 필요합니다. 예를 들어, 다음의 표현식을 생각해 보십시오.

```
X = Y or X = Z
```

이 구문의 원래 의도는 분명히 다음과 같습니다.

```
(X = Y) or (X = Z)
```

그러나 괄호가 없다면 컴파일러는 연산자 우선 순위에 따라 다음과 같이 읽습니다.

```
(X = (Y or X)) = Z
```

이 경우 *Z*가 부울값이 아니면 컴파일 오류가 발생합니다.

엄밀히 말해 괄호를 많이 사용하는 것이 코드를 보다 읽고 쓰기 쉽게 합니다. 따라서 위의 첫 번째 예제는 다음과 같이 쓸 수 있습니다.

```
X + (Y * Z)
```

여기서 괄호는 컴파일러에는 불필요하지만 괄호를 사용함으로써 프로그래머와 코드를 읽어보는 사용자들이 연산자 우선 순위를 고려하지 않아도 됩니다.

함수 호출

함수는 값을 반환하기 때문에 함수 호출은 표현식입니다. 예를 들어, 두 개의 정수 인수를 가지고 있고 정수를 반환하는 *Calc*라는 함수를 정의했다면, 함수 호출 `Calc(24, 47)`은 정수 표현식입니다. *I*와 *J*가 정수 변수인 경우 `I + Calc(J, 8)` 역시 정수 표현식입니다. 함수 호출의 예제는 다음과 같습니다.

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

함수에 대한 자세한 내용은 6장 "프로시저 및 함수"를 참조하십시오.

집합 생성자

집합 생성자는 집합 타입의 값을 나타냅니다. 예를 들면, 다음과 같습니다.

```
[5, 6, 7, 8]
```

이 문장은 요소가 5, 6, 7, 8인 집합을 나타냅니다. 집합 생성자가 다음과 같다면,

```
[ 5..8 ]
```

이 문장은 위의 집합과 동일한 집합을 나타냅니다.

집합 생성자에 대한 구문은 다음과 같습니다.

[$item_1$, ..., $item_n$]

여기서 *item*은 집합의 기본 타입 순서를 나타내는 표현식이거나 사이에 두 개의 점(**..**)이 있는 표현식의 쌍입니다. *item*의 모양이 *x*..*y*인 경우 *x*부터 *y* 범위에 있는 모든 수를

간단하게 표기한 것입니다. 그러나 *x*가 *y*보다 큰 경우, *x*..*y*는 아무것도 나타내지 않고 [*x*..*y*]는 공집합니다. 집합 생성자 [ ]는 공집합을 나타내는 반면, [*x*]는 값 *x*를 유일한 요소로 가진 집합을 나타냅니다.

집합 생성자의 예는 다음과 같습니다.

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

집합에 대한 자세한 내용은 5−17 페이지의 "집합"을 참조하십시오.

인덱스

문자열, 배열, 배열 속성 및 문자열이나 배열에 대한 포인터를 인덱싱할 수 있습니다. 예를 들어, *FileName*이 문자열 변수인 경우 표현식 `FileName[3]`은 *FileName*이 나타내는 문자열에서 세 번째 문자를 반환하는 반면, `FileName[I + 1]`은 *I*번째 인덱스의 다음 문자를 반환합니다. 문자열에 대한 내용은 5−10 페이지의 "문자열 타입(String types)"을 참조하십시오. 배열과 배열 속성에 대한 내용은 5−18 페이지의 "배열" 및 7−19 페이지의 "배열 속성"을 참조하십시오.

타입 변환

표현식이 다른 타입에 속하는 것처럼 표현식을 처리할 때 유용하게 사용됩니다. 타입 변환을 이용하면 표현식의 타입을 일시적으로 변화시킬 수 있습니다. 예를 들어, `Integer('A')`는 *A*를 정수로 변환합니다.

타입 변환 구문은 다음과 같습니다.

typeIdentifier(*expression*)

표현식이 변수인 경우 결과는 *변수 타입 변환*이라 합니다. 그렇지 않은 경우 결과는 *값 타입 변환*입니다. 변수 타입 변환과 값 타입 변환은 구문이 동일하지만 두 종류의 타입 변환에 다른 규칙이 적용됩니다.

값 타입 변환

값 타입 변환에서 타입 식별자와 타입 변환 표현식은 모두 순서 타입 또는 포인터 타입이어야 합니다. 값 타입 변환의 예는 다음과 같습니다.

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

결과 값은 괄호 속의 표현식을 변환하여 구합니다. 지정된 타입의 크기가 표현식의 크기와 다른 경우 잘라내기나 확장이 일어날 수도 있습니다. 표현식 부호는 항상 보호됩니다.

구문이 다음과 같을 경우

```
I := Integer('A');
```

이 구문은 Integer('A')의 값, 즉 65를 변수 *I*에 할당합니다.
값 타입 변환 뒤에는 한정자가 올 수 없고, 할당문의 왼쪽에 나타날 수 없습니다.

변수 타입 변환

크기가 동일하고 정수와 실수를 함께 사용하지 않는다면, 모든 타입에 대한 변수를 타입 변환할 수 있습니다. (숫자 타입을 타입 변환하려면, *Int*와 *Trunc* 같은 표준 함수를 사용하십시오.) 변수 타입 변환의 예는 다음과 같습니다.

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

변수 타입 변환은 할당문의 어느 쪽에서든지 나타날 수 있습니다. 따라서 다음과 같은 경우,

```
var MyChar: char;
⋮
Shortint(MyChar) := 122;
```

이 문장은 *MyChar*에 *z* 문자(ASCII 122)를 지정합니다.

변수를 프로시저 타입으로 변환할 수 있습니다. 예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
type Func = function(X:Integer):Integer;
var
  F: Func;
  P:Pointer;
  N:Integer;
```

다음 할당문을 사용할 수 있습니다.

```
F := Func(P);          { Assign procedural value in P to F }
Func(P) := F;          { Assign procedural value in F to P }
@F := P;               { Assign pointer value in P to F }
P := @F;               { Assign pointer value in F to P }
N := F(N);             { Call function via F }
N := Func(P)(N);       { Call function via P }
```

다음의 예와 같이 변수 타입 변환을 한정자와 함께 사용할 수 있습니다.

```
type
  TByteRec = record
    Lo, Hi:Byte;
  end;
  TWordRec = record
    Low, High:Word;
  end;
  PByte = ^Byte;
var
  B:Byte;
  W:Word;
  L:Longint;
  P:Pointer;
begin
  W := $1234;
```

```
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $01234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

이러한 예에서, *TByteRec*는 워드의 하위 및 상위 바이트에 액세스하는 데 사용되고, *TWordRec*은 longint의 하위 및 상위 바이트에 액세스하는 데 사용됩니다. 같은 목적으로 이미 정의된 함수 *Lo*와 *Hi*를 호출할 수도 있지만 변수 타입 변환은 할당문의 왼쪽에서 사용할 수 있는 잇점이 있습니다.

타입 변환 포인터에 대한 내용은 5-25 페이지의 "포인터와 포인터 타입(Pointer types)"을 참조하십시오 클래스와 인터페이스 타입 변환에 대한 내용은 7-25 페이지의 "as 연산자" 및 10-10 페이지의 "인터페이스 타입 변환"을 참조하십시오.

선언과 문장

프로그램은 **uses** 절 및 **implementation** 같은 예약어와 별도로, 모두 *선언*과 *문장*으로 구성되며 *블록*으로 이루어집니다.

선언

변수, 상수, 타입, 필드, 속성, 프로시저, 함수, 프로그램, 유닛, 라이브러리 및 패키지의 이름은 *식별자*라고 합니다. 26057과 같은 숫자 상수는 식별자가 아닙니다. 식별자는 사용하기 전에 *선언*되어야 합니다. 단, 컴파일러가 자동으로 이해하는 몇몇 이미 정의된 타입, 루틴 및 상수, 그리고 함수 블록에서 나타나는 변수 *Result*와 메소드 구현에서 나타나는 변수 *Self*는 예외입니다.

선언은 식별자를 정의하고 적절한 곳에 식별자에 대해 메모리를 할당합니다. 예를 들면, 다음과 같습니다.

```
var Size:Extended;
```

이 문장이 *Extended*(실수) 값을 가지고 있는 *Size*라는 변수를 선언하는 반면,

```
function DoThis(X, Y:string}:Integer;
```

이 문장은 인수로 두 개의 문자열을 가지고 정수를 반환하는 *DoThis*라는 함수를 선언합니다. 각 선언은 세미콜론으로 끝납니다. 여러 개의 변수, 상수, 타입 및 레이블을 동시에 선언하는 경우에는 적절한 예약어를 다음과 같이 한 번만 사용해야 합니다.

```
var
  Size:Extended;
  Quantity:Integer;
  Description:string;
```

선언의 구문과 배치는 사용자가 정의하는 식별자의 종류에 따라 다릅니다. 일반적으로 선언은 **uses** 절 뒤에 있는 블록의 시작 또는 유닛의 인터페이스 섹션이나 구현 섹션의 시작 시에만 나타납니다. 변수, 상수, 타입, 함수 등의 선언에 대한 특별 문법은 각 장의 해당 항목에 설명되어 있습니다.

"힌트" 지시어인 **platform**, **deprecated** 및 **library**는 유닛이 **deprecated**로 선언될 수 없는 경우를 제외한 모든 선언에 올 수 있습니다. 프로시저나 함수 선언의 경우 힌트 지시어를 나머지 선언과 세미콜론으로 구분해야 합니다. 예를 들면, 다음과 같습니다.

```
procedure SomeOldRoutine; stdcall; deprecated;
var VersionNumber: Real library;
type AppError = class(Exception)
⋮
end platform;
```

소스 코드가 **{$HINTS ON} {$WARNINGS ON}** 모드로 컴파일되면, 이러한 지시어 중 하나와 함께 선언된 식별자에 대한 참조마다 적절한 힌트나 경고를 만듭니다. Windows나 Linux와 같은 특정 운영 체제에 고유한 항목을 표시하려면 **platform**, 역 호환성에 대해 항목이 구식인지, 또는 항목이 지원되는지를 나타내려면 **deprecated**를 사용하고, 특정 라이브러리를 사용하는지, 또는 VCL과 CLX와 같은 컴포넌트 프레임워크를 사용하는지를 나타내려면 **library**를 사용합니다.

문장

문장은 프로그램 내의 알고리즘 동작을 정의합니다. 할당문과 프로시저 호출문 같은 일반문을 조합하면 순환, 조건문 및 기타 구조문을 만들 수 있습니다.

블록내의 여러 문장과 유닛의 초기화 섹션이나 완료 섹션에서 사용된 여러 문장들은 세미콜론으로 구분합니다.

일반문(Simple statements)

일반문은 기타 다른 문장을 포함하지 않습니다. 일반문에는 할당문, 프로시저와 함수 호출 및 **goto** 문이 포함됩니다.

할당문

할당문의 형식은 다음과 같습니다.

variable := *expression*

여기서, *variable*은 변수, 변수 타입 변환, 역참조(dereference) 포인터 및 구조(structured) 변수의 컴포넌트와 같은 변수 참조를 나타내며, *expression*은 할당문에 지정할 수 있는 표현식을 나타냅니다. 함수 블록에서 *variable*은 정의된 함수 이름으로 바꿀 수 있습니다. 6장 "프로시저 및 함수"를 참조하십시오. 경우에 따라 **:=** 기호를 *할당 연산자*라고도 합니다.

할당문은 *variable*의 현재 값을 *expression*의 값으로 바꿉니다. 예를 들면, 다음과 같습니다.

```
I := 3;
```

이 문장은 변수*I*에 값 3을 지정합니다. 할당 연산자의 왼쪽에 있는 변수 참조를 오른쪽에 있는 표현식에 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
I := I + 1;
```

이 문장은 *I*의 값을 증가시킵니다. 다른 할당문의 예는 다음과 같습니다.

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

프로시저 및 함수 호출

프로시저 호출은 한정자가 있거나 없는 프로시저 이름들로 구성되는데, 경우에 따라 매개변수 목록이 있는 것도 있습니다. 예제는 다음과 같습니다.

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

확장 구문이 활성화된 경우(**{$X+}**), 프로시저 호출과 마찬가지로 함수 호출은 문장으로 처리될 수 있습니다.

```
MyFunction(X);
```

이런 방식으로 함수 호출을 사용하면 반환 값은 버립니다.

프로시저와 함수에 대한 자세한 내용은 6장 "프로시저 및 함수"를 참조하십시오.

Goto 문

goto 문의 형식은 다음과 같습니다.

```
goto label
```

이 문장은 지정한 레이블이 표시된 문장으로 프로그램 실행을 옮깁니다. 문장에 레이블을 표시하려면, 먼저 레이블을 선언해야 합니다. 그런 다음, 표시하려는 문장 앞에 다음과 같이 레이블과 콜론을 표시합니다.

```
label: statement
```

레이블은 다음과 같이 선언합니다.

```
label label;
```

다음과 같이 한 번에 여러 개의 레이블을 선언할 수도 있습니다.

```
label label1, ..., labeln;
```

유효한 식별자 또는 0과 9999 사이의 숫자를 레이블로 사용해야 합니다.

레이블 선언, 레이블이 표시된 문장 및 **goto** 문은 동일한 블록에 있어야 합니다.(4-27 페이지의 "블록과 유효 범위" 참조) 따라서 프로시저나 함수의 안이나 밖으로 이동하는 것은 불가능합니다. 블록에 둘 이상의 문장을 같은 레이블로 표시하지 마십시오.

예를 들면, 다음과 같습니다.

```
label StartHere;
⋮
StartHere: Beep;
goto StartHere;
```

이 문장은 *Beep* 프로시저를 반복적으로 호출하는 무한 순환을 만듭니다.

일반적으로 구조적 프로그램에서는 **goto** 문을 사용하지 않는 것이 좋습니다. 그러나 다음 예제처럼 중첩 순환으로부터 탈출하는 방법으로 사용되는 경우도 있습니다.

```
procedure FindFirstAnswer;
var X, Y, Z, Count:Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;

  ⋮ {code to execute if no answer is found}
  Exit;

  FoundAnAnswer:
    ⋮ { code to execute when an answer is found }
end;
```

중첩 순환을 *탈출*하기 위해 **goto** 문을 사용한다는 사실에 주의하십시오. 예기치 못한 결과를 야기할 수 있으므로 순환이나 기타 구조문으로 *들어가지* 마십시오.

구조문

구조문은 다른 문장들로 만듭니다. 다른 문장들을 순차적으로 또는 조건을 주어 실행하거나, 반복적으로 실행하고자 하는 경우 구조문을 사용합니다.

- 복합문 또는 **with** 문은 구조문을 구성하는 문장들을 차례로 실행합니다.
- 조건문 즉, **if** 또는 **case** 문은 지정된 조건에 따라 문장에서 실행됩니다.
- **repeat**, **while** 및 **for**를 비롯한 순환문은 문장을 차례로 반복해서 실행합니다.
- **raise**, **try...except** 및 **try...finally**를 비롯한 특수한 문장들은 *예외*를 생성하고 처리합니다. 예외 생성과 처리에 대한 내용은 7-26 페이지의 "예외"를 참조하십시오.

복합문

복합문은 작성된 순서대로 실행되는 일반문 또는 구조문입니다. 복합문은 예약어 **begin** 및 **end**로 묶이고, 복합문 내의 문장들은 세미콜론으로 구분합니다. 예를 들면, 다음과 같습니다.

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

end 앞의 마지막 세미콜론은 옵션입니다. 그러므로 다음과 같이 쓸 수 있습니다.

```
begin
  Z := X;
  X := Y;
  Y := Z
end;
```

복합문은 오브젝트 파스칼 구문에서 하나의 문장이 필요한 텍스트에서 중요합니다. 프로그램, 함수 및 프로시저 블록 외에도, 조건문이나 순환문과 같은 구조문 내에 나타납니다. 예를 들면, 다음과 같습니다.

```
begin
  I := SomeConstant;
  while I > 0 do
  begin
    f
    I := I - 1;
  end;
end;
```

문장이 단 하나인 복합문을 작성할 수도 있습니다. 복잡한 문장에서의 괄호처럼 **begin**과 **end**를 사용하여 명확하게 표시함으로써 코드의 가독성을 높일 수 있습니다. 또한 빈 복합문을 사용하여 내용이 없는 블록을 만들 수도 있습니다.

```
begin
end;
```

With 문

with 문은 레코드 필드 또는 객체의 필드, 속성 및 메소드를 참조하기 위한 간단한 방법입니다. **with** 문의 형식은 다음과 같습니다.

```
with obj do statement
```

또는

```
with obj1, ..., objn do statement
```

여기서, *obj*는 객체나 레코드를 나타내는 변수 참조이고, *statement*는 일반문 또는 구조문입니다. *statement* 내에서 한정자 없이 식별자만을 사용하여 *obj*의 필드, 속성 및 메소드를 나타낼 수 있습니다.

예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
type TDate = record
  Day:Integer;
```

```
  Month:Integer;
  Year:Integer;
end;

var OrderDate:TDate;
```

다음과 같은 **with** 문을 작성할 수 있습니다.

```
with OrderDate do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1;
  end
  else
    Month := Month + 1;
```

이 문장은 다음과 동일합니다.

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month := 1;
  OrderDate.Year := OrderDate.Year + 1;
end
else
  OrderDate.Month := OrderDate.Month + 1;
```

*obj*의 해석에 배열 인덱싱이나 포인터 역참조가 포함되면, 이러한 동작은 *statement*가 실행되기 전에 한 번 수행됩니다. 이 문장은 **with** 문을 간결하고 효율적으로 만듭니다. 또한 *statement* 내의 변수에 대한 지정이 **with** 문을 실행하는 동안은 *obj*의 해석에 영향을 미칠 수 없다는 것을 의미합니다.

with 문의 각 변수 참조 이름이나 메소드 이름은 가능한 경우 지정 객체나 레코드의 멤버로서 해석됩니다. **with** 문에서 액세스하려는 변수나 메소드의 이름이 동일한 경우, 다음 예제에서와 같이 한정자를 사용해야 합니다.

```
with OrderDate do
  begin
    Year := Unit1.Year
    ⋮
  end;
```

with 뒤에 객체나 레코드가 여러 개 있으면, 중첩된 **with** 문처럼 처리됩니다. 따라서,

with obj_1, obj_2, ..., obj_n do *statement*

이 문장은 다음과 같습니다.

with obj_1 do
  with obj_2 do
    ⋮
    with obj_1 do
      statement

이 경우에, *statement*에 있는 각 변수 참조 이름이나 메소드 이름은 가능하면 obj_n의 멤버로 해석됩니다. 그렇지 않은 경우 가능한 obj_{n-1}의 멤버로 해석됩니다. *obj*를 해석

하는 데 동일한 규칙이 적용됩니다. 예를 들어 obj_n가 obj_1과 obj_2 모두의 멤버인 경우, $obj_2.obj_n$으로 해석됩니다.

If 문

if 문의 형태는 다음 두 가지, **if...then**과 **if...then...else**가 있습니다. **if...then** 문의 구문은 다음과 같습니다.

```
if expression then statement
```

여기서, *expression*은 부울 값을 반환합니다. *expression*이 *True*인 경우 *statement*가 실행됩니다. 그렇지 않은 경우 실행되지 않습니다. 예를 들면, 다음과 같습니다.

```
if J <> 0 then Result := I/J;
```

if...then...else 문의 구문은 다음과 같습니다.

```
if expression then statement1 else statement2
```

여기서, *expression*은 부울 값을 반환합니다. *expression*이 *True*인 경우 $statement_1$이 실행됩니다. 그렇지 않은 경우 $statement_2$가 실행됩니다. 예를 들면, 다음과 같습니다.

```
if J = 0 then
  Exit;
else
  Result := I/J;
```

then과 **else** 절은 각각 하나의 문장을 가지고 있지만, 구조문이 될 수 있습니다. 예를 들면, 다음과 같습니다.

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
else if Count = Last then
  Done := True
else
  Exit;
```

then 절과 **else** 사이에 세미콜론이 없다는 사실에 주의하십시오. 블록 내의 다음 문장과 구분하기 위해 전체 **if** 문 뒤에 세미콜론을 사용할 수 있지만 **then**과 **else** 절 사이에는 공백이나 캐리지 리턴 이외에 다른 것은 필요 없습니다. **if** 문의 **else** 바로 앞에 세미콜론을 사용하는 것은 일반적인 프로그래밍 오류입니다.

일련의 중첩된 **if** 문을 사용할 때에는 특히 주의를 기울여야 합니다. 일부 **if** 문에는 **else** 절이 있고, 다른 **if** 문에서는 **else** 절이 없기 때문에 이러한 문제가 발생합니다. **if** 문에 비해 **else** 절이 적은 일련의 중첩 조건문에서는, 어떤 **if**에 속하는 **else** 절인지 명확하지 않을 수도 있습니다.

```
if expression1 then if expression2 then statement1 else statement2;
```

이를 파싱하는 방법에는 두 가지가 있습니다.

```
if expression1 then [ if expression2 then statement1 else statement2 ];
```

```
if expression1 then [ if expression2 then statement1 ] else statement2;
```

컴파일러는 항상 첫 번째 방법으로 파싱합니다. 즉, 실제 코드에서 다음 문장은

```
if ...{ expression1 } then
  if ...{ expression2 } then
    ... { statement1 }
  else
    ... { statement2 } ;
```

다음 문장과 동일합니다.

```
if ...{ expression1 } then
begin
  if ...{ expression2 } then
    ... { statement1 }
  else
    ... { statement2 }
end;
```

가장 내부에 있는 조건문부터 시작하여 각각의 **else**를 왼쪽 방향으로 가장 가까운 **if**로 묶으면서, 중첩된 조건문을 파싱합니다. 컴파일러가 두 번째 방식으로 예제를 읽도록 하려면 다음과 같이 명시적으로 작성해야만 합니다.

```
if ...{ expression1 } then
begin
  if ...{ expression2 } then
    ... { statement1 }
end
else
  ... { statement2 } ;
```

Case 문

case 문은 복잡하게 중첩된 **if** 조건을 읽기 쉽게 합니다. **case** 문의 형식은 다음과 같습니다.

```
case selectorExpression of
  caseList1:statement1;
  ⋮
  caseListn:statementn;
end
```

여기서, *selectorExpression*은 순서 타입(문자 타입은 사용할 수 없음) 표현식이고, *caseList*는 다음 중 하나입니다.

- 컴파일러가 프로그램을 실행하지 않고 계산할 수 있는 숫자, 선언된 상수 또는 기타 표현식. *selectorExpression*과 호환되는 순서 타입이어야 합니다. 따라서 `7`, `True`, `4 + 5 * 3`, `'A'` 및 `Integer('A')`는 모두 *caseList*로 사용할 수 있지만 변수와 대부분의 함수 호출은 사용할 수 없습니다. (*Hi*와 *Lo* 같은 몇몇 내장 함수는 *caseList*에 사용할 수 있습니다. 5-41 페이지의 "상수 표현식" 참조)
- *First..Last* 형식의 부분범위. 여기서, *First*와 *Last*는 위의 조건을 모두 만족해야 하며 *First*는 *Last*보다 작거나 같아야 합니다.
- $item_1,...,\ item_n$ 형식의 목록. 여기서, 각 *item*은 조건 중 하나를 만족해야 합니다.

*caseList*가 나타내는 각각의 값은 **case** 문에서 고유해야 합니다. 즉, 부분범위와 목록이 중복될 수 없습니다. **case** 문의 마지막에는 **else** 절이 올 수 있습니다.

```
case selectorExpression of
  caseList1:statement1;
  ⋮
  caseListn:statementn;
else
  statements;
end
```

여기서, *statements*는 세미콜론으로 구분된 여러 개의 문장을 나타냅니다. **case** 문이 실행되는 경우 $statement_1$ ... $statement_n$ 중에 많아야 하나의 문장만이 실행됩니다. *selector Expression*과 값이 같은 *caseList*의 *statement*가 실행됩니다. *caseList* 중에서 *selectorExpression*과 같은 값이 없을 경우 **else** 절이 있으면 **else** 절의 문장이 실행됩니다.

다음 **case** 문은

```
case I of
  1..5: Caption := 'Low';
  6..9: Caption := 'High';
  0, 10..99: Caption := 'Out of range';
else
  Caption := '';
end;
```

중첩된 다음 조건과 동일합니다.

```
if I in [1..5] then
  Caption := 'Low'
  else if I in [6..10] then
    Caption := '';
    else if (I = 0) or (I in [10..99]) then
      Caption := 'Out of range'
      else
        Caption := '';
```

case 문의 다른 예제는 다음과 같습니다.

```
case MyColor of
  Red:X := 1;
  Green:X := 2;
  Blue:X := 3;
  Yellow, Orange, Black:X := 0;
end;
```

```
case Selection of
  Done: Form1.Close;
  Compute: CalculateTotal(UnitCost, Quantity);
else
  Beep;
end;
```

순환문

실행이 중단되는 때를 정하는 제어 조건이나 변수를 사용하여, 순환문으로 일련의 문장을 반복해서 실행할 수 있습니다. 오브젝트 파스칼의 순환문에는 **repeat** 문, **while** 문 및 **for** 문이 있습니다.

표준 *Break*와 *Continue* 프로시저를 사용하여 **repeat**, **while**, **for** 문의 흐름을 제어할 수 있습니다. *Break*는 Break가 있는 곳에서 문장을 종료하는 반면, *Continue*는 순환문 내에서 *continue* 이후의 문장을 무시하고 순환문의 다음 반복을 실행합니다. 이러한 프로시저에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

Repeat 문

repeat 문의 구문 형식은 다음과 같습니다.

```
repeat statement1; ...; statementn; until expression
```

여기서, *expression*은 부울 값을 반환합니다. **until** 앞에 있는 마지막 세미콜론은 옵션입니다. **repeat** 문은 repeat 문 내에 있는 문장들을 차례로 실행하고, 각 반복이 끝난 후에 *expression*을 테스트합니다. *expression*이 *True*를 반환하는 경우, **repeat** 문은 종료됩니다. 첫 번째 반복의 끝에서 *expression*을 계산하기 때문에 repeat 내에 있는 문장들은 최소한 한 번 이상 실행됩니다.

repeat 문의 예제는 다음과 같습니다.

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;
```

```
repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While 문

while 문은 while 문 내에 있는 문장들을 실행하기 전에 제어 조건을 계산한다는 점을 제외하고 **repeat** 문과 유사합니다. 따라서 조건이 false면 while 문 내에 있는 문장들은 실행되지 않습니다.

while 문의 구문 형식은 다음과 같습니다.

```
while expression do statement
```

여기서, *expression*은 부울 값을 반환하고 *statement*는 복합문이 될 수 있습니다. **while** 문은 문장 내의 *statement*를 반복적으로 실행하고 각각을 반복하기 전에 *expression*을 테스트합니다. *expression*이 *True*를 반환하는 한, while 문은 계속 실행됩니다.

while 문의 예제는 다음과 같습니다.

```
while Data[I] <> X do I := I + 1;
```

```
while I > 0 do
begin
```

```
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

For 문

repeat 또는 **while** 문과 달리 **for** 문은 순환 반복 횟수를 명시적으로 지정해야 합니다. **for** 문의 구문 형태는 다음과 같습니다.

```
for counter := initialValue to finalValue do statement
```

또는

```
for counter := initialValue downto finalValue do statement
```

여기서,

- *counter*는 **for** 문이 있는 블록에서 선언된 순서 타입 지역 변수로 한정자가 없습니다.
- *initialValue*와 *finalValue*는 *counter*에 지정할 수 있는 표현식입니다.
- *statement*는 *counter*의 값을 변경하지 않는 일반문 또는 구조문입니다.

for 문은 *initialValue* 값을 *counter*에 지정한 다음, *statement*를 반복해서 실행하여, 각각의 반복이 끝나면 *counter*를 증가시키거나 감소시킵니다. **for...to** 구문은 *counter*를 증가시키는 반면, **for...downto** 구문은 감소시킵니다. *counter*가 *finalValue*와 동일한 값을 반환하고, *statement*가 한 번 더 실행되면, **for** 문이 종료됩니다. 다시 말하면, *statement*가 *initialValue*부터 *finalValue*까지의 범위에 있는 각 값에 대해 한 번씩 실행됩니다. *initialValue*가 *finalValue*와 같을 경우, *statement*는 한 번만 실행됩니다. *initialValue*가 **for...to** 문의 *finalValue*보다 크거나, **for...downto** 문의 *finalValue*보다 작으면, *statement*는 실행되지 않습니다. **for** 문이 종료된 후에 *counter*의 값은 정의되지 않습니다.

표현식 *initialValue*와 *finalValue*는 순환의 실행을 제어하기 위해 순환이 시작되기 전에 단 한 번만 계산됩니다. 따라서 **for...to** 문은 다음 **while** 구문과 완전히는 아니지만 거의 동일합니다.

```
begin
  counter := initialValue;
  while counter <= finalValue do
  begin
    statement;
    counter := Succ(counter);
  end;
end
```

이 구문과 **for...to** 문의 차이는 **while** 순환이 각각의 반복 이전에 *finalValue*를 다시 계산한다는 것입니다. *finalValue*가 복잡한 표현식인 경우 이 때문에 눈에 띄게 느려질 수

있고, 이는 *statement* 내의 *finalValue* 값을 변경하면 순환의 실행에 영향을 줄 수 있다는 것을 의미합니다.

for 문의 예제는 다음과 같습니다.

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];

for I := ListBox1.Items.Count - 1 downto 0 do
  ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);

for I := 1 to 10 do
  for I := 1 to 10 do
  begin
    X = 0;
    for I := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;

for C := Red to Blue do Check(C);
```

블록과 유효 범위

선언과 문장은 *블록*으로 구성되어 레이블과 식별자에 대한 로컬 네임스페이스 또는 *유효 범위(scope)*를 정의합니다. 변수 이름 같은 하나의 식별자는 프로그램의 다른 블록에서 다른 의미를 가질 수 있습니다. 각 블록은 프로그램, 함수 또는 프로시저의 선언의 일부분입니다. 즉, 각 프로그램, 함수 또는 선언은 하나의 블록을 가집니다.

블록

블록은 복합문이 있는 일련의 선언들로 구성됩니다. 모든 선언은 블록의 시작 시 함께 나타나야 합니다. 그러므로 블록의 형식은 다음과 같습니다.

```
declarations
begin
  statements
end
```

declarations 섹션에는 변수, 리소스 문자열을 비롯한 상수, 타입, 프로시저, 함수 및 레이블에 대한 선언을 순서에 상관없이 선언합니다. 프로그램 블록에서는 *declarations* 섹션이 둘 이상의 **exports** 절을 포함할 수도 있습니다(9장 "라이브러리 및 패키지" 참조).

예를 들어, 다음과 같은 함수 선언에서

```
function UpperCase(const S:string):string;
var
  Ch:Char;
  L:Integer;
  Source, Dest:PChar;
begin
  ⋮
end;
```

선언의 첫 번째 행은 함수 헤더이고, 나머지 행들은 블록을 구성합니다. *Ch*, *L*, *Source* 및 *Dest*는 지역 변수입니다. 즉, 지역 변수 선언은 *UpperCase* 함수 블록에만 적용되므로, 프로그램 블록이나 유닛의 인터페이스 섹션 또는 구현 섹션에서 이 지역 변수를 오버라이드할 수 있습니다.

유효 범위

변수나 함수 이름과 같은 식별자는 선언 *유효 범위(scope)* 내에서만 사용할 수 있습니다. 선언의 위치로 유효 범위가 결정됩니다. 프로그램, 함수, 프로시저의 선언 내에서 선언된 식별자는 선언된 블록에 제한된 유효 범위를 가집니다. 유닛의 인터페이스 섹션에서 선언된 식별자는 식별자가 선언된 유닛을 사용하는 다른 유닛이나 프로그램을 포함하는 유효 범위를 가집니다. 좁은 유효 범위를 가지는 식별자 특히, 함수와 프로시저에서 선언된 식별자를 때때로 *지역*이라고 하고, 넓은 범위를 가지는 식별자를 *전역*이라고 합니다.

식별자 범위를 정하는 규칙은 다음과 같이 요약할 수 있습니다.

코드에서 식별자의 선언 위치	식별자의 유효 범위
프로그램, 함수 또는 프로시저의 선언	선언된 위치부터 현재 블록 끝까지(유효 범위 내의 모든 블록 포함).
유닛의 인터페이스 섹션	선언된 위치부터 유닛의 끝까지, 그리고 선언된 유닛을 사용하는 다른 유닛이나 프로그램까지. (3장 "프로그램 및 유닛" 참조)
함수나 프로시저의 블록 이외의 유닛 구현 섹션	선언된 위치부터 구현 섹션의 끝까지. 구현 섹션 내의 함수 또는 프로시저 모두에서 식별자를 사용할 수 있습니다.
레코드 타입의 정의 (즉, 식별자는 레코드에서 필드 이름)	선언된 위치부터 필드 타입 정의의 끝까지. (5-21 페이지의 "레코드" 참조)
클래스 정의(즉, 식별자는 클래스에서 속성 또는 메소드의 이름)	선언된 위치부터 클래스 타입 정의의 끝까지. 클래스의 자손과 클래스 및 클래스의 자손의 모든 멤버 메소드 블록 포함.(7장 "클래스 및 객체" 참조)

이름 충돌

하나의 블록이 다른 블록을 둘러싸는 경우 둘러싸는 블록을 *외부 블록(outer block)*이라 하고 둘러싸인 블록을 *내부 블록(inner block)*이라고 합니다. 외부 블록에서 선언된 식별자가 내부 블록에서 다시 선언되는 경우 내부 선언은 외부 블록을 오버라이드하고 내부 블록이 유효한 영역에 대해 식별자의 의미를 정합니다. 예를 들어, 사용자가 유닛의 인터페이스 섹션에서 *MaxValue*라는 변수를 선언하고 나서 그 유닛 내에서의 함수 선언에서 동일한 이름으로 새로운 변수를 선언하는 경우에 그 함수 블록에서 한정되지 않은 *MaxValue*는 지역 선언인 두 번째 변수로 인식합니다. 마찬가지로, 다른 함수 내에서 선언된 함수는, 외부 함수가 사용하는 식별자가 로컬로 다시 선언되어 내부 유효 범위(inner scope)를 새로 만듭니다.

유닛을 여러 개 사용하면 유효 범위 정의가 더 복잡해집니다. **uses** 절에 나열된 각 유닛은 사용된 잔여 유닛과 **uses** 절을 포함하는 프로그램이나 유닛을 둘러싸는 새로운 유효 범위를 부과합니다. **uses** 절에서 첫 번째 유닛은 가장 외부 범위를 나타내고, 다음 유닛은 이전 범위 내부의 새로운 범위를 나타냅니다. 두 개 이상의 유닛이 인터페이스 섹션에서 동일한 식별자를 선언하는 경우 식별자에 대해 한정되지 않은 참조는 가장 내부 유

효 범위(innermost scope) 즉, 참조 자체가 나타나는 유닛에서 선언을 선택하거나, 그 유닛이 식별자를 선언하지 않는 경우 식별자에 대한 비정식 참조는 식별자를 선언한 **uses** 절의 마지막 유닛이 지정됩니다.

시스템 유닛은 각 프로그램이나 유닛에 의해서 자동으로 사용됩니다. 컴파일러가 자동으로 이해하는 이미 정의된 타입, 루틴 및 상수와 함께 *시스템*에서 선언은 항상 가장 외부 범위를 가집니다.

한정된 식별자(4-2 페이지의 "한정된 식별자" 참조) 또는 **with** 문(4-20 페이지의 "With 문" 참조)을 사용하면 이러한 유효 범위에 관한 규칙들은 오버라이드할 수 있으며 또한 내부 선언을 무시할 수 있습니다.

5

데이터 타입 , 변수 및 상수

*데이터 타입*이란 본질적으로 데이터 종류에 대한 이름입니다. 변수를 선언할 때는 변수의 타입을 지정해주어야 합니다. 이러한 변수의 타입은 변수가 가질 수 있는 값과 변수에서 수행될 수 있는 동작들을 결정합니다. 모든 함수와 마찬가지로 모든 표현식은 특정 타입의 데이터를 반환합니다. 대부분의 함수와 프로시저는 특정 타입의 매개변수를 필요로 합니다.

오브젝트 파스칼은 "철저하게 타입이 지정된" 랭귀지입니다. 즉, 다양한 데이터 타입을 구별하며, 어떤 하나의 타입에 대해 다른 타입을 대체할 수 없습니다. 따라서 컴파일러에서 데이터를 지능적으로 처리하고 코드를 더 철저히 검증하여, 진단하기 어려운 런타임 오류를 예방하기 때문에 유용합니다. 하지만 보다 넓은 유용성이 필요한 경우에는 강력한 타입 지정을 우회할 수 있는 방법도 있습니다. 이러한 방법으로는 *타입 변환*(4-14 페이지의 "타입 변환" 참조), *포인터*(5-25 페이지의 "포인터와 포인터 타입(Pointer types)" 참조), *가변 타입*(5-30 페이지의 "가변 타입(Variant types)" 참조), 레코드의 *가변 타입 부분*(5-22 페이지의 "레코드의 가변 부분" 참조) 및 변수의 *절대 주소 지정*(5-38 페이지의 "절대 주소" 참조) 등이 있습니다.

타입 정보

다음은 오브젝트 파스칼 데이터 타입의 몇 가지 분류 방식입니다.

- 일부 타입은 *이미 정의된*(또는 *기본 제공된*) 타입이며, 이들 타입은 선언을 하지 않아도 컴파일러에서 자동으로 인식됩니다. 이 설명서에서 설명되는 거의 대부분의 타입은 이미 정의된 타입입니다. 다른 타입들은 선언에 의해 생성되며, 이 타입들에는 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 오브젝트 파스칼의 모든 구현에서 동일합니다. 일반 타입의 범위와 형식은 플랫폼에 따라 다르며, 여러 가지 구현에 따라 다를 수 있습니다. 대부분의 이미 정의된 타입은 기본 타입이지만, 일부 정수, 문자, 문자열 및 포인터 타입은 일반 타입(generic type)입니다. 일반 타입은 최적의 성능과 뛰어난 이식성을 제공하기 때문에 가능하다면 일반 타입을 사용하는 것이 좋습

니다. 하지만 일반(generic) 타입의 한 구현에서 다른 구현으로 저장 형식에 변화가 발생하면 호환성 문제가 발생할 수도 있습니다(예를 들어, 데이터를 파일에 스트리밍하려는 경우).

- 타입은 *일반 타입(simple types)*, *문자열 타입*, *구조 타입*, *포인터 타입*, *프로시저 타입* 또는 *가변 타입*으로 분류될 수 있습니다. 또한, 타입 식별자 자체도 특정 함수(*High*, *Low* 및 *SizeOf* 등)에 매개변수로 전달할 수 있기 때문에 특수한 "타입"에 속하는 것으로 간주될 수 있습니다.

다음은 오브젝트 파스칼 데이터 타입을 간략히 분류한 것입니다.

일반 타입(Simple types)
  순서
    정수
    문자
    부울
    열거
    부분범위
  실수
문자열 타입(String types)
구조 타입(Structured types)
  집합
  배열
  레코드
  파일
  클래스
  클래스 참조
  인터페이스
포인터 타입(Pointer types)
프로시저 타입(Procedural types)
가변 타입(Variant types)
타입 식별자

표준 함수 *SizeOf*는 모든 변수와 타입 식별자에서 사용할 수 있습니다. 이 함수는 지정된 타입의 데이터를 저장하기 위해 사용된 메모리의 양(바이트)을 나타내는 정수를 반환합니다. 예를 들어, *Longint* 변수는 4바이트의 메모리를 사용하기 때문에 `SizeOf (Longint)`는 4를 반환합니다.

타입 선언은 다음 단원에서 설명합니다. 타입 선언에 대한 일반적인 내용은 5-36 페이지의 "타입 선언"을 참조하십시오.

일반 타입(Simple types)

일반 타입에는, *순서 타입*과 *실수 타입*이 있으며, 정렬된 값들을 정의합니다.

순서 타입(Oridinal types)

순서 타입에는 *정수 타입*, *문자 타입*, *부울 타입*, *열거 타입* 및 *부분범위 타입*이 있습니다. 순서 타입은 처음 값을 제외한 모든 값에 고유한 *선행자*가 있고 마지막 값을 제외한

모든 값에는 고유한 *후행자*가 있는 정렬된 값을 정의합니다. 또한 각 값은 타입의 순서를 결정하는 *순서*를 가집니다. 대부분의 경우 값에 순서 *n*이 있으면, 이 값의 선행자는 *n*−1 순서를 가지고 후행자는 *n*+1 순서를 가집니다.

- 정수 타입의 경우에는 값 자체가 값의 순서입니다.
- 부분범위 타입은 기본 타입의 순서를 유지합니다.
- 다른 순서 타입의 경우에는 기본적으로 처음 값이 순서 0이고 다음 값은 순서 1, 순서 2 등과 같이 됩니다. 열거 타입의 선언은 명시적으로 이 기본값을 무시할 수 있습니다.

일부 이미 정의된 함수는 순서 값과 타입 식별자에 따라 작동합니다. 이미 정의된 함수에서 중요한 내용은 다음과 같이 요약할 수 있습니다.

함수	매개변수	반환 값	주의
Ord	순서 표현식	표현식 값의 순서	*Int64* 인수를 사용하지 마십시오.
Pred	순서 표현식	표현식 값의 선행자	**write** 프로시저가 있는 속성에 대해서는 사용하지 마십시오.
Succ	순서 표현식	표현식 값의 후행자	**write** 프로시저가 있는 속성에 대해서는 사용하지 마십시오.
High	순서 타입 식별자 또는 순서 타입의 변수	타입의 최고값	또한 짧은 문자열 타입과 배열에 대해서도 작동합니다.
Low	순서 타입 식별자 또는 순서 타입의 변수	타입의 최저값	짧은 문자열 타입과 배열에 대해서도 작동합니다.

예를 들어, `High(Byte)`는 *바이트* 타입의 최고값이 255이기 때문에 255를 반환하고 `Succ(2)`는 3이 2의 후행자이기 때문에 3을 반환합니다.

표준 프로시저 *Inc*와 *Dec*는 순서 변수의 값을 늘리거나 줄입니다. 예를 들어 `Inc(I)`는 `I := Succ(I)`와 동일하며, *I*가 정수 변수라면 `I := I + 1`입니다.

정수 타입 (Integer types)

정수 타입은 전체 숫자의 서브셋을 나타냅니다. 일반적인 정수 타입은 *Integer*와 *Cardinal*입니다. 기본 CPU와 운영 체제에서 정수는 최적의 수행 결과를 가져 오기 때문에 가능한 모든 경우에 정수 타입을 사용하는 것이 좋습니다. 다음 표는 현재 32비트 오브젝트 파스칼 컴파일러의 정수 타입 범위와 저장 형식을 보여줍니다.

표 5.1 오브젝트 파스칼 32 비트 구현의 일반적인 정수 타입

타입	범위	타입
정수	−2147483648..2147483647	부호있는 32비트
기수	0..4294967295	부호없는 32비트

기본 정수 타입에는 *Shortint*, *Smallint*, *Longint*, *Int64*, *Byte*, *Word* 및 *Longword*가 있습니다.

표 5.2 기본 정수 타입

타입	범위	타입
Shortint	−128..127	부호있는 8비트
Smallint	−32768..32767	부호있는 16비트
Longint	−2147483648..2147483647	부호있는 32비트
Int64	$-2^{63}..2^{63}-1$	부호있는 64비트
Byte	0..255	부호없는 8비트
Word	0..65535	부호없는 16비트
Longword	0..4294967295	부호없는 32비트

일반적으로 정수에서의 산술 연산은 32비트의 *Longint*와 동일한 *정수* 타입 값을 반환합니다. *Int64* 피연산자에서 수행되는 연산에만 *Int64* 타입 값을 반환합니다. 따라서 다음과 같은 코드는 부적절한 결과를 발생시킵니다.

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

Int64 반환 값을 얻으려면, *I*를 *Int64*로 타입 변환합니다.

```
  ⋮
J := Int64(I) + 1;
```

자세한 내용은 4−6 페이지의 "산술 연산자"를 참조하십시오.

참고 정수 인수가 필요한 대부분의 표준 루틴은 *Int64* 값을 32비트 값으로 자릅니다. 그러나 *High*, *Low*, *Succ*, *Pred*, *Inc*, *Dec*, *IntToStr* 및 *IntToHex* 루틴은 *Int64* 인수를 완벽하게 지원합니다. 또한 *Round*, *Trunc*, *StrToInt64* 및 *StrToInt64Def* 함수는 *Int64* 값을 반환합니다. *Ord*를 비롯한 몇몇 루틴은 *Int64* 값을 인수로 전혀 사용할 수 없습니다.

정수 타입의 마지막 값을 증가시키거나 처음 값을 감소시키는 경우에는 결과값이 범위의 시작 값이나 끝 값으로 랩어라운드됩니다. 예를 들어 *Shortint* 타입은 범위가 −128..127 입니다. 따라서 다음과 같이 코드를 실행할 경우

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

I 값은 −128입니다. 하지만 컴파일러의 범위 검사를 활성화하면 이 코드는 런타임 오류가 발생합니다.

문자 타입 (Character types)

기본 문자 타입은 *AnsiChar*와 *WideChar*입니다. *AnsiChar* 값은 대부분 멀티바이트인 로케일 문자 집합에 따라 정렬된 바이트 크기(8비트)의 문자입니다. *AnsiChar*은 본래는 이름을 그대로 가진 ANSI 문자 집합을 따라서 작성된 것이지만 지금은 현재의 로케일 문자 집합를 따르도록 확장되었습니다.

WideChar 문자는 모든 문자를 나타내는데 2바이트 이상을 사용합니다. 현재의 문자 집합 구현에서, *WideChar*는 앞으로의 문자 집합 구현에서는 더 커질 수 있다는 전제 하에 유니코드 문자 집합에 따라 정렬한 워드 크기(16비트)의 문자입니다. 처음 256 유니코드 문자는 ANSI 문자와 일치합니다.

일반적인 문자 타입은 *AnsiChar*와 동일한 *Char* 타입입니다. *Char* 타입의 구현은 변경될 수 있기 때문에 크기가 다양한 문자를 취급해야 하는 프로그램을 작성할 때는 하드 코딩된 상수를 사용하기 보다는 표준 함수인 *SizeOf*를 사용하는 것이 바람직합니다.

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

범위 검사가 활성화되지 않은 경우, 정수와 마찬가지로 문자값은 값을 증가시키거나 감소시킬 때 범위를 벗어 나면, 랩어라운드됩니다. 예를 들어, 다음과 같이 코드를 실행할 경우

```
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.

유니코드 문자에 대한 자세한 내용은 5-13 페이지의 "확장 문자 집합 정보"와 5-13 페이지의 "Null 종료 문자열 사용"을 참조하십시오.

부울 타입 (Boolean types)

네 개의 이미 정의된 부울 타입은 *Boolean*, *ByteBool*, *WordBool* 및 *LongBool*입니다. *Boolean* 타입을 가장 많이 사용합니다. 다른 타입들은 다른 랭귀지 및 운영 체제와의 호환성을 제공하기 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 *False*로 표시됩니다. 다음은 부울 타입의 관계를 보여 줍니다.

부울	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 *True*로 간주됩니다. *부울* 값이 있어야 할 위치에 그러한 값이 있으면 컴파일러는 자동으로 0이 아닌 순서 값을 *True*로 변환합니다.

위의 설명은 값 자체가 아닌 부울 값의 순서와 관련된 것입니다. 오브젝트 파스칼에서 부울 표현식은 정수 또는 실수와 동등하게 다룰 수 없습니다. 따라서 *X*가 정수 변수인 경우 다음 구문은

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

컴파일 오류를 생성합니다. 변수를 부울 타입으로 타입 변환하는 것은 신뢰성이 떨어지지만 다음의 두 구문 모두가 가능합니다.

```
if X <> 0 then ...;        { use longer expression that returns Boolean value }
var OK: Boolean            { use Boolean variable }
 ⋮
if X <> 0 then OK := True;
if OK then ...;
```

열거 타입 (Enumerated types)

열거 타입은 값을 나타내는 식별자를 단순히 나열함으로써 정렬된 값을 정의합니다. 값에는 특별한 고유의 의미가 없습니다. 열거 타입을 선언하려면 다음 구문을 사용합니다.

type *typeName* = (val_1, ..., val_n)

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

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

*Suit*라는 열거 타입을 정의하며, 이 열거 타입의 가능한 값은 *Club*, *Diamond*, *Heart* 및 *Spade*가 됩니다. 여기서 `Ord(Club)`의 반환값은 0이 되고, `Ord(Diamond)`의 반환값은 1이 됩니다.

열거 타입을 선언할 때는 각각의 *val*이 *typeName* 타입의 상수가 되도록 선언합니다. *val* 식별자가 동일한 유효 범위(scope) 내에서 다른 목적으로 사용되는 경우에는 이름 충돌이 발생합니다. 예를 들어 다음과 같이 타입을 선언한다고 가정하십시오.

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

공교롭게도 *Click*은 또한 *TControl* 및 이로부터 파생된 VCL 또는 CLX의 모든 객체에 대해 정의된 메소드 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 구문은 컴파일 오류가 발생합니다. 컴파일러는 프로시저의 유효 범위 내에서 *Click*을 *TForm*의 *Click* 메소드로 해석합니다. 이러한 문제는 식별자를 한정하여 해결할 수 있습니다. 즉, *TSound*가 *MyUnit*에서 선언된다면 다음과 같이 합니다.

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

보다 나은 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);
```

(val_1, ..., val_n) 구문을 마치 데이터 타입 이름인 것처럼 변수 선언에 직접 사용할 수 있습니다.

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

하지만 *MyCard*를 이런 방식으로 선언하면 이들 상수 식별자를 사용하여 동일한 유효 범위 내에서 다른 변수를 선언할 수 없습니다. 따라서

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

이 구문은 컴파일 오류가 발생합니다. 그러나,

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

이 구문은 다음과 같이 정확하게 컴파일 됩니다.

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

순서가 명시적으로 할당된 열거 타입

기본적으로 열거 값의 순서는 0에서 시작하고, 식별자가 타입 선언에 나열된 순서를 따릅니다. 선언의 일부 또는 모든 값에 순서를 명시적으로 할당하여 순서를 무시할 수 있습니다. 값에 순서를 할당하려면 식별자 뒤에 = *constantExpression*을 둡니다. 여기서 *constantExpression*은 정수값을 반환하는 상수 표현식입니다(5-41 페이지의 "상수 표현식" 참조). 예를 들면 다음과 같습니다.

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

이 구문은 *Size*라는 타입을 정의하고 이 타입의 가능한 값은 *Small*, *Medium* 및 *Large*가 됩니다. 여기서 `Ord(Small)`은 5를 반환하고, `Ord(Medium)`은 10을 반환하고, `Ord(Large)`는 15를 반환합니다.

사실 열거 타입은 최소 및 최대 값은 선언에서의 상수의 최소 및 최대 순서에 해당합니다. 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 이름이 있습니다. 나머지 값들은 *Pred*, *Succ*, *Inc* 및 *Dec* 같은 루틴 및 타입 변환을 통해서 액세스할 수 있습니다. 다음 예제에서, *Size*의 범위 내의 "익명" 값은 변수 *X*에 지정됩니다.

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

순서가 명시적으로 할당되지 않은 모든 값은 목록에서 이전 값의 순서보다 1이 큰 순서를 가집니다. 처음 값에 순서가 할당되지 않은 경우, 처음 값의 순서는 0입니다. 즉, 다음과 같은 선언에서,

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

*SomeEnum*은 2개의 값만 가집니다. Ord(e1)은 0을 반환하고, Ord(e2)는 1을 반환하고, Ord(e3) 역시 1을 반환합니다. *e2*와 *e3*은 동일한 순서를 갖고 있기 때문에 이들은 같은 값을 나타냅니다.

부분범위 타입 (Subrange types)

부분범위 타입은 다른 순서 타입 (즉, *기본(base) 타입*) 값들의 서브셋을 나타냅니다. *Low*와 *High*가 동일한 순서 타입의 상수 표현식이고 *Low*가 *High*보다 작은 경우, *Low*..*High* 형식의 모든 구문은 *Low*와 *High* 사이의 모든 값을 포함하는 부분범위 타입을 식별합니다. 예를 들어 다음 열거 타입을 선언합니다.

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

그런 다음 다음과 같이 부분범위 타입을 정의할 수 있습니다.

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

여기서 *TMyColors*는 *Green*, *Yellow*, *Orange*, *Purple* 및 *White* 값을 포함합니다.

숫자 상수와 문자(길이가 1인 문자열 상수)를 사용하여 부분범위 타입을 정의할 수 있습니다.

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

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

Low..*High* 구문 자체는 타입 이름과 같은 역할을 합니다. 따라서 이를 직접 변수 선언문에 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

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

1부터 500 사이 범위의 값이 될 수 있는 정수 변수를 선언합니다.

부분범위의 각 값의 순서는 기본(base) 타입에서 결정됩니다. 위의 첫 예제에서 *Color*가 *Green* 값을 가지는 변수라면 Ord(Color)는 *Color*가 *TColors* 타입이든,

TMyColors 타입이든 상관 없이 2를 반환합니다. 기본 타입이 정수 타입이든, 문자 타입이든 상관 없이 값은 부분범위의 시작 또는 끝으로 랩어라운드되지 않습니다. 부분 범위의 경계를 넘어선 값으로 증가시키거나 감소시키면 이 값은 기본 타입으로 변환됩니다. 따라서 다음과 같은 경우,

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

이 구문은 오류가 발생합니다. 반면,

```
⋮
I := 99;
Inc(I);
```

컴파일러의 범위 검사가 활성화되지 않으면 이 구문은 *I*에 값 100을 할당합니다.

부분범위 정의에서 상수 표현식을 사용하게 되면 구문상의 어려움이 있습니다. 모든 타입 선언에서 = 다음의 첫 의미 있는 문자가 왼쪽 괄호인 경우, 컴파일러는 열거 타입이 정의되는 것으로 간주합니다. 따라서 다음과 같은 코드에서는,

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

오류가 발생합니다. 타입 선언에서 다음과 같이 시작 괄호를 사용하지 않으면 이 문제를 해결할 수 있습니다.

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

실수 타입(Real types)

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

표 5.3 기본 실수 타입

타입	범위	유효 자릿수	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}	19-20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	19-20	8
Currency	-922337203685477.5808.. 922337203685477.5807	19-20	8

일반적인 *실수 타입*은 현재의 타입 구현에서는 *Double*과 동일합니다.

표 5.4 일반적인 실수 타입

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

참고 이전 버전의 오브젝트 파스칼에서는 6바이트 *Real48* 타입을 *실수*라고 했습니다. 이전의 6바이트 *실수* 타입을 사용하는 코드를 다시 컴파일하는 경우, 이를 *Real48*로 변경할 수 있습니다. 또한 {$REALCOMPATIBILITY ON} 컴파일러 지시문을 사용하여 *실수*를 다시 6바이트 타입으로 변환할 수 있습니다.

다음의 설명은 기본 실수 타입에 적용됩니다.

- *Real48*은 역 호환성을 위해 유지됩니다. 이 타입의 저장 형식은 Intel CPU 제품군의 원시 형식이 아니므로 다른 부동 소수점 타입보다는 성능이 낮아질 수 있습니다.
- *Extended*는 다른 실수 타입보다 큰 정밀도를 제공하지만, 이식성이 떨어집니다. 플랫폼 간에 공유할 데이터 파일을 작성하는 경우에는 *Extended* 사용에 주의하십시오.
- *Comp* (computational) 타입은 Intel CPU의 원시 형식이며 64비트 정수를 표시합니다. 하지만 이 타입은 순서 타입처럼 작동하지 않기 때문에 실수로 분류됩니다. (예를 들어, *Comp* 값은 증가시키거나 감소시킬 수 없습니다.) *Comp*는 역 호환성을 위해서만 유지됩니다. 성능을 향상시키려면 *Int64* 타입을 사용하십시오.
- *Currency*는 금액 계산에서 반올림 오류를 최소화하는 고정 소수점 데이터 타입입니다. 이 타입은 암시적으로 소수점 위치를 나타내는 4개의 최하위 숫자가 있는 64비트 정수로 저장됩니다. 할당문과 표현식에서 다른 실수 타입과 혼합되면, *Currency* 값은 자동적으로 10000으로 나눠지거나 곱해집니다.

문자열 타입(String types)

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

표 5.5 문자열 타입

타입	최대 길이	필요 메모리	용도
ShortString	255 문자	2에서 256바이트	역 호환성
AnsiString	~2^{31} 문자	4바이트에서 2기가 바이트	8비트(ANSI) 문자
WideString	~2^{30} 문자	4바이트에서 2기가 바이트	유니코드 문자, 다중 사용자 서버 및 다중 랭귀지 애플리케이션

*AnsiString*은 *긴 문자열*이라고 하며, 많이 사용되는 타입입니다.

문자열 타입은 지정문이나 표현식에서 사용할 수 있습니다. 컴파일러는 자동으로 필요한 변환을 수행합니다. 그러나 함수나 프로시저에 대한 참조로 전달된(**var** 및 **out** 매개변수 같이) 문자열은 적절한 타입이어야 합니다. 문자열은 다른 문자열 타입으로 명시적으로 타입 변환될 수 있습니다. 4-14 페이지의 "타입 변환"을 참조하십시오.

예약어 **string**은 일반적인 타입 식별자와 같은 기능을 합니다. 예를 들면, 다음과 같습니다.

```
var S:string;
```

문자열을 갖는 변수 *S*를 생성합니다. 기본 **{$H+}** 상태에서 컴파일러는 string 다음에 숫자가 들어 있는 괄호가 없으면 **string**을 *AnsiString*으로 해석합니다. **string**을 *ShortString* 으로 변환하려면 **{$H-}** 지시문을 사용하십시오.

표준 함수 *Length*는 문자열의 문자 수를 반환합니다. *SetLength* 프로시저는 문자열의 길이를 조정합니다. 자세한 내용은 온라인 도움말을 참조하십시오.

문자열 비교는 해당 위치에서의 문자 순서에 의해 정의됩니다. 길이가 서로 다른 두 문자열에서는, 짧은 문자열에 해당 문자가 없을 경우 길이가 긴 문자열의 문자들은 더 큰 값을 가집니다. 예를 들어, "AB"는 "A"보다 더 큽니다. 즉, `'AB' > 'A'`는 *True*를 반환합니다. 길이가 0인 문자열은 최저값을 가집니다.

배열에서와 같이 문자열 변수를 인덱싱할 수 있습니다. *S*가 문자열 변수이고 *i*는 정수 표현식인 경우, *S*[*i*]는 *S*의 *i*번째 문자 또는 엄밀히 말해서 *i*번째의 바이트를 나타냅니다. *ShortString*이나 *AnsiString*이면 *S*[*i*]는 *AnsiChar* 타입입니다. *WideString*이면 *S*[*i*]는 *WideChar* 타입입니다. `MyString[2] := 'A';` 문은 값 *A*를 *MyString*의 두 번째 문자에 할당합니다. 다음 코드는 표준 *UpCase* 함수를 사용하여 *MyString*을 대문자로 변환합니다.

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

이러한 방식으로 문자열을 인덱싱하는 것은 문자열의 마지막을 겹쳐쓰게 되어 액세스 위반을 일으킬 수도 있기 때문에 주의해야 합니다. 또한 긴 문자열 인덱스를 **var** 매개변수로서 전달하는 것도 비효율적인 코드가 될 수 있으므로 피하십시오.

변수에 문자열 상수의 값 또는 문자열을 반환하는 모든 다른 표현식을 할당할 수 있습니다. 문자열의 길이는 할당될 때마다 동적으로 변합니다. 예를 들면, 다음과 같습니다.

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';              { space }
MyString := '';               { empty string }
```

자세한 내용은 4-4 페이지의 "문자열" 및 4-9 페이지의 "문자열 연산자"를 참조하십시오.

짧은 문자열(Short strings)

*ShortString*은 문자의 길이가 0에서 255입니다. *ShortString*의 길이는 동적으로 변할 수 있지만, 이 문자열의 메모리는 정적으로 256바이트가 할당되어 있습니다. 첫 바이트에는 문자열의 길이가 저장되고, 나머지 255바이트는 문자을 저장하는데 사용됩니다. *S*가 *ShortString* 변수인 경우, `Ord(S[0])`은 `Length(S)`처럼 *S*의 길이를 반환합니다. *SetLength* 호출처럼 `S[0]`에 값을 할당하면 *S*의 길이가 변합니다. *ShortString*은 8비트 ANSI 문자를 사용하며 역 호환성을 위해서만 유지됩니다.

오브젝트 파스칼은 0에서 255 문자가 최대 길이이고 사실상 *ShortString*의 하위 타입인 짧은 문자열 타입을 지원합니다. 이러한 타입은 예약어 **string**에 괄호로 묶은 숫자를 추가하여 표시합니다. 예를 들면, 다음과 같습니다.

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

최대 길이가 100 문자인 *MyString*이라는 변수를 만듭니다. 이것은 다음 선언과 동일합니다.

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

이러한 방식으로 선언된 변수는 타입에서 필요로 하는 만큼의 메모리가 할당됩니다. 즉, 지정된 최대 길이에 1바이트를 더한 크기가 할당됩니다. 이 예제에서, 이미 정의된 *ShortString* 타입 변수에 대한 256바이트와 비교하면, *MyString*은 101바이트를 사용합니다.

짧은 문자열 변수에 값을 지정할 때, 타입에 대한 최대 길이를 초과하는 경우에는 문자열이 잘립니다.

표준 함수 *High*와 *Low*는 짧은 문자열 타입 식별자와 변수에서 작동합니다. *High*는 짧은 문자열 타입의 최대 길이를 반환하고 *Low*는 0을 반환합니다.

긴 문자열(Long strings)

*긴 문자열*이라고 하는 *AnsiString*은 사용 가능한 메모리에 의해서만 최대 길이가 제한되는 동적으로 할당된 문자열을 나타냅니다. 긴 문자열은 8비트 ANSI 문자를 사용합니다.

긴 문자열 변수는 4바이트 메모리를 사용하는 포인터입니다. 변수가 비어 있는 경우, 즉 문자열의 길이가 0인 경우에 포인터는 **nil**이고 문자열은 추가 메모리를 사용하지 않습니다. 변수가 비어 있지 않은 경우에는 문자열 값, 32비트 길이 지시자 및 32비트 참조 카운트를 포함하는 동적으로 할당된 메모리 블록을 가리킵니다. 이 메모리는 힙에 할당되지만 완전히 자동으로 관리되기 때문에 사용자 코드가 필요하지 않습니다.

긴 문자열 변수는 포인터이기 때문에 이들 중 두 개 이상은 추가로 메모리를 쓰지 않고도 동일한 값을 참조할 수 있습니다. 컴파일러는 이러한 점을 이용하여 리소스를 절약하고 할당을 더 빨리 수행합니다. 긴 문자열 변수가 없어지거나 새로운 값이 지정될 때마다 이전 문자열(변수의 이전 값)의 참조 카운트는 감소되고 새로운 값의 참조 카운트는 증가됩니다. 문자열의 참조 카운트가 0이 되면, 메모리는 해제됩니다. 이러한 프로세스를 *참조 카운팅*이라고 합니다. 문자열에서 한 문자의 값을 변경하기 위해 인덱싱을 사용하는 경우에 참조 카운트가 1보다 크면 문자열의 사본이 만들어집니다. 이것을 *copy-on-write* 의미론이라고 합니다.

WideString

WideString 타입은 16비트 유니코드 문자의 동적으로 할당된 문자열을 나타냅니다. 대부분의 경우 WideString 타입은 *AnsiString*과 비슷합니다.

Win32에서 *WideString*은 COM *BSTR* 타입과 호환됩니다. Borland 개발 도구는 *AnsiString* 값을 *WideString*으로 변환하는 기능을 지원하지만, 명시적으로 문자열을 *WideString*으로 타입 변환해야 합니다.

확장 문자 집합 정보

Windows와 Linux는 모두 *유니코드* 뿐만 아니라 *싱글바이트* 및 *멀티바이트* 문자 집합을 지원합니다. 싱글바이트 문자 집합(SBCS)에서 문자열의 각 바이트는 문자 하나를 나타냅니다. 영어권 운영 체제에서 사용하는 ANSI 문자 집합은 싱글바이트 문자 집합입니다.

멀티바이트 문자 집합(MBCS)에서, 일부 문자는 1바이트로 표시하고 그 외 문자들은 2바이트 이상으로 표시합니다. 멀티바이트 문자의 첫 바이트는 *선행 바이트*라고 합니다. 일반적으로 멀티바이트 문자 집합의 하위 128 문자는 7비트의 ASCII 문자에 매핑됩니다. 그리고 순서값이 127 이상인 모든 바이트는 멀티바이트 문자의 선행 바이트입니다. 싱글바이트 문자만 Null 값(#0)을 가질 수 있습니다. 멀티바이트 문자 집합, 특히 더블바이트 문자 집합(DBCS)은 아시아 언어권에서 널리 사용되며, Linux에서 사용되는 UTF-8 문자 집합은 유니코드의 멀티바이트 인코딩 방식입니다.

유니코드 문자 집합에서 각 문자는 2바이트로 표시합니다. 따라서 유니코드 문자열은 개별적인 바이트의 연속이 아니라 2바이트 워드의 연속입니다. 유니코드 문자와 문자열은 또한 *와이드 문자* 및 *와이드 문자열*이라고 합니다. 첫 256 유니코드 문자는 ANSI 문자 집합에 매핑됩니다. Windows 운영 체제는 유니코드(UCS-2)를 지원합니다. Linux 운영 체제는 UCS-2의 상위 집합인 UCS-4를 지원합니다. Delphi와 Kylix는 두 플랫폼 모두에서 UCS-2를 지원합니다.

오브젝트 파스칼은 *Char*, *PChar*, *AnsiChar*, *PAnsiChar* 및 *AnsiString* 타입을 통해 싱글바이트와 멀티바이트 문자 및 문자열을 지원합니다. 멀티바이트 문자열의 인덱싱은 *S*[*i*]가 *S*에서 *i*번째 바이트(항상 *i*번째의 문자를 나타내지는 않음)를 나타내기 때문에 신뢰성이 떨어집니다. 하지만 표준 문자열 처리 함수들은 로케일 특성 문자 순서를 구현하는 다른 멀티바이트 활성 대안이 있습니다. 멀티바이트 함수의 이름은 대개 *Ansi*로 시작합니다. 예를 들어, *StrPos*의 멀티바이트 버전은 *AnsiStrPos*입니다. 멀티바이트 문자 지원은 운영 체제에 종속적이며, 사용하는 운영체제 로케일을 기준으로 합니다.

오브젝트 파스칼은 *WideChar*, *PWideChar* 및 *WideString* 타입을 통해 유니코드 문자와 문자열을 지원합니다.

Null 종료 문자열 사용

C와 C++를 포함한 많은 프로그래밍 랭귀지들은 문자열 전용 데이터 타입이 부족합니다. 이들 랭귀지와 이러한 랭귀지로된 환경은 *Null 종료* 문자열에 의존합니다. Null 종료 문자열은 인덱스가 0부터 시작하고 NULL (#0)로 끝나는 문자 배열입니다. 이 배열은 길이 지시자가 없기 때문에 첫 NULL 문자는 문자열의 마지막을 표시합니다. *SysUtils* 유닛에서 오브젝트 파스칼 구문과 특수 루틴을 사용하면(8장 "표준 루틴 및

I/O" 참조) Null 종료 문자열을 사용하는 시스템에서 데이터를 공유해야하는 경우 이 문자열을 처리할 수 있습니다.

예를 들어, 다음의 타입 선언은 Null 종료 문자열을 저장하기 위해 사용할 수 있습니다.

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

확장 구문 활성화(**{$X+}**)를 사용하면 정적으로 할당되고 인덱스가 0부터 시작하는 문자 배열에 문자열 상수를 할당할 수 있습니다. 동적 배열은 이러한 용도로 사용하지 않습니다. 배열의 선언된 길이보다 더 짧은 문자열의 배열 상수를 초기화하면, 남은 문자들은 #0으로 설정됩니다. 포인터에 대한 자세한 내용은 5-18 페이지의 "배열"을 참조하십시오.

포인터 , 배열 및 문자열 상수 사용

Null 종료 문자열을 처리하면 포인터를 자주 사용하게 됩니다.(5-25 페이지의 "포인터와 포인터 타입(Pointer types)" 참조) *Char* 및 *WideChar* 값의 Null 종료 배열에 포인터를 나타내는 *PChar* 및 *PWideChar* 타입은 문자열 상수에 할당할 수 있는 타입입니다. 예를 들면, 다음과 같습니다.

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

*P*는 "Hello world!"의 Null 종료 사본이 포함된 메모리 영역을 가리킵니다. 이것은 다음과 동일합니다.

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

예를 들어, `StrUpper('Hello world!')` 처럼 문자열 상수를 *PChar* 또는 *PWideChar* 타입의 **const** 매개변수나 값을 가지는 모든 함수에 전달할 수 있습니다. *PChar*에 지정하는 것처럼 컴파일러는 문자열의 Null 종료 사본을 생성하고 함수에 해당 사본에 대한 포인터를 돌려 줍니다. 마지막으로 단독으로 또는 구조 타입의 문자열 리터럴로 *PChar* 또는 *PWideChar* 상수를 초기화할 수 있습니다. 예를 들면, 다음과 같습니다.

```
const
Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits:array[0.0.9] of Char;
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine'};
```

인덱스가 0부터 시작하는 문자 배열은 *PChar* 및 *PWideChar*와 호환됩니다. 포인터 값의 자리에 문자 배열을 사용하면, 컴파일러는 배열 첫 번째 요소의 주소에 해당하는 값을 가지는 포인터 상수로 배열을 변환합니다. 예를 들면, 다음과 같습니다.

```
var
  MyArray:array[0..32] of Char;
  MyPointer:PChar;
```

```
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

이 코드는 동일한 값으로 *SomeProcedure*를 두 번 호출합니다.

문자 포인터는 배열처럼 인덱싱할 수 있습니다. 위의 예제에서 `MyPointer[0]`는 *H*를 반환합니다. 인덱스는 역참조(dereference)하기 전에 포인터에 추가된 오프셋을 지정합니다. (*PWideChar* 변수의 경우, 인덱스는 자동적으로 두 배가 됩니다.) 그러므로 *P*는 문자 포인터이고, *P*`[0]`은 *P*^와 동일하며, 배열에서 첫 문자을 지정합니다. *P*`[1]`은 배열의 두 번째 문자를 지정합니다. *P*`[-1]`은 *P*`[0]`의 바로 왼쪽에 있는 "문자"를 지정합니다. *컴파일러는 이들 인덱스에서 범위 검사를 수행하지 않습니다.*

StrUpper 함수는 Null 종료 문자열을 통해 포인터 인덱싱을 반복적으로 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;
```

파스칼 문자열과 Null 종료 문자열의 혼합

표현식과 지정문에서 긴 문자열(*AnsiString* 값)과 Null 종료 문자열(*PChar* 값)을 혼합할 수 있습니다. 그리고 *PChar* 값을 매개변수가 긴 문자열인 함수나 프로시저에 전달할 수 있습니다. 할당문 *S* := *P*(여기서 *S*는 문자열 변수이고 *P*는 *PChar* 표현식)는 Null 종료 문자열을 긴 문자열에 복사합니다.

이항 연산에서 한 피연산자는 긴 문자열이고 다른 피연산자는 *PChar*인 경우 *PChar* 피연산자는 긴 문자열로 변환됩니다.

PChar 값을 긴 문자열로 타입 변환할 수 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 변환할 수도 있습니다. 다음 규칙이 적용됩니다.

- *S*가 긴 문자열 표현식인 경우, `PChar(`*S*`)`는 *S*를 Null 종료 문자열로 타입 변환하고, *S*의 첫 문자에 대한 포인터를 반환합니다.

Windows 인 경우 : 예를 들어 , *Str1* 및 *Str2* 가 긴 문자열인 경우 , 다음과 같이 Win32 API *MessageBox* 함수를 호출할 수 있습니다 .

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

(*MessageBox* 의 선언은 *Windows* 인터페이스 유닛에 있습니다 .)

Linux 인 경우 : 예를 들어 , *Str* 이 긴 문자열인 경우 , 다음과 같이 *opendir* 시스템 함수를 호출할 수 있습니다 .

```
opendir(PChar(Str));
```

(*opendir* 선언은 *Libc* 인터페이스 유닛에 있습니다 .)

- `Pointer(S)`를 사용하여 긴 문자열을 타입이 지정되지 않은 포인터로 변환할 수도 있습니다. 하지만 *S*가 비어 있는 경우, 타입 변환은 **nil**을 반환합니다.
- 긴 문자열 변수를 포인터로 타입 변환할 때, 포인터는 변수에 새로운 값이 지정되거나 유효 범위를 벗어나기 전까지는 유효한 상태로 남습니다. 다른 긴 문자열 표현식을 포인터로 타입 변환하는 경우, 포인터는 타입 변환이 수행되는 구문 내에서만 유효합니다.
- 긴 문자열 표현식을 포인터로 변환하는 경우, 포인터는 대개 읽기 전용으로 간주됩니다. 다음의 모든 조건이 만족될 때에만 긴 문자열을 수정하는데 포인터를 안전하게 사용할 수 있습니다.
  - 표현식 타입 변환은 긴 문자열 *변수*입니다 .
  - 문자열이 비어 있지 않습니다 .
  - 문자열이 고유합니다. 즉, 1개의 참조 카운트를 가집니다. 문자열이 고유한지를 확인하기 위해서는 *SetLength*, *SetString* 또는 *UniqueString* 프로시저를 호출합니다 .
  - 타입 변환 이후에 문자열이 수정되지 않았습니다 .
  - 수정된 문자는 모두 문자열 내에 있습니다. 포인터에서 범위를 벗어나는 인덱스를 사용하지 않도록 주의하십시오 .

WideString 값과 *PWideChar* 값을 혼합할 때도 동일한 규칙이 적용됩니다.

구조 타입(Structured types)

구조 타입의 인스턴스는 둘 이상의 값을 가집니다. 구조 타입에는 *클래스*, *클래스 참조* 및 *인터페이스* 타입 뿐만 아니라 *집합*, *배열*, *레코드* 및 *파일*이 포함됩니다. 클래스와 클래스 참조 타입에 대한 내용은 7장 "클래스 및 객체"를 참조하십시오. 인터페이스에 대한 내용은 10장 "객체 인터페이스"를 참조하십시오. 순서 값만 가지는 집합 타입을 제외하고, 구조 타입은 다른 구조 타입을 포함할 수 있으며, 구조의 제한되지 않는 단계를 가질 수 있습니다.

기본적으로 구조 타입의 값은 더 빨리 액세스하기 위해 워드 또는 더블 워드 경계로 정렬됩니다. 구조 타입을 선언할 때 압축 데이터 저장을 구현하기 위해 예약어 **packed**를 사용할 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 11장 "메모리 관리"를 참조하십시오.

집합

집합은 동일한 순서 타입을 가진 값의 컬렉션입니다. 값은 고유한 순서를 갖지 않으며, 집합에 두 번 포함되지도 않습니다.

집합 타입의 범위는 *기본(base) 타입*으로 불리는 특정 순서 타입의 멱집합입니다. 즉, 집합 타입의 가능한 값은 공집합을 포함한 기본 타입의 모든 서브셋입니다. 기본 타입은 257 이상의 값을 가질 수 없으며, 기본 타입의 순서는 0에서 255사이여야 합니다. 집합은 다음 형식 중 하나입니다.

```
set of baseType
```

여기서 *baseType*은 적절한 순서 타입이며, 집합 타입을 명시합니다.

기본 타입에 대한 크기 제한 때문에 집합 타입은 대개 부분범위로 정의됩니다. 예를 들어, 다음 선언을 보십시오.

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

값이 1부터 250사이의 정수 컬렉션인 *TIntSet*이라는 집합 타입을 생성합니다. 다음과 같은 경우도 동일한 결과를 얻습니다.

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

이 선언에서 다음과 같은 집합을 만들 수 있습니다.

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

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

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

집합 타입의 다른 예제는 다음과 같습니다.

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

in 연산자는 집합의 요소인지를 테스트합니다.

```
if 'a' in MySet then ... { do something } ;
```

모든 집합 타입은 []으로 표시되는 공집합을 가질 수 있습니다. 집합에 대한 자세한 내용은 4-13 페이지의 "집합 생성자" 및 4-10 페이지의 "집합 연산자"를 참조하십시오.

배열

배열은 동일한 타입(*기본 타입*)을 가진 요소의 인덱싱된 컬렉션을 나타냅니다. 각 요소는 고유한 인덱스를 갖기 때문에 배열은 집합과는 달리 같은 값을 두 번 이상 포함할 수 있습니다. 배열은 *정적*으로 또는 *동적*으로 할당할 수 있습니다.

정적 배열

정적 배열은 다음과 같은 구문으로 표시됩니다.

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

여기서 각 *indexType*은 범위가 2GB를 초과하지 않는 순서 타입입니다. *indexType*은 배열을 인덱싱하기 때문에 배열이 가질 수 있는 요소의 수는 *indexType* 크기의 곱에 의해 제한됩니다. 실제로, *indexType*은 대개 정수 부분범위입니다.

1차원 배열의 가장 간단한 예는 *indexType*이 하나만 있는 것입니다. 예를 들면, 다음과 같습니다.

```
var MyArray:array[1.0.100] of Char;
```

100문자 값의 배열을 가지는 *MyArray*라는 변수를 선언합니다. 이 선언에서, `MyArray[3]`은 *MyArray*의 세 번째 문자를 표시합니다. 정적 배열을 생성하고 모든 요소에 값을 할당하지 않으면 사용되지 않은 요소는 여전히 할당된 상태로 남아, 초기화되지 않은 변수 같은 임의의 데이터를 포함하게 됩니다.

다차원 배열은 배열들을 배열한 것입니다. 예를 들어,

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

다음과 동일합니다.

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

어떤 방식으로 *TMatrix*를 선언하든지간에, 이것은 500개의 실수 값의 배열을 나타냅니다. *TMatrix* 타입의 *MyMatrix* 변수를 `MyMatrix[2,45]`처럼 인덱싱하거나 `MyMatrix[2][45]`로 인덱싱 할 수 있습니다. 마찬가지로,

```
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 *High*는 배열 타입 식별자와 변수에서 작동합니다. 이들은 배열의 첫 번째 인덱스 타입의 최저 및 최고 경계를 반환합니다. 표준 함수 *Length*는 배열에서 첫 번째 차원의 요소 수를 반환합니다.

1차원의 압축된 *Char* 값을 가지는 정적 배열은 *압축 문자열*이라고 합니다. 압축 문자열 타입은 문자열 타입 및 요소 수가 동일한 다른 압축 문자열 타입과 호환됩니다. 5-34 페이지의 "타입 호환 및 구분"을 참조하십시오.

`array[0..x] of Char` 형태의 배열 타입은 *인덱스가 0부터 시작하는 문자 배열입니다*. 인덱스가 0부터 시작하는 문자 배열은 Null 종료 문자열을 저장하기 위해 사용되며, *PChar* 값과 호환됩니다. 5-13 페이지의 "Null 종료 문자열 사용"을 참조하십시오.

동적 배열

동적 배열은 크기나 길이가 고정되어 있지 않습니다. 그 대신, 값을 배열에 할당하거나 *SetLength* 프로시저에 전달할 때 동적 배열에 대한 메모리가 다시 할당됩니다. 동적 배열 타입은 다음과 같은 구문으로 표시됩니다.

```
array of baseType
```

예를 들면, 다음과 같습니다.

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

실수의 1차원 동적 배열을 선언합니다. 선언문은 *MyFlexibleArray*에 메모리를 할당하지 않습니다. 메모리에 배열을 생성하려면 *SetLength*를 호출합니다. 예를 들어, 위와 같은 선언일 때,

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

0에서 19까지 인덱싱된 20개의 실수 배열을 할당합니다. 동적 배열은 항상 정수로 인덱싱되며 0에서 시작합니다.

동적 배열 변수는 암시적인 포인터이며, 긴 문자열과 동일한 참조 카운팅 기법으로 관리합니다. 동적 배열을 해제하려면, 배열을 참조하는 변수에 **nil**을 할당하거나 *Finalize*에 변수를 전달합니다. 이 두 방법 모두에 참조가 없는 경우 배열을 해제합니다. 길이가 0인 동적 배열은 **nil** 값을 가집니다. 동적 배열 변수에 역참조 연산자(**^**)를 적용하거나 이를 *New* 또는 *Dispose* 프로시저에 전달하지 마십시오.

*X*와 *Y*가 동일한 동적 배열 타입 변수인 경우, *X* := *Y*는 *X*를 *Y*와 동일한 배열을 가리키도록 합니다. 이 연산을 수행하기 전에 *X*에 대해 메모리를 할당할 필요가 없습니다. 문자열과 정적 배열과는 달리 동적 배열은 작성되기 전에 자동으로 복사되지 않습니다. 예를 들어, 다음 코드를 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. (*A*와 *B*가 정적 배열이라면, A[0]은 여전히 1일 것입니다.)

예를 들어, MyFlexibleArray[2] := 7같은 동적 배열 인덱스에 대한 할당은 배열을 다시 할당하지 않습니다. 인덱스가 범위를 벗어나더라도 컴파일 시 보고되지 않습니다.

동적 배열 변수가 비교될 때는 배열 값이 아닌 이들의 참조가 비교됩니다. 따라서 다음 코드를 수행합니다.

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

A = B는 *False*가 반환되지만 A[0] = B[0]는 *True*가 반환됩니다.

동적 배열을 자르려면, 배열을 *SetLength* 또는 *Copy*에 전달하고 결과를 배열 변수에 역으로 지정합니다. 일반적으로 *SetLength* 프로시저가 더 빠릅니다. 예를 들어, *A*가 동적 배열인 경우, A := SetLength(A, 0, 20)은 *A*의 처음 20개의 요소를 제외한 나머지를 잘라 버립니다.

동적 배열을 할당한 후에는 이를 표준 함수 *Length*, *High* 및 *Low*로 전달할 수 있습니다. *Length*는 배열의 요소 수를 반환하고, *High*는 배열의 최대 인덱스(즉, *Length* − 1)를 반환하고, *Low*는 0을 반환합니다. 길이가 0인 배열의 경우, *High*는 −1을 반환(*High* < *Low*인 예외의 결과)합니다.

참고 일부 함수 및 프로시저 선언에서 배열 매개변수는 지정된 인덱스 타입 없이 array of *baseType*으로 나타납니다. 예를 들면, 다음과 같습니다.

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

이 코드는 크기나 인덱스 또는 정적이나 동적으로 할당되었는지에 상관없이 지정된 기본 타입의 모든 배열에서 함수가 작동된다는 것을 나타냅니다. 6−14 페이지의 "개방형 (open) 배열 매개변수"를 참조하십시오.

다차원 동적 배열

다차원 동적 배열을 선언하려면, 반복된 **array of ...** 구문을 사용합니다. 예를 들면, 다음과 같습니다.

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

이것은 2차원 문자열 배열을 선언합니다. 이 배열을 인스턴스화하려면, 정수 인수 두 개로 *SetLength*를 호출합니다. 예를 들어, *I*와 *J*가 정수값을 가지는 변수라면 다음과 같이 합니다.

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

*I*행 *J*열을 할당하고 Msgs[0,0]은 해당 배열의 요소를 표시합니다.

사각 타입이 아닌 다차원 동적 배열을 생성할 수 있습니다. 첫 단계는 *SetLength*를 호출하여 배열의 첫 번째 *n* 차원에 대한 매개변수를 전달하는 것입니다. 예를 들면, 다음과 같습니다.

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

*Ints*에 대해서 열이 아닌 행을 10개 할당합니다. 그런 후, 한 번에 하나 씩 길이가 다른 열을 할당할 수 있습니다. 예를 들면, 다음과 같습니다.

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

정수가 5개 있는 *Ints*의 세 번째 열을 만듭니다. 다른 열을 할당하지 않고도 Ints[2,4] := 6과 같이 세 번째 열에 값을 지정할 수 있습니다.

다음 예제는 동적 배열과 *SysUtils* 유닛에서 선언된 *IntToStr* 함수를 사용하여 문자열의 삼각 매트릭스를 만듭니다.

```
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;
```

배열 타입과 지정문

배열은 서로 동일한 타입인 경우에만 할당 호환성이 있습니다. 파스칼은 타입에 대해서 이름 동등성(name-equivalence)을 사용하기 때문에 다음 코드는 컴파일되지 않습니다.

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

지정이 작동되도록 하려면, 다음과 같이 변수를 선언합니다.

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

또는 다음과 같이 선언합니다.

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

레코드

레코드(일부 랭귀지에서는 *구조*와 유사한 의미를 지님)는 이질적인 요소 집합을 의미합니다. 각 요소는 *필드*라고 합니다. 레코드 타입의 선언은 각 필드에 대한 이름과 타입을 지정합니다. 레코드 타입 선언 구문은 다음과 같습니다.

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

여기서 *recordTypeName*은 유효한 식별자이며, 각 *type*은 타입을 의미하며, 각 *fieldList*는 유효한 식별자 또는 쉼표로 구분된 식별자 목록입니다. 마지막의 세미콜론은 옵션입니다.

예를 들어, 다음의 선언은 *TDateRec*라는 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 *TDateRec*은 세 개의 필드를 가지고 있습니다. 이들 필드는 *Year* 정수 값, *Month* 열거 타입 값, 그리고 1에서 31까지의 *Day* 정수 값입니다. *Year*, *Month* 및 *Day* 식별자는 *TDateRec*에 대한 *필드 지정자*이며, 이 식별자들은 변수처럼 동작합니다. 그러나 *TDateRec* 타입 선언은 *Year*, *Month* 및 *Day* 필드에 어떤 메모리도 할당하지 않습니다. 메모리는 다음과 같이 레코드를 인스턴스화하는 경우에 할당됩니다.

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

이 가변 타입 선언은 *TDateRec*의 두 가지 인스턴스, *Record 1*과 *Record 2*를 만듭니다.

필드 지정자를 레코드의 이름으로 한정하여 레코드의 필드에 액세스할 수 있습니다.

```
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 *Record2*에 복사할 수 있습니다.

```
Record2 := Record1;
```

필드 지정자의 유효 범위(scope)는 선언된 레코드에 제한되기 때문에 필드 지정자와 다른 변수간의 이름 충돌에 대해서는 걱정할 필요가 없습니다.

레코드 타입을 정의하는 대신 변수 선언에 직접 **record ...** 구문을 사용할 수 있습니다.

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

그러나 이와 같이 선언하면 비슷한 변수 그룹의 반복적 코딩을 줄이려는 레코드의 목적에 위배됩니다. 더욱이 이러한 방식으로 따로 선언된 레코드들은 구조가 동일한 경우라도 할당 호환성이 없습니다.

레코드의 가변 부분

레코드 타입은 **case** 문처럼 보이는 *가변* 부분을 가질 수 있습니다. 가변 부분은 레코드 선언의 마지막 필드에 와야 합니다.

가변 부분이 있는 레코드 타입을 선언하려면 다음과 같은 구문을 사용합니다.

```
type recordTypeName = record
  fieldList1:type1;
  ⋮
```

```
    fieldList_n:type_n;
  case tag: ordinalType of
    constantList_1: (variant_1);
    ⋮
    constantList_n: (variant_n);
  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 식별자가 될 수 있습니다. *tag*를 생략하려면, tag 다음에 오는 콜론(:)도 같이 생략해야 합니다.
- *ordinalType*은 순서 타입을 나타냅니다.
- 각 *constantList*는 *ordinalType* 타입의 값을 표시하는 상수이거나 상수를 쉼표로 구분한 목록입니다. 결합된 *constantList*에서는 값이 두 번 이상 나타날 수 없습니다.
- 각각의 *variant*는 레코드 타입의 주요 부분에서 *fieldList*:*type* 구문과 비슷한 선언이 쉼표로 구분된 목록입니다. 즉, *가변*은 다음 형식을 가집니다.

  ```
  fieldList_1:type_1;
    ⋮
  fieldList_n:type_n;
  ```

  여기서 각 *fieldList* 는 유효한 식별자 또는 쉼표로 구분된 식별자의 목록이며 , 각 *type* 은 타입을 나타내며 , 마지막 세미콜론은 옵션입니다 . *type* 은 긴 문자열 , 동적 배열 , 가변 (즉 , *가변* 타입) 또는 인터페이스일 수 없으며 , 긴 문자열 , 동적 배열 , 가변 또는 인터페이스를 포함하는 구조 타입일 수 없습니다 . 하지만 이들은 이들 타입에 대한 포인터일 수 있습니다 .

가변 부분이 있는 레코드는 구문적으로는 복잡하지만 의미상으로는 매우 간단합니다. 레코드의 가변 부분에 메모리에서 동일한 공간을 공유하는 몇 개의 *가변*이 있습니다. 언제든지 모든 필드의 모든 *가변*에 읽거나 쓸 수 있습니다. 하지만 한 *가변*의 필드에 쓰고 나서 또 다른 *가변*의 필드에 쓰려고 하면, 데이터를 겹쳐쓰게 될 수 있습니다. *tag*를 사용하는 경우에는 태그가 레코드의 비 가변 부분의 여분의 필드(*ordinalType* 타입)와 같은 기능을 수행합니다.

가변 부분은 두 가지 용도가 있습니다. 첫째는 여러 종류의 데이터에 대한 필드를 가지는 하나의 레코드 타입을 작성하려고 하지만 단일 레코드 인스턴스에서 필드를 모두 사용해야 할지에 대해서는 모르는 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 급여나 시간당 근무 수당을 받지만, 모든 직원이 둘 다 받는 것은 아닙니다. 그래서 *TEmployee* 인스턴스를 작성할 때는 충분한 메모리를 양쪽 필드에 모두 할당할 필요가 없습니다. 이런 경우, *가변 타입* 사이의 유일한 차이점은 필드 이름입니다

. 하지만, 이런 식으로 필드들은 매우 쉽게 여러가지 타입을 가질 수 있습니다. 이제 조금 더 복잡한 예제를 들어보겠습니다.

```
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;
```

각 레코드 인스턴스의 경우 컴파일러는 최대 *가변*의 모든 필드를 수용할 수 있는 충분한 메모리를 할당합니다. 옵션인 *tag*와 위의 마지막 예제의 *Rectangle*, *Triangle*과 같은 *constantList*는 컴파일러가 필드를 관리하는 방식에 아무런 역할도 수행하지 않습니다. 이들은 단지 프로그래머의 편의를 위해서만 여기에 존재합니다.

가변 부분을 사용하는 두 번째 이유는 이것을 사용하면 컴파일러가 타입 변환을 허용하지 않는 경우라도 동일한 데이터를 다른 여러 타입에 속하는 것처럼 취급할 수 있기 때문입니다. 예를 들어, 한 *가변*의 첫 필드로서 64비트의 *실수*를 사용하고, 다른 가변의 첫 필드에는 32비트의 *정수*를 사용하는 경우, 값을 *실수* 필드에 할당한 다음 이를 *정수* 필드의 값인 것처럼 정수 매개변수를 필요로하는 함수에 값을 전달하여 이 값의 처음 32비트를 다시 읽어낼 수 있습니다.

파일 타입(File types)

파일은 동일한 타입 요소의 정렬된 집합니다. 표준 I/O 루틴은 줄 단위로 구성된 문자를 포함하는 파일을 나타내는 이미 정의된 *텍스트 파일*이나 *텍스트* 타입을 사용합니다. 파일 입출력에 대한 자세한 내용은 8장 "표준 루틴 및 I/O"를 참조하십시오.

파일 타입을 선언하려면 다음 구문을 사용합니다.

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

여기서 *fileTypeName*은 유효한 식별자이고, *type*은 고정 크기 타입입니다. 포인터 타입(암시적이든 또는 명시적이든)은 허용되지 않습니다. 따라서 파일에는 동적 배열, 긴 문자열, 클래스, 객체, 포인터, 가변, 다른 파일 또는 이들 중 어느 것이라도 포함된 구조 타입이 포함될 수 없습니다.

예를 들면, 다음과 같습니다.

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

이름과 전화 번호를 기록하기 위해 파일 타입을 선언합니다.

또한 file of ... 구문을 변수 선언에 직접 사용할 수도 있습니다. 예를 들면, 다음과 같습니다.

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

file 단어 자체는 다음과 같이 타입이 지정되지 않은 파일을 나타냅니다.

```
var DataFile:file;
```

자세한 내용은 8−4 페이지의 "타입이 지정되지 않은 파일"을 참조하십시오.

파일은 배열이나 레코드에서는 허용되지 않습니다.

포인터와 포인터 타입(Pointer types)

포인터는 메모리 주소를 나타내는 변수입니다. 포인터가 다른 변수의 주소를 가질 때는 이를 메모리의 해당 변수의 위치 또는 이 위치에 저장된 데이터를 *가리킨다*고 합니다. 배열이나 다른 구조 타입의 경우, 포인터는 구조의 첫 번째 요소의 주소를 가리킵니다.

포인터는 포인터가 가리키는 주소에 저장된 데이터의 종류를 알려주기 위해 *타입이 지정*됩니다. 범용 *포인터* 타입은 모든 데이터에 대한 포인터를 나타내며, 좀 더 특수화된 포인터 타입은 단지 해당되는 특정 타입의 데이터만 참조합니다. 포인터는 4바이트 메모리를 사용합니다.

포인터 개요

포인터의 작동 원리를 이해하려면 다음 예제를 보십시오.

```
1   var
2     X, Y: Integer;   // X and Y are Integer variables
3     P: ^Integer;     // P points to an Integer
4   begin
5     X := 17;         // assign a value to X
6     P := @X;         // assign the address of X to P
7     Y := P^;         // dereference P; assign the result to Y
8   end;
```

2번 행에서는 *X*와 *Y*를 *정수* 타입의 변수로 선언합니다. 3번 행에서는 *P*를 *정수* 값에 대한 포인터로 선언합니다. 이것은 *P*가 *X* 또는 *Y*의 위치를 가리킬 수 있다는 의미입니다. 5번 행에서 *X*에 값을 지정하고, 6번 행에서는 *P*에 *X*의 주소를 할당합니다(@X로 표시). 마지막으로 7번 행에서는 *P*가 가리키는 위치에 있는 값(^P로 표시)을 *Y*에 지정합니다. 이 코드를 실행하고 나면 *X*와 *Y*는 같은 값 17을 갖게 됩니다.

@ 연산자는 위 예제에서 변수의 주소를 얻기 위해 사용되었지만, 함수와 프로시저에서도 사용할 수 있습니다.자세한 내용은 4-12 페이지의 "@ 연산자" 및 5-29 페이지의 "문장 및 표현식의 프로시저 타입"을 참조하십시오.

^ 기호는 두 가지의 목적을 갖고 있으며, 이 두 가지 목적 모두 다음 예제에서 설명합니다. ^ 기호가 타입 식별자 앞에 오는 경우

^typeName

typeName 타입 변수에 대한 포인터를 나타내는 타입을 표시합니다. ^ 기호가 포인터 변수 뒤에 오는 경우

*pointer^*

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

이 예제는 마치 단순한 지정문으로 한 변수의 값을 또 다른 무엇으로 복사하는 과정을 간접적으로 설명한 것처럼 보입니다. 하지만 포인터는 여러가지 이유로 유용합니다. 첫째, 포인터는 종종 코드에 명시적으로 드러나지 않고 사용되기 때문에 포인터를 이해하면 오브젝트 파스칼을 이해하는데 도움이 됩니다. 대규모의 동적으로 할당된 메모리 블록을 필요로 하는 모든 데이터 타입은 포인터를 사용합니다. 가령, 긴 문자열 변수는 클래스 변수와 마찬가지로 암시적인 포인터입니다. 또한 일부 고급 프로그래밍 기법을 적용하려면 포인터를 사용해야 합니다.

마지막으로 포인터는 때때로 오브젝트 파스칼의 엄격한 데이터 타입 지정을 우회할 수 있는 유일한 방법입니다. 범용 *포인터*로 변수를 참조하고, 보다 특수한 타입으로 *포인터*를 타입 변환하고, 이를 역참조하는 과정을 사용함으로써, 어떠한 변수로 저장된 데이터라도 타입에 상관없이 처리할 수 있습니다. 예를 들어, 다음 코드는 실수 변수에 저장된 데이터를 정수 변수에 지정합니다.

```
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 지정문에서는 데이터 변환없이 원래의 바이너리 데이터를 *R*에서 *I*로 단순히 복사합니다.

@ 연산의 결과를 지정하는 것 외에도 포인터에 값을 제공하기 위해 몇 가지 표준 루틴을 사용할 수 있습니다. *New*와 *GetMem* 프로시저는 메모리 주소를 기존의 포인터에 할당하며, *Addr* 함수와 *Ptr* 함수는 지정된 주소나 변수로 포인터를 반환합니다.

역참조된 포인터는 한정될 수 있으며, `P1^.Data^` 표현식에서와 같이 한정자로 사용할 수 있습니다.

예약어 **nil**은 모든 포인터에 할당할 수 있는 특별한 상수입니다. **nil**이 포인터에 할당되면, 포인터는 아무것도 참조하지 않습니다.

포인터 타입(Pointer types)

다음 구문을 사용하여 어떤 타입이든 상관없이 포인터를 선언할 수 있습니다.

```
type pointerTypeName = ^type
```

레코드나 다른 데이터 타입을 정의할 때도 또한 대부분 해당 타입에 대한 포인터를 정의하는 것이 일반적입니다. 이렇게 함으로써 대규모의 메모리 블록을 복사하지 않고도 타입의 인스턴스를 쉽게 처리할 수 있습니다.

표준 포인터 타입은 여러 가지 목적으로 사용됩니다. 다방면으로 가장 많이 사용되는 것이 *포인터*로서, 어떠한 종류의 데이터라도 가리킬 수 있습니다. 하지만 *포인터* 변수는 역참조될 수 없으며, *포인터* 변수 뒤에 **^** 기호를 사용하면 컴파일 오류가 발생합니다. *포인터* 변수로 참조되는 데이터에 액세스하려면 먼저 이를 다른 포인터 타입으로 변환한 다음 역참조합니다.

문자 포인터

기본 타입 *PAnsiChar*와 *PWideChar*는 각각의 *AnsiChar* 및 *WideChar* 변수에 대한 포인터를 나타냅니다. 일반적인 *PChar*는 *Char*에 대한 포인터를 나타냅니다. 즉, *AnsiChar*에 대한 포인터를 나타냅니다. 이들 문자 포인터는 Null 종료 문자열을 처리하기 위해 사용됩니다.(5-13 페이지의 "Null 종료 문자열 사용" 참조)

기타 표준 포인터 타입

*시스템*과 *SysUtils* 유닛은 일반적으로 사용되는 많은 표준 포인터 타입을 선언합니다.

표 5.6 시스템과 SysUtils 에서 선언된 선택된 포인터 타입

포인터 타입	가리키는 변수 타입
PAnsiString, PString	*AnsiString*
PByteArray	*SysUtils*에 선언된 *TByteArray*. 배열 액세스를 위해 동적으로 할당된 메모리의 타입 변환을 위해 사용됩니다.
PCurrency, PDouble, PExtended, PSingle	*Currency, Double, Extended, Single*
PInteger	*Integer*
POleVariant	*OleVariant*
PShortString	*ShortString*. 이전의 *PString* 타입을 사용하는 레거시 코드를 포팅할 때 유용합니다.
PTextBuf	*SysUtils*에 선언된 *TTextBuf*. *TTextBuf*는 *TTextRec* 파일 레코드의 내부 버퍼 타입입니다.
PVarRec	*시스템*에 선언된 *TVarRec*.
PVariant	*Variant*
PWideString	*WideString*
PWordArray	*SysUtils*에 선언된 *TWordArray*. 2바이트 값의 배열에 대한 동적으로 할당된 메모리를 타입 변환하기 위해 사용됩니다.

프로시저 타입(Procedural types)

프로시저 타입은 프로시저와 함수를 변수에 할당하거나 다른 프로시저와 함수에 전달할 수 있는 값처럼 취급합니다. 예를 들어, 두 개의 정수 매개변수를 사용하고 정수를 반환하는 *Calc*라는 함수를 정의한다고 가정합니다.

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

변수 *F*에 *Calc* 함수를 할당할 수 있습니다.

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

프로시저나 함수의 헤더에서 **procedure** 또는 **function** 뒤에 식별자를 제거하면 프로시저타입의 이름이 남게 됩니다. 이러한 타입 이름을 위의 예제에서와 같이 변수 선언에 직접 사용하거나 또는 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;            { F is a parameterless function that returns an
integer }
  Proc: TProcedure;               { Proc is a parameterless procedure }
  SP: TStrProc;                   { SP is a procedure that takes a string parameter }
  M: TMathFunc;                   { M is a function that takes a Double (real)
parameter
                                     and returns a Double }
procedure FuncProc(P: TIntegerFunction};  { FuncProc is a procedure whose only
parameter
                                          is a parameterless integer-valued
function }
```

위의 변수들은 모두 *프로시저 포인터*입니다. 즉, 프로시저나 함수의 주소를 가리키는 포인터입니다. 인스턴스 객체의 메소드를 참조하려는 경우에는(7장 "클래스 및 객체" 참조), 프로시저 타입 이름에 **of object**를 추가해야 합니다. 예를 들면, 다음과 같습니다.

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

이들 타입은 *메소드 포인터*를 나타냅니다. 메소드 포인터는 한 쌍의 포인터로 이루어지며, 첫 번째 포인터는 메소드의 주소를 저장하고, 두 번째 포인터는 메소드가 속한 객체에 대한 참조를 저장합니다. 다음과 같이 선언했다고 가정해 보십시오.

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

다음과 같은 할당문을 만들 수 있습니다.

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

다음의 경우 두 개의 프로시저 타입이 서로 호환됩니다.

- 동일한 호출 규칙을 사용하는 경우
- 반환값이 동일한 경우, 또는 반환값이 없는 경우
- 해당 위치의 매개변수의 타입이 동일하고, 매개변수의 수가 동일한 경우 (매개변수 이름은 중요하지 않음)

프로시저 포인터 타입은 항상 메소드 포인터 타입과는 호환되지 않습니다. **nil** 값은 모든 프로시저 타입에 지정될 수 있습니다.

중첩된 프로시저와 함수(다른 루틴 내에 선언된 루틴)는 절차 값으로서 사용될 수 없으며, 프로시저나 함수를 미리 정의할 수 없습니다. 절차 값으로서 *Length* 같은 이미 정의된 루틴을 사용하려면 이에 대한 랩퍼를 작성합니다.

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

문장 및 표현식의 프로시저 타입

할당문의 왼쪽에 프로시저 변수가 있으면, 컴파일러는 오른쪽에 프로시저 값이 있을 것으로 예상합니다. 할당문은 왼쪽의 변수를 오른쪽에 표시된 함수나 프로시저에 대한 포인터로 만듭니다. 하지만 다른 구문에서 프로시저 변수를 사용하면 참조되는 프로시저나 함수를 호출합니다. 프로시저 변수를 사용하여 매개변수를 전달할 수도 있습니다.

```
var
  F: function(X:Integer):Integer;
  I:Integer;
function SomeFunction(X:Integer):Integer;
 ⋮
F := SomeFunction;  // assign SomeFunction to F
I := F(4);          // call function; assign result to I
```

할당문에서 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;  // assign SomeFunction to F
G := F;             // copy F to G
I := G;             // call function; assign result to I
```

첫 번째 할당문은 프로시저 값을 *F*에 지정합니다. 두 번째 할당문은 값을 또 다른 변수로 복사합니다. 세 번째 할당문은 참조되는 함수를 호출하고 결과를 *I*에 지정합니다. *I*는 프로시저 변수가 아닌 정수 변수이기 때문에, 마지막 할당에서 실제로 함수(정수를 반환하는)가 호출됩니다.

경우에 따라서 프로시저 변수를 해석하는 방법이 명확하지 않을 수도 있습니다. 다음 구문을 보십시오.

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

이 경우에 *F*로 인해 함수가 호출됩니다. 컴파일러는 *F*가 가리키는 함수를 호출한 다음 *MyFunction* 함수를 호출하고, 결과를 비교합니다. 표현식 내에서 프로시저 변수를 사용할 때마다 참조되는 프로시저나 함수 호출이 발생한다는 것이 규칙입니다. *F*가 값을 반환하지 않는 프로시저를 참조하거나 *F*가 매개변수를 사용하는 함수를 참조하는 경우에는 위의 구문은 컴파일 오류가 발생합니다. *F*의 절차 값을 *MyFunction*과 비교하려면 다음을 사용합니다.

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

`@F`는 주소를 포함하는 타입이 지정되지 않은 포인터 변수로 *F*를 변환하고, `@MyFunction`은 *MyFunction*의 주소를 반환합니다.

저장된 주소가 아닌 절차 변수의 메모리 주소를 얻으려면, **@@**를 사용합니다. 예를 들어, `@@F`는 *F*의 주소를 반환합니다.

프로시저 변수에 타입이 지정되지 않은 포인터 값을 할당하기 위해 **@** 연산자를 사용할 수도 있습니다. 예를 들면, 다음과 같습니다.

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

GetProcAddress 함수를 호출하고 결과로 *StrComp*를 가리킵니다.

모든 프로시저 변수는 아무것도 가리키지 않는 **nil** 값을 가질 수 있습니다. 하지만 **nil** 값의 프로시저 변수를 호출하려고 시도하면 오류가 발생합니다. 프로시저 변수가 할당되었는지 검사하려면 표준 함수*Assigned*를 사용합니다.

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

가변 타입(Variant types)

경우에 따라 타입이 달라지거나 컴파일 시에 결정할 수 없는 데이터를 처리해야 하는 경우가 있습니다. 이런 경우, 런타임 시 변경될 수 있는 값을 나타내는 *가변* 타입의 변수와 매개변수를 사용할 수 있습니다. 가변 타입은 유연하지만, 일반적인 변수보다 많은 메모리를 사용하며, 경계가 정적인 타입보다 연산이 느립니다. 뿐만 아니라, 가변 타입에서는 암시적인 연산으로 인해, 일반적인 변수를 사용할 경우에는 컴파일 시에 발견할 수 있는 유사한 실수가 런타임 오류로 나타날 수 있습니다. 또한 사용자 정의 가변 타입을 만들 수도 있습니다.

가변 타입은 레코드, 집합, 정적 배열, 파일, 클래스, 클래스 참조 및 포인터를 제외한 모든 타입의 값을 가질 수 있습니다. 다시 말해 가변은 구조 타입과 포인터를 제외한 타입이 될 수 있다는 뜻입니다. 가변 타입은 COM과 CORBA 객체를 가질 수 있습니다. 이 두 객체의 메소드와 속성을 가변 타입을 통해 액세스할 수 있습니다.(10장 "객체 인터페이스" 참조) 가변 타입은 동적 배열을 가질 수 있으며, *가변 타입 배열*과 같은 특수한 종류의 정적 배열을 가질 수 있습니다.(5-33 페이지의 "가변 타입 배열" 참조) 가변 타

입은 다른 가변 타입 및 표현식 및 할당문의 정수, 실수, 문자열, 부울 값과 혼합될 수 있습니다. 컴파일러는 자동으로 타입 변환됩니다.

문자을 포함하는 가변 타입을 인덱싱할 수 없습니다. 즉, *V*가 문자열 값을 갖는 가변 타입인 경우, V[1] 구문에서 런타임 오류가 발생합니다

가변 타입은 16바이트의 메모리를 사용하고 코드에서 지정된 타입의 타입 코드 및 값 또는 값에 대한 포인터로 구성됩니다. 모든 가변 타입은 생성 시 특수 값 *Unassigned*로 초기화됩니다. 특수 값 *Null*은 알려지지 않거나 누락된 데이터를 나타냅니다.

표준 함수 *VarType*은 가변 타입 코드를 반환합니다. *varTypeMask* 상수는 *VarType*의 반환 값에서 코드를 추출하기 위해 사용되는 비트 마스크입니다. 예를 들면 다음과 같습니다.

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

이 구문은 *V*에 *Double* 타입이나 *Double* 타입 배열이 있는 경우 *True*를 반환합니다. (마스크는 가변에 배열이 있는지 여부를 나타내는 첫 번째 비트를 단순히 숨깁니다.) *시스템* 유닛에서 정의된 *TVarData* 레코드 타입은 가변 타입을 타입 변환하기 위해 사용될 수 있으며 내부 표현 방식에 대한 액세스를 얻을 수 있습니다. 코드 목록은 *VarType*에 대한 온라인 도움말을 참조하십시오. 새로운 타입 코드는 오브젝트 파스칼 다음 버전에 추가될 수도 있습니다.

가변 타입 변환

가변 타입은 모든 정수 타입, 실수 타입, 문자열 타입, 문자 타입 및 부울 타입에 지정할 수 있는 타입입니다. 표현식은 명시적으로 가변 타입으로 타입 변환될 수 있습니다. 그리고 *VarAsType*과 *VarCast* 표준 루틴은 가변의 내부 표현 방식을 변경하기 위해 사용될 수 있습니다. 다음 코드는 가변 타입이 사용되는 방식과 가변 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; { integer value }
  V2 := 1234.5678; { real value }
  V3 := 'Hello world!'; { string value }
  V4 := '1000'; { string value }
  V5 := V1 + V2 + V4; { real value 2235.5678}
  I := V1; { I = 1 (integer value) }
  D := V2; { D = 1234.5678 (real value) }
  S := V3; { S = 'Hello world!' (string value) }
  I := V4; { I = 1000 (integer value) }
  S := V5; { S = '2235.5678' (string value) }
end;
```

컴파일러는 다음의 규칙에 따라 타입 변환을 수행합니다.

표 5.7 가변 타입 변환 규칙

소스 \ 대상	정수	실수	문자열	문자	부울
정수	정수타입으로 변환	실수로 변환	문자열 표현 방식으로 변환	문자열과 동일 (왼쪽)	0이면 *False* 반환, 0이 아니면 *True* 반환
실수	가장 가까운 정수로 반올림	실수타입으로 변환	지역 설정을 사용하여 문자열 표현 방식으로 변환	문자열과 동일 (왼쪽)	0이면 *False* 반환, 0이 아니면 *True* 반환
문자열	정수로 변환, 필요 시 잘라냄. 문자열이 숫자가 아닌 경우에는 예외 발생	지역 설정을 사용하여 실수로 변환. 문자열이 숫자가 아닌경우에는 예외 발생	문자열/문자 타입으로 변환	문자열과 동일 (왼쪽)	문자열이 "false" (대소문자 구분 안함)이거나 0과 같은 숫자 문자열인 경우에는 *False*를 반환하고, 문자열이 "true" 또는 0이 아닌 숫자 문자열인 경우에는 *True*를 반환. 그 외에는 예외 발생.
문자	문자열과 동일 (위쪽)	문자열과 동일 (위쪽)	문자열과 동일 (위쪽)	문자열 대 문자열과 동일	문자열과 동일 (위쪽)
부울	*False* = 0, *True* = −1 (바이트이면 255)	*False* = 0, *True* = −1	*False* = "0", *True* = "−1"	문자열과 동일 (왼쪽)	*False* = *False*, *True* = *True*
Unassigned	0 반환	0 반환	빈 문자열 반환	문자열과 동일 (왼쪽)	*False* 반환
Null	예외 발생	예외 발생	예외 발생	문자열과 동일 (왼쪽)	예외 발생

범위를 벗어난 할당은 대상 변수가 가질 수 있는 범위에서 변수가 최고값을 얻는 결과가 자주 발생합니다. 잘못된 할당문이나 타입 변환 시 *EVariantError* 예외가 발생합니다.

특수 변환 규칙은 *시스템* 유닛에서 선언된 *TDateTime* 실수 타입에 적용됩니다. *TDateTime*이 다른 타입으로 변환될 때는 보통 *Double* 타입으로 취급됩니다. 정수, 실수 또는 부울이 *TDateTime*으로 변환될 때, 먼저 *Double*로 변환된 다음 날짜 시간 값으로 읽습니다. 문자열이 *TDateTime*으로 변환될 때, 지역 설정을 사용하여 날짜 시간 값으로 해석합니다. *Unassigned* 값이 *TDateTime*으로 변환될 때는 실수나 정수 0과 같이 취급합니다. *Null* 값을 *TDateTime*으로 변환하면 예외가 발생합니다.

Windows에서 가변 타입이 COM 객체를 참조하는 경우, 이를 변환하려고 하면 객체의 기본 속성을 읽고나서 해당 값을 요청된 타입으로 변환합니다. 객체에 기본 속성이 없는 경우에는 예외가 발생합니다.

표현식의 가변 타입

^를 제외한 모든 연산자는 가변 타입 피연산자를 사용합니다. 가변 연산자는 *가변*값을 반환하고, 하나 또는 두 피연산자가 모두 *Null*인 경우에는 *Null*을 반환하고, 그리고 하나 또는 두 피연산자가 모두 *타입이 지정되지 않았으면* 예외가 발생합니다. 이항 연산의 경우, 단지 하나의 피연산자만 가변 타입이면 다른 피연산자가 가변 타입으로 변환됩니다.

연산의 반환 타입은 피연산자에 의해 결정됩니다. 일반적으로 경계가 정적인 타입의 피연산자에 적용되는 동일한 규칙이 가변 타입에 적용됩니다. 예를 들어, *V1*과 *V2*가 정수 및 실수 값을 가지는 가변 타입인 경우, `V1 + V2`는 실수값의 가변 타입을 반환합니다.(4-6 페이지의 "연산자" 참조) 하지만 가변으로, 정적으로 타입이 지정된 표현식에서는 허용되지 않는 값 조합에 대한 이항 연산을 수행할 수 있습니다. 가능한 경우 컴파일러는 표 5.7에 요약된 규칙에 따라 일치하지 않는 가변 타입을 변환합니다. 예를 들어, *V3*과 *V4*가 숫자 문자열과 정수를 가지는 가변 타입인 경우, 표현식 `V3 + V4`는 정수값의 가변 타입을 반환합니다. 숫자 문자열은 연산이 수행되기 전에 정수로 변환됩니다.

가변 타입 배열

일반적인 정적 배열은 가변 타입으로 변환할 수 없습니다. 대신, 표준 함수 *VarArrayCreate* 또는 *VarArrayOf* 중 하나를 호출하여 *가변 타입 배열*을 생성할 수 있습니다. 예를 들면, 다음과 같습니다.

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

정수의 가변 타입 배열(길이 10)을 생성하고 이를 가변 타입 변수 *V*에 할당합니다. 이 배열은 이 `V[0]`, `V[1]` 등을 사용하여 인덱싱할 수 있습니다. 하지만 가변 타입 배열 요소를 **var** 매개변수로서 전달하는 것은 불가능합니다. 가변 타입 배열은 항상 정수로 인덱싱됩니다.

VarArrayCreate 호출의 두 번째 매개변수는 배열의 기본 타입에 대한 타입 코드입니다. 이들 코드의 목록은 *VarType*에 대한 온라인 도움말을 참조하십시오. *varString* 코드를 절대로 *VarArrayCreate*로 전달하지 마십시오. 문자열의 가변 타입 배열을 생성하려면 *varOleStr*을 사용합니다.

가변은 크기, 차원 및 기본 타입이 다른 가변 타입 배열을 가질 수 있습니다. 가변 타입 배열의 요소는 *ShortString* 및 *AnsiString*을 제외한 가변에서 허용되는 모든 타입일 수 있으며, 배열의 기본 타입이 *Variant*인 경우는 요소가 이질적일 수도 있습니다. 가변 타입 배열의 크기를 조정하려면 *VarArrayRedim* 함수를 사용합니다. 가변 타입 배열에서 작동되는 다른 표준 루틴에는 *VarArrayDimCount*, *VarArrayLowBound*, *VarArrayHighBound*, *VarArrayRef*, *VarArrayLock* 및 *VarArrayUnlock*이 있습니다.

가변 타입 배열을 포함하는 가변이 다른 가변에 할당되거나 값 매개변수로서 전달되는 경우에는 전체 배열이 복사됩니다. 이 연산은 메모리를 비효율적으로 사용하기 때문에 꼭 필요하지 않다면 이 연산을 수행하지 마십시오.

OleVariant

OleVariant 타입은 Windows와 Linux 플랫폼 모두에 존재합니다. *Variant*와 *OleVariant*의 주요 차이점은 *Variant*는 현재 애플리케이션에서 어떻게 처리해야 할지 알고 있는 데이터 타입만을 포함할 수 있다는 것입니다. *OleVariant*는 프로그램 사이에 또는 네트워크를 통해 어느 한쪽이 데이터 처리 방법을 알고 있는지에 대해 걱정할 필요 없이 전달될 수 있는 데이터 타입을 의미하는 Ole Automation과 호환되도록 정의된 데이터 타입만 포함할 수 있습니다.

사용자 정의 데이터(Pascal 문자열이나 새로운 사용자 정의 가변 타입 중의 하나와 같은)를 포함하는 *Variant*를 *OleVariant*에 할당하면, 런타임 라이브러리는 *Variant*를 *OleVariant* 표준 데이터 타입(예를 들어, Pascal 문자열을 Ole BSTR 문자열로 변환) 중 하나로 변환하려고 시도합니다. 예를 들어, *AnsiString*을 포함하는 가변이 *OleVariant*에 할당되면, *AnsiString*은 *WideString*이 됩니다. 그리고 *Variant*를 *OleVariant* 함수 매개변수로 전달하는 경우에도 동일합니다.

타입 호환 및 구분

어떤 연산이 어떤 표현식에서 수행될 수 있는지를 이해하려면, 타입과 값들 사이의 여러 가지 호환성을 구별할 수 있어야 합니다. 여기에는 *타입 구분*, *타입 호환* 및 *할당 호환*이 포함됩니다.

타입 구분

타입 구분은 거의 직접적입니다. 하나의 타입 식별자를 다른 타입 식별자를 사용하여 한 정자 없이 선언하는 경우에는 동일한 타입을 표시하게 됩니다. 따라서, 다음과 같이 선언했다고 가정해 보십시오.

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

T1, *T2*, *T3*, *T4* 및 *Integer*는 모두 동일한 타입을 표시합니다. 구분되는 타입을 만들려면 선언에서 **type**을 반복해야 합니다. 예를 들면, 다음과 같습니다.

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

*Integer*와는 다른 새로운 *TMyInteger*라는 타입을 생성합니다.

타입 이름과 같은 기능을 하는 랭귀지 구문은 선언할 때마다 다른 타입을 표시합니다. 그러므로 다음 선언은,

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

두 개의 서로 다른 타입인 *TS1*과 *TS2*를 생성합니다. 이와 비슷하게 다음의 변수 선언,

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

이 구문은 두 개의 서로 다른 타입의 변수를 생성합니다. 동일한 타입의 변수들을 생성하려면 다음을 사용합니다.

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

또는

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

타입 호환성

모든 타입은 그 자체로 호환됩니다. 두 개의 서로 다른 타입은 다음 조건 중 하나 이상을 만족할 경우에만 호환됩니다.

- 양쪽 모두 실수타입인 경우
- 양쪽 모두 정수 타입인 경우
- 한 타입이 다른 타입의 부분범위 타입인 경우
- 양쪽 모두 동일한 타입의 부분범위 타입인 경우
- 양쪽 모두 호환가능한 기본(base) 타입을 가진 집합 타입인 경우
- 양쪽 모두 구성 요소의 숫자가 동일한 압축 문자열 타입인 경우
- 하나는 문자열 타입이고 다른 하나는 문자열, 압축 문자열 또는 *Char* 타입인 경우
- 하나는 *Variant*이고 다른 하나는 정수, 실수, 문자열, 문자 또는 부울 타입인 경우
- 양쪽 모두 클래스, 클래스 참조 또는 인터페이스 타입이고, 하나가 다른 하나로부터 파생인 경우
- 하나가 *PChar* 또는 *PWideChar*이고 다른 하나는 `array[0..n] of Char` 타입의 인덱스가 0부터 시작하는 문자 배열인 경우
- 하나는 *포인터*(타입이 지정되지 않은 포인터)이고 다른 하나는 임의의 포인터 타입인 경우
- 양쪽 모두 동일한 타입에 대한 (타입이 지정된) 포인터이고 **{$T+}** 컴파일러 지시문이 영향을 주는 경우
- 양쪽 모두 동일한 결과 타입이 있고, 매개변수 수가 동일하고, 각각의 위치의 매개변수 사이에 타입 구분이 있는 프로시저 타입인 경우

할당 호환

할당 호환은 대칭적 관계가 아닙니다. 표현식의 값이 *T1*의 범위 내에 있고 다음 조건 중 하나 이상이 만족되면 *T2* 타입의 표현식은 *T1* 타입의 변수에 할당될 수 있습니다.

- *T1*과 *T2*가 동일한 타입이고, 어떤 수준에서든 파일 타입이나 파일을 포함하는 구조 타입이 아닌 경우
- *T1*과 *T2*가 호환 가능한 순서 타입인 경우
- *T1*과 *T2*가 모두 실수 타입인 경우
- *T1*은 실수 타입이고 *T2*는 정수 타입인 경우
- *T1*이 *PChar*이거나 임의의 문자열 타입이고 표현식이 문자열 상수인 경우
- *T1*과 *T2*가 모두 문자열 타입인 경우
- *T1*이 문자열 타입이고 *T2*는 *Char* 또는 압축 문자열 타입인 경우
- *T1*이 긴 문자열이고 *T2*는 *PChar*인 경우
- *T1*과 *T2*가 호환 가능한 압축 문자열 타입인 경우
- *T1*과 *T2*가 호환 가능한 집합 타입인 경우
- *T1*과 *T2*가 호환 가능한 포인터 타입인 경우
- *T1*과 *T2*가 모두 클래스, 클래스 참조 또는 인터페이스 타입이고, *T2*가 *T1*에서 파생된 경우
- *T1*이 인터페이스 타입이고 *T2*가 *T1*을 구현하는 클래스 타입인 경우
- *T1*이 *PChar* 또는 *PWideChar*이고 *T2*가 `array[0..`*n*`] of Char` 타입의 인덱스가 0부터 시작하는 문자 배열인 경우
- *T1*과 *T2*가 호환 가능한 프로시저 타입인 경우 (함수 또는 프로시저 식별자는 특정 지정문에서 프로시저 타입의 표현식으로 취급됩니다. 5-29 페이지의 "문장 및 표현식의 프로시저 타입"을 참조하십시오.)
- *T1*은 *가변 타입*이고 *T2*는 정수, 실수, 문자열, 문자, 부울 또는 인터페이스 타입인 경우
- *T1*은 정수 타입, 실수 타입, 문자열 타입, 문자 타입 또는 부울 타입이고 *T2*는 *가변 타입*입니다.
- *T1*이 *IUnknown* 또는 *IDispatch* 인터페이스 타입이고 *T2*가 *가변 타입*인 경우 (가변의 타입 코드는 *T1*이 *IUnknown*인 경우에는 *varEmpty*, *varUnknown* 또는 *varDispatch*여야 하며, *T1*이 *IDispatch*인 경우에는 *varEmpty* 또는 *varDispatch*여야 합니다.)

타입 선언

타입 선언에서 타입을 표시하는 식별자를 지정합니다. 타입 선언 구문은 다음과 같습니다.

```
type newTypeName = type
```

여기서 *newTypeName*은 유효한 식별자입니다. 예를 들어, 다음과 같이 타입 선언했다고 가정해 보십시오.

```
type TMyString = string;
```

변수 선언을 다음과 같이 할 수 있습니다.

```
var S: TMyString;
```

타입 식별자의 유효 범위에 타입 선언 자체는 포함되지 않습니다(포인터 타입 제외). 따라서 예를 들어 그 자체를 재귀적으로 사용하는 레코드 타입은 정의할 수 없습니다.

기존의 타입과 동일한 타입을 선언할 때는 컴파일러가 새로운 타입 식별자를 기존의 식별자에 대한 알리아스로 간주합니다. 따라서, 다음과 같이 선언했다고 가정해 보십시오.

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

*X*와 *Y*는 동일한 타입입니다. 런타임 시 *TValue*와 *Real*을 구별할 수 있는 방법이 없습니다. 이것은 일반적으로 거의 없는 결과이지만, 새로운 타입을 정의하는 데 목적이 있다면 런타임 정보를 이용합니다. 예를 들면, 특정 타입의 속성과 속성 편집기를 연결시킬 때 "다른 이름"과 "다른 타입" 사이의 구별은 중요합니다. 이런 경우에는 다음 구문을 사용합니다.

```
type newTypeName = type type
```

예를 들면, 다음과 같습니다.

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

컴파일러에게 *TValue*라는 새로운 구별된 타입을 만들도록 합니다.

변수

변수는 런타임 시에 값이 변경될 수 있는 식별자입니다. 달리 말하면, 변수는 메모리의 위치에 대한 이름입니다. 이 이름을 사용하면 해당 메모리 위치에서 읽거나 쓸 수 있습니다. 변수는 데이터의 컨테이너와 비슷합니다. 그리고 변수는 타입을 지정할 수 있기 때문에 컴파일러에게 변수가 갖고 있는 데이터를 처리하는 방법을 알려줄 수 있습니다.

변수 선언

변수 선언문의 기본 구문은 다음과 같습니다.

```
var identifierList: type;
```

여기서 *identifierList*는 유효한 식별자의 쉼표로 구분된 목록이고, *type*은 임의의 유효한 타입입니다. 예를 들면, 다음과 같습니다.

```
var I:Integer;
```

정수 타입 변수 *I*를 선언합니다.

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

X 및 *Y* 두 변수를 *실수* 타입으로 선언합니다.

연속적인 변수 선언문은 예약어 **var**를 반복해서 사용하지 않아도 됩니다.

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

```
  Okay:Boolean;
```

프로시저나 함수 내에서 선언된 변수는 *지역 변수*라고 하고, 그 외의 변수들은 *전역 변수*라고 합니다. 전역 변수는 다음 구문을 사용하여 선언과 동시에 초기화할 수 있습니다.

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

여기서 *constantExpression*은 *type* 타입의 값을 나타내는 임의의 상수 표현식입니다. 상수 표현식에 대한 자세한 내용은 5-41 페이지의 "상수 표현식"을 참조하십시오.

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

이 선언문은 다음 선언문 및 문장과 동일합니다.

```
var I:Integer;
 ⋮
I := 7;
```

다중 변수 선언문(`var X, Y, Z:Real;`과 같은)은 초기화를 포함할 수 없으며, 가변 및 파일 타입 변수를 선언할 수 없습니다.

전역 변수를 명시적으로 초기화하지 않으면, 컴파일러는 이를 0으로 초기화합니다. 반면에 지역 변수는 변수의 선언문에서 초기화할 수 없으며, 변수에 값이 할당되기 전까지는 임의의 데이터가 들어 있습니다.

변수를 선언하면 메모리가 할당됩니다. 변수가 더 이상 사용되지 않으면 자동적으로 해제됩니다 특히, 지역 변수는 이들이 선언된 함수나 프로시저로부터 프로그램이 종료될 때까지만 존재합니다. 변수와 메모리 할당에 대한 자세한 내용은 11장 "메모리 관리"를 참조하십시오.

절대 주소

다른 변수와 동일한 주소에 상주하는 새로운 변수를 생성할 수 있습니다. 이렇게 하려면, **absolute** 지시어를 새로운 변수의 선언문 타입 이름 다음에 두어야 하고 앞에는 이전에 선언된 기존 변수의 이름이 있어야 합니다. 예를 들면, 다음과 같습니다.

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

StrLen 변수가 *Str*과 같은 주소에서 시작한다는 것을 지정합니다. 짧은 문자열의 첫 번째 바이트에는 문자열의 길이가 있기 때문에 *StrLen*의 값은 *Str*의 길이입니다.

absolute 선언문에서 변수를 초기화할 수 없으며, 다른 지시어와 **absolute** 선언문을 조합할 수 없습니다.

동적 변수

*GetMem*이나 *New* 프로시저를 호출하여 동적 변수를 생성할 수 있습니다. 이러한 변수는 힙에 할당되며 자동으로 관리되지 않습니다. 변수를 생성한 이후에는 변수의 메모리를 해제하는 것은 궁극적으로 사용자의 책임입니다. *GetMem*으로 생성된 변수는 *FreeMem*으로 해제하고, *New*로 생성된 변수는 *Dispose*로 해제합니다. 동적 변수에서 작동하는 다른 표준 루틴으로는 *ReallocMem*, *Initialize*, *StrAlloc* 및 *StrDispose*가 있습니다.

긴 문자열, 와이드 문자열, 동적 배열, 가변 및 인터페이스 역시 힙이 할당된 동적 변수이지만, 이들 메모리는 자동으로 관리됩니다.

스레드 지역 변수

스레드 지역 변수 또는 *스레드* 변수는 다중 스레드 애플리케이션에서 사용됩니다. 스레드 지역 변수는 전역 변수와 비슷하지만, 각 스레드를 실행하면 변수가 자체 개별적으로 복사되며, 이 사본은 다른 스레드에서 액세스할 수 없습니다. 스레드 지역 변수는 **var** 대신, **threadvar**로 선언됩니다. 예를 들면, 다음과 같습니다.

```
threadvar X:Integer;
```

스레드 변수 선언문은 다음과 같은 제약이 있습니다.

- 하나의 프로시저나 함수 내에 올 수 없습니다.
- 초기화를 포함할 수 없습니다.
- **absolute** 지시어를 명시할 수 없습니다.

포인터 타입 또는 프로시저 타입 스레드 변수를 생성하지 마십시오. 그리고 동적으로 로드 가능한 라이브러리(패키지가 아닌)에서 스레드 변수를 사용하지 마십시오.

컴파일러(긴 문자열, 와이드 문자열, 동적 배열, 가변 및 인터페이스)에 의해 일반적으로 관리되는 동적 변수는 **threadvar**로 선언될 수 있지만, 컴파일러는 각 스레드 실행에 따라 생성된 힙이 할당된 메모리를 자동으로 해제하지 않습니다. 스레드 변수에서 이들 데이터 타입을 사용하는 경우에는 사용자가 이들 메모리를 해제해야 합니다. 예를 들면, 다음과 같습니다.

```
threadvar S:AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
 ⋮
S := '';  // free the memory used by S
```

가변 타입은 *Unassigned*로 설정함으로써 해제할 수 있으며, 인터페이스나 동적 배열은 **nil**로 설정함으로써 해제할 수 있습니다.

선언된 상수

일부 다른 랭귀지의 구문에서는 "상수"라고 합니다. 이들은 17과 같은 숫자 상수(*숫자*라고도 함)이며, 'Hello world!'와 같은 문자열 상수(*문자 문자열* 또는 *문자열 리터럴*이라고도 함)입니다. 숫자 및 문자열 상수에 대한 내용은 4장 "구문 요소"를 참조하십시오. 모든 열거 타입은 해당 타입의 값을 나타내는 상수를 정의합니다. *True*, *False* 및 **nil**과 같은 이미 정의된 상수가 있습니다. 마지막으로 변수와 같이 선언문에 의해 개별적으로 생성된 상수가 있습니다.

선언된 상수는 *true 상수*이거나 *타입이 지정된 상수*입니다. 이 두 종류의 상수는 외관상으로 비슷하지만, 서로 다른 규칙이 적용되며 다른 목적으로 사용됩니다.

True 상수

True 상수는 값이 변경될 수 없는 선언된 식별자입니다. 예를 들면, 다음과 같습니다.

```
const MaxValue = 237;
```

정수 237을 반환하는 *MaxValue*라는 상수를 선언합니다. True 상수 선언 구문은 다음과 같습니다.

```
const identifier = constantExpression
```

여기서 *identifier*는 임의의 유효한 식별자이고 *constantExpression*은 컴파일러가 프로그램을 실행해보지 않고도 계산할 수 있는 표현식입니다. 자세한 내용은 5−41 페이지의 "상수 표현식"을 참조하십시오.

*constantExpression*이 순서값을 반환하는 경우에는 값 타입 변환을 사용하여 선언된 상수의 타입을 지정할 수 있습니다. 예를 들면, 다음과 같습니다.

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

정수 17을 반환하는 *Int64* 타입의 *MyNumber* 상수를 선언합니다. 그외에는 선언된 상수의 타입은 *constantExpression* 타입이 됩니다.

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

표 5.8 정수 상수의 타입

상수 범위 (16진수)	상수 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 선언문의 예제입니다.

```
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 표현식*은 컴파일러가 표현식이 있는 프로그램을 실행해보지 않고도 계산할 수 있는 표현식입니다. 상수 표현식에는 숫자, 문자 문자열, True 상수, 열거 타입 값, 특수 상수 *True*, *False* 및 **nil**과 연산자, 타입 변환 및 집합 생성자가 있는 이들 요소로부터 독점적으로 생성된 표현식이 포함됩니다. 상수 표현식에 다음의 이미 정의된 함수의 호출을 제외한 변수, 포인터 또는 함수 호출을 포함할 수 없습니다.

Abs	*High*	*Low*	*Pred*	*Succ*
Chr	*Length*	*Odd*	*Round*	*Swap*
Hi	*Lo*	*Ord*	*SizeOf*	*Trunc*

*상수 표현식*의 이 정의는 오브젝트 파스칼의 구문 지정의 일부 위치에서 사용됩니다. 상수 표현식은 전역 변수 초기화, 부분범위 타입 정의, 열거 타입의 값에 순서 할당, 기본 매개변수 값 지정, **case** 문 쓰기 및 True 및 타입이 지정된 상수 모두의 선언에 필요합니다.

다음은 상수 표현식의 예제입니다.

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

리소스 문자열

리소스 문자열은 리소스로 저장되며, 프로그램을 다시 컴파일하지 않아도 수정할 수 있도록 실행 파일이나 라이브러리와 연결됩니다. 자세한 내용은 애플리케이션 지역화에 대한 온라인 도움말을 참조하십시오.

const가 **resourcestring**으로 대체된 것을 제외하고 리소스 문자열의 선언은 다른 True 상수와 유사합니다. **=** 기호 오른쪽의 표현식은 상수 표현식이어야 하고 문자열 값을 반환해야 합니다. 예를 들면, 다음과 같습니다.

```
resourcestring
  CreateError = 'Cannot create file %s';        {  for explanations of format
specifiers, }
  OpenError = 'Cannot open file %s';            { see 'Format strings' in the online
Help }
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks\000\000';
  SomeResourceString = SomeTrueConstant;
```

컴파일러는 다른 라이브러리의 리소스 문자열 사이의 이름 충돌을 자동으로 해결합니다.

타입이 지정된 상수

타입이 지정된 상수는 True 상수와는 달리 배열, 레코드, 절차 및 포인터 타입의 값을 가질 수 있습니다. 타입이 지정된 상수는 상수 표현식에서 발생할 수 없습니다.

기본 **{$J-}** 컴파일러 상태에서 타입이 지정된 상수는 새로운 값이 지정될 수 없습니다. 타입이 지정된 상수 사실 읽기 전용 변수입니다. 하지만 **{$J+}** 컴파일러 지시문이 사용되는 경우, 타입이 지정된 상수에 새로운 값을 할당할 수 있습니다. 이들은 초기화된 변수처럼 행동합니다.

다음과 같이 타입이 지정된 상수를 선언합니다.

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

여기서 *identifier*는 임의의 유효한 식별자이고, *type*은 파일 타입과 가변 타입을 제외한 임의의 타입이 될 수 있으며, *value*는 *type* 타입의 표현식입니다. 예를 들면, 다음과 같습니다.

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

대부분의 경우, *value*는 상수 표현식이어야 합니다. 하지만 *type*이 배열, 레코드, 절차 또는 포인터 타입인 경우에는 특수 규칙이 적용됩니다.

배열 상수

배열 상수를 선언하려면, 쉼표로 구분된 배열의 요소 값의 선언 앞뒤에 괄호를 사용합니다. 이 값들은 상수 표현식으로 나타납니다. 예를 들면, 다음과 같습니다.

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

문자 배열을 갖는 *Digits*라는 타입을 가지는 상수를 선언합니다.

인덱스가 0부터 시작하는 문자 배열은 자주 Null 종료 문자열이 나타납니다. 이러한 이유로 문자열 상수는 문자 배열을 초기화하기 위해 사용될 수 있습니다. 그러므로 위의 선언문은 다음과 같이 더 편리하게 나타낼 수 있습니다.

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

다차원 배열 상수를 정의하려면 각 차원의 값에 괄호를 사용하고 이를 쉼표로 구분합니다. 예를 들면, 다음과 같습니다.

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

다음과 같은 *Maze* 배열이 생성됩니다.

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 괄호 내에 *fieldName*:*value*와 같이 세미콜론으로 필드를 지정하여 각 필드의 값을 할당합니다. 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 가변 부분이 있으면 태그 필드에서 선택된 가변만 값이 할당될 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 호환성이 있는 함수 또는 프로시저의 이름을 지정합니다. 예를 들면, 다음과 같습니다.

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

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

이들 선언문의 경우에는 다음과 같이 함수 호출에 *MyFunction* 프로시저 상수를 사용할 수 있습니다.

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

또한 프로시저 상수에 **nil** 값을 할당할 수도 있습니다.

포인터 상수

포인터 상수를 선언할 때는 컴파일 시 최소한 상대 주소로 해석할 수 있는 값으로라도 초기화해야 합니다. 이렇게 할 수 있는 방법으로는 **@** 연산자를 사용하거나, **nil**을 사용

하거나, 상수가 *PChar* 타입인 경우는 문자열 리터럴을 사용하는 방법이 있습니다. 예를 들어, *I*가 *Integer* 타입 전역 변수인 경우에는 다음과 같이 상수를 선언할 수 있습니다.

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

전역 변수는 코드 세그먼트의 부분이기 때문에 컴파일러는 이를 해석할 수 있습니다. 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!';
```

스택 할당된 지역 변수 및 힙 할당된 동적 변수의 주소는 포인터 상수에 할당될 수 없습니다.

6

프로시저 및 함수

*루틴*이라고 일반적으로 불리는 프로시저 및 함수는 독립적인 문장 블록으로, 프로그램의 다른 루틴에서 호출할 수 있습니다. *함수*는 실행 시 값을 반환하는 루틴입니다. *프로시저*는 값을 반환하지 않는 루틴입니다.

함수 호출은 값을 반환하므로 할당문과 연산에서 표현식으로 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

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

*SomeFunction*을 호출하고 결과를 *I*에 할당합니다. 함수 호출은 할당문의 왼쪽에 나타날 수 없습니다.

프로시저 호출과 확장 구문이 활성화되었을 때(**{$X+}**)의 함수 호출은 완전한 문장으로 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
DoSomething;
```

DoSomething 루틴을 호출하여 *DoSomething*이 함수이면 반환값은 버려집니다.

프로시저 및 함수를 재귀적으로 호출할 수 있습니다.

프로시저 및 함수 선언

프로시저 또는 함수를 선언할 때 이름, 매개변수의 수와 타입을 할당하고 함수의 경우에는 반환값의 타입을 할당합니다. 선언의 이 부분을 *프로토타입*, *헤딩*, *헤더*라고 합니다. 그런 다음, 프로시저나 함수 호출 시 실행될 코드 블록을 작성합니다. 선언의 이 부분을 루틴의 *몸체* 또는 *블록*이라고 합니다.

표준 프로시저 *Exit*는 프로시저 또는 함수의 몸체 내에서 사용할 수 있습니다. *Exit*는 루틴의 실행을 정지시킨 다음, 곧바로 루틴을 호출한 곳으로 프로그램 제어를 넘겨줍니다.

프로시저 선언

프로시저 선언의 구조는 다음과 같습니다.

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

여기서, *procedureName*은 유효한 식별자이고 *statements*는 프로시저가 호출될 때 실행하는 일련의 문장이며 (*parameterList*), *directives* 및 *localDeclarations*는 옵션입니다.

- *parameterList*에 대한 자세한 내용은 6-10 페이지의 "매개변수"를 참조하십시오.
- *지시문*에 대한 자세한 내용은 6-5 페이지의 "호출 규칙(Calling conventions)", 6-6 페이지의 "Forward 선언문 및 인터페이스 선언문", 6-6 페이지의 "외부 선언", 6-8 페이지의 "프로시저 및 함수 오버로드" 및 9-3 페이지의 "동적으로 로드할 수 있는 라이브러리 작성"을 참조하십시오. 둘 이상의 지시어를 포함할 경우 세미콜론으로 구분합니다.
- 로컬 식별자를 선언하는 *localDeclarations*에 대한 자세한 내용은 6-9 페이지의 "지역 선언"을 참조하십시오.

다음은 프로시저 선언의 예입니다.

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

이와 같은 선언에 대해 *NumString* 프로시저를 다음과 같이 호출할 수 있습니다.

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

이 프로시저 호출은 값 "17"을 *MyString* (**string** 변수여야함)에 할당합니다.

프로시저의 문 블록에서 해당 프로시저의 *localDeclarations* 부분에서 선언된 변수와 기타 식별자를 사용할 수 있습니다. 위의 예제에 있는 *N*과 *S* 같은 매개변수 이름을 매개변수 목록에 사용할 수도 있습니다. 매개변수 목록은 지역 변수를 정의하므로 *localDeclarations* 섹션에서 매개변수 이름을 재선언하지 마십시오. 마지막으로, 프로시저 선언이 해당하는 유효 범위 내에서 식별자를 사용할 수 있습니다.

함수 선언

함수 선언은 반환 타입과 반환값을 할당하는 것을 제외하고는 프로시저 선언과 같습니다. 함수 선언의 구조는 다음과 같습니다.

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

여기서, *functionName*은 유효한 식별자이고 *returnType*은 타입이며 *statements*는 함수가 호출될 때 실행하는 일련의 문장이며 (*parameterList*), *directives*; 및 *localDeclarations*는 옵션입니다.

- *parameterList*에 대한 자세한 내용은 6-10 페이지의 "매개변수"를 참조하십시오.
- *지시어*에 대한 자세한 내용은 6-5 페이지의 "호출 규칙(Calling conventions)", 6-6 페이지의 "Forward 선언문 및 인터페이스 선언문", 6-6 페이지의 "외부 선언", 6-8 페이지의 "프로시저 및 함수 오버로드" 및 9-3 페이지의 "동적으로 로드할 수 있는 라이브러리 작성"을 참조하십시오. 둘 이상의 지시어를 포함할 경우 세미콜론으로 구분합니다.
- 로컬 식별자를 선언하는 *localDeclarations*에 대한 자세한 내용은 6-9 페이지의 "지역 선언"을 참조하십시오.

함수의 문장 블록에는 프로시저에 적용되는 것과 같은 규칙이 적용됩니다. 문장 블록에서 함수의 *localDeclarations* 부분에서 선언된 변수와 기타 식별자, 매개변수 목록의 매개변수 이름, 함수 선언이 속하는 유효 범위(scope) 내의 식별자를 사용할 수 있습니다. 또한 함수 이름은 이미 정의된 변수 *Result*이기 때문에 함수의 반환값이 있는 특별한 변수처럼 동작합니다.

예를 들면, 다음과 같습니다.

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

매개변수가 없고 항상 정수값 17을 반환하는 *WF*라는 상수 함수를 정의합니다. 이 선언은 다음과 동일합니다.

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

다음과 같이 더 복잡한 함수 선언도 있습니다.

```
function Max(A: array of Real; N:Integer):Real;
var
  X:Real;
  I:Integer;
begin
  X := A[0];
```

```
    for I := 1 to -1 do
      if X < A[I] then X := A[I];
    Max := X;
  end;
```

값의 타입이 선언된 반환 타입과 일치하기만 하면, 문장 블록 내의 *Result*나 함수 이름에 값을 반복적으로 할당할 수 있습니다. 함수의 실행이 종료되면 *Result*나 함수 이름에 마지막으로 할당된 값이 함수의 반환값이 됩니다. 예를 들면, 다음과 같습니다.

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

Result 및 함수 이름은 항상 같은 값을 나타냅니다. 따라서

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

이 구문은 값 11을 반환합니다. 그러나 *Result* 모두를 함수 이름으로 바꿀 수 없습니다. 함수 이름이 할당문의 왼쪽에 나타나면 컴파일러는 함수 이름이 *Result* 같이 반환값을 추적하는 데 사용된다고 이해합니다. 그러나 함수 이름이 문장 블록의 아무 곳에나 나타나면 컴파일러는 함수 이름을 이 함수의 재귀 호출로 해석합니다. 반면에 *Result*는 연산, 타입 변환, 생성자 설정, 인덱스, 다른 루틴 호출 등에서 변수로 사용할 수 있습니다.

확장 구문이 활성화된 상태에서(**{$X+}**), *Result*는 암시적으로 모든 함수에서 선언됩니다. 재선언하지 마십시오.

*Result*나 함수 이름에 값이 할당되지 않고 실행이 종료되면 함수의 반환값은 정의되지 않습니다.

호출 규칙(Calling conventions)

프로시저 또는 함수를 선언할 때 지시어 **register**, **pascal**, **cdecl**, **stdcall**, **safecall** 가운데 하나를 사용하여 *호출 규칙*을 지정할 수 있습니다. 예를 들면, 다음과 같습니다.

```
function MyFunction(X, Y:Real}: Real; cdecl;
  ⋮
```

호출 규칙은 매개변수를 루틴으로 전달하는 순서를 결정합니다. 또한 호출 규칙은 스택에서 매개변수 제거, 전달한 매개변수에 대한 레지스터 사용, 오류 및 예외 처리 등에 영향을 줍니다. 기본 호출 규칙은 **register**입니다.

- **register** 및 **pascal** 규칙은 매개변수를 왼쪽에서 오른쪽으로 전달합니다. 즉, 가장 왼쪽에 있는 매개변수 값을 계산하여 처음으로 전달하고 가장 오른쪽에 있는 매개변수 값을 계산하여 마지막으로 전달합니다. **cdecl**, **stdcall** 및 **safecall** 규칙은 매개변수를 오른쪽에서 왼쪽으로 전달합니다.
- **cdecl**을 제외한 모든 규칙에서 프로시저 또는 함수는 반환할 때 스택에서 매개변수를 제거합니다. **cdecl** 규칙에서 호출자는 호출이 반환될 때 스택에서 매개변수를 제거합니다.
- 다른 규칙들은 스택에 있는 모든 매개변수를 전달하는 반면, **register** 규칙은 CPU 레지스터를 최대 3개까지 사용하여 매개변수를 전달합니다.
- **safecall** 규칙은 "방화벽" 예외를 구현합니다. Windows에서 이 규칙은 프로세스간 COM 오류를 공지합니다.

아래의 표는 호출 규칙을 요약한 것입니다.

표 6.1 호출 규칙

지시어	매개변수 순서	클린업	레지스터에 매개변수 전달 여부
register	왼쪽에서 오른쪽	루틴	예
pascal	왼쪽에서 오른쪽	루틴	아니오
cdecl	오른쪽에서 왼쪽	호출자	아니오
stdcall	오른쪽에서 왼쪽	루틴	아니오
safecall	오른쪽에서 왼쪽	루틴	아니오

기본 **register** 규칙은 보통 스택 프레임을 생성하지 않기 때문에 가장 효율적입니다. (published 속성에 대한 액세스 메소드는 반드시 **register**를 사용해야 합니다.) **cdecl** 규칙은 C 또는 C++ 랭귀지로 개발된 공유 라이브러리에서 함수를 호출할 때 유용하고 **stdcall** 및 **safecall** 규칙은 일반적으로 외부 코드를 호출할 때 사용하는 것이 좋습니다. Windows에서 운영체제 API는 **stdcall** 및 **safecall** 규칙을 사용합니다. 기타 운영 체제는 일반적으로 **cdecl** 규칙을 사용합니다. **stdcall** 규칙이 **cdecl** 규칙보다 효율적입니다.

safecall 규칙은 이중 인터페이스 메소드 (10장 "객체 인터페이스" 참조)를 선언하는 데 사용해야 합니다. 역 호환성을 위해 **pascal** 규칙은 유지됩니다. 호출 규칙에 대한 자세한 내용은 12장 "프로그램 제어"를 참조하십시오.

지시어 **near**, **far** 및 **export**는 16비트 Windows 프로그래밍에서의 호출을 위해 사용됩니다. 32비트 애플리케이션에는 영향을 미치지 않고 역 호환성을 위해서만 유지됩니다.

Forward 선언문 및 인터페이스 선언문

forward 지시문은 프로시저 또는 함수 선언에서 지역 변수 선언과 문장을 비롯하여 블록을 교체합니다. 예를 들면, 다음과 같습니다.

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

*Calculate*라는 함수를 선언합니다. **foreward** 선언문 다음에서 루틴은 블록이 들어 있는 *정의 선언*에서 재선언되어야 합니다. *Calculate*에 대한 정의 선언은 다음과 같습니다.

```
function Calculate;
    { declarations }
begin
    { statement block }
end;
```

원래 정의 선언은 루틴 매개변수 목록이나 반환 타입을 반복하지 않아도 됩니다. 그러나 정의 선언이 루틴 매개변수 목록이나 반환 타입을 반복하지 않으면 이들은 기본 매개변수를 생략할 수 있다는 것만 제외하고는 **forward** 선언문에 있는 것과 정확하게 일치해야 합니다. **forward** 선언문이 오버로드된 프로시저 또는 함수(6-8 페이지의 "프로시저 및 함수 오버로드" 참조)를 지정하면 정의 선언은 반드시 매개변수 목록을 반복해야 합니다.

forward 선언문과 그 정의 선언 사이에는 선언문만올 수 있습니다. 정의 선언은 **external** 또는 **assembler** 선언이 될 수 있으나 다른 **forward** 선언문은 될 수 없습니다.

forward 선언문은 프로시저 또는 함수 식별자의 유효 범위를 소스 코드의 이전까지 확장하기 위한 것입니다. 이렇게 하면 **forward** 선언문 루틴이 실제로 정의되기 전에 다른 프로시저 및 함수에서 이를 호출할 수 있습니다. 코드를 더 유연하게 만드는 것 외에 **forward** 선언문은 상호 재귀(mutual recursion)를 위해서도 필요합니다.

forward 지시어는 유닛의 **interface**섹션에서는 사용할 수 없습니다. 그러나 **interface** 섹션의 프로시저 및 함수 헤더는 **forward** 선언문처럼 동작하므로 **implementation** 섹션에 정의 선언이 있어야 합니다. **interface** 섹션에서 선언된 루틴은 유닛의 어느 곳에서나 사용할 수 있으며 다른 유닛이나 루틴이 선언된 유닛을 사용하는 프로그램에서도 사용할 수 있습니다.

외부 선언

프로시저 또는 함수 선언의 블록을 교체하는 **external** 지시어를 사용하면 프로그램과 별도로 컴파일된 루틴을 호출할 수 있습니다. 객체 파일이나 동적으로 로드할 수 있는 라이브러리에서 외부 루틴를 호출할 수 있습니다.

매개변수의 수가 가변적인 C++ 함수를 가져오려면 **varargs** 지시어를 사용하십시오. 예를 들면, 다음과 같습니다.

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

varargs 지시어는 외부 루틴과 **cdecl** 호출 규칙에만 작동합니다.

객체 파일과 연결

별도의 컴파일된 객체 파일에서 루틴을 호출하려면 먼저 **$L** 또는 **$LINK** 컴파일러 지시어를 사용하여 객체 파일을 애플리케이션과 연결하십시오. 예를 들면, 다음과 같습니다.

```
Windows인 경우: {$L BLOCK.OBJ}
Linux인 경우:   {$L block.o}
```

BLOCK.OBJ (Windows) 또는 block.o (Linux)를 프로그램이나 유닛으로 연결합니다. 그런 다음, 호출하려는 함수와 프로시저를 선언하십시오.

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

이제 BLOCK.OBJ (Windows) 또는 block.o (Linux) 에서 *MoveWord* 및 *FillWord* 루틴을 호출할 수 있습니다.

위와 같은 선언은 종종 어셈블리어로 작성된 외부 루틴에 액세스하는 데 사용합니다. 또는 어셈블리어 루틴을 오브젝트 파스칼 소스 코드에 직접 사용할 수 있습니다. 자세한 내용은 13장 "인라인 어셈블러 코드"를 참조하십시오.

라이브러리에서 함수 가져오기

동적으로 로드할 수 있는 라이브러리 (.so 또는 .DLL) 에서 루틴을 가져오려면 다음과 같은 형식의 지시어를

```
external stringConstant;
```

일반 프로시저 또는 함수 헤더의 끝에 붙입니다. 작은 따옴표로 둘러싼 *stringConstant*는 라이브러리 파일의 이름입니다. 예를 들어, Windows인 경우

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

strlib.dll에서 *SomeFunction*이라는 함수를 가져옵니다.

Linux인 경우,

```
function SomeFunction(S:string}: string; external 'strlib.so';
```

strlib.so에서 *SomeFunction*이라는 함수를 가져옵니다.

라이브러리에 있는 루틴 이름이 아닌 다른 이름으로 루틴을 가져올 수 있습니다. 이렇게 하려면 **external** 지시어에서 원래 이름을 할당하십시오.

```
external stringConstant1 name stringConstant2;
```

여기서, 첫 번째 *stringConstant*는 라이브러리 파일의 이름이고 두 번째 *stringConstant*는 루틴의 원래 이름입니다.

Windows인 경우: 예를 들어, 다음 선언은 Windows API의 일부인 user32.dll 에서 함수를 가져옵니다.

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

함수의 원래 이름은 *MessageBoxA*이지만, *MessageBox*라는 이름으로 가져옵니다.

이름 대신 번호를 사용하여 가져오려는 루틴을 식별할 수 있습니다.

```
external stringConstant index integerConstant;
```

여기서, *integerConstant*는 export 테이블에 있는 루틴의 인덱스입니다.

Linux인 경우: 예를 들어, 다음 선언은 libc.so.6에서 표준 시스템 함수를 import합니다.

```
function OpenFile(const PathName:PChar; Flags:Integer): Integer; cdecl;
  external 'libc.so.6' name 'open';
```

함수의 원래 이름은 *open*이지만, *OpenFile*이라는 이름으로 가져옵니다.

가져오기 선언에서 루틴의 이름의 철자나 대/소문자가 정확한지 확인하십시오. 나중에 가져온 루틴을 호출할 때 이름의 대/소문자를 구분합니다.

라이브러리에 대한 자세한 내용은 9장 "라이브러리 및 패키지"를 참조하십시오.

프로시저 및 함수 오버로드

같은 유효 범위 내에서 하나 이상의 루틴을 같은 이름으로 선언할 수 있습니다. 이를 *오버로드*라고 합니다. 오버로드된 루틴은 **overload** 지시어로 선언해야 하여 매개변수 목록으로 구분합니다. 예를 들어, 다음 선언을 생각해 보십시오.

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

function Divide(X, Y:Integer}: Integer; overload;
begin
  Result := X div Y;
end;
```

이 선언에서 *Divide*라는 함수를 두 개 만듭니다. 두 함수는 매개변수 타입이 다릅니다. *Divide*를 호출하면 컴파일러는 호출에서 전달한 실제 매개변수를 따져보고 둘 중 어떤 함수를 호출할 지 결정합니다. 예를 들어, Divide(6.0, 3.0)는 해당 인수가 실수이기 때문에 첫 번째 *Divide* 함수를 호출합니다.

루틴 선언의 매개변수와는 타입이 다른 오버로드된 루틴 매개변수로 전달할 수 있습니다. 그러나 둘 이상의 선언에서 이 매개변수 타입은 루틴 선언에 할당할 수 있는 타입이어야 합니다. 이는 루틴이 다른 정수 타입이나 실수 타입으로 오버로드될 때 가장 많이 발생합니다.

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

이 경우, 확실하게 호환이되면, 컴파일러는 호출 시 넘겨진 매개변수를 포함하면서 타입 범위가 가장 작은 매개변수가 있는 루틴을 호출합니다. 실수 타입 상수 표현식은 항상 *확장(Extended)* 타입임을 염두에 두십시오.

오버로드된 루틴은 매개변수의 수 또는 매개변수의 타입으로 구별할 수 있어야 합니다. 따라서 다음 선언 한 쌍은 컴파일 오류를 일으킵니다.

```
function Cap(S:string}: string; overload;
 ⋮
ccc Cap(var Str: string}; overload;
 ⋮
```

그러나 다음과 같은 선언은 적합합니다.

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

오버로드된 선언이 **forward** 선언문 또는 인터페이스 선언에서 선언되면 정의 선언은 루틴의 매개변수 목록을 반복해야 합니다.

오버로드된 루틴에서 기본 매개변수를 사용한다면 모호한 매개변수 시그너처에 주의하십시오. 자세한 내용은 6-17 페이지의 "기본 매개변수 및 오버로드된 루틴"을 참조하십시오.

호출할 때 루틴의 이름을 한정하여 오버로드의 잠재적인 효과를 제한할 수 있습니다. 예를 들어, `Unit1.MyProcedure(X, Y)`는 *Unit1*에서 선언된 루틴만 호출할 수 있습니다. 호출의 이름 및 매개변수 목록과 일치하는 루틴이 *Unit1*에 없으면 오류가 발생합니다.

클래스 계층 구조에서 오버로드된 메소드 구별에 대한 자세한 내용은 7-12 페이지의 "메소드 오버로드"를 참조하십시오. 공유 라이브러리에서 오버로드된 루틴 내보내기에 대한 자세한 내용은 9-5 페이지의 "exports 절"을 참조하십시오.

지역 선언

함수 또는 프로시저의 몸체는 종종 루틴의 문장 블록에서 사용하는 지역 변수의 선언으로 시작합니다. 이러한 선언은 상수, 타입 및 기타 루틴도 포함합니다. 로컬 식별자의 유효 범위는 식별자가 선언된 루틴으로 제한됩니다.

중첩 루틴

함수 및 프로시저는 때때로 블록의 지역 선언 섹션에 기타 함수와 프로시저를 포함할 수 있습니다. 예를 들어, 다음 *DoSomething*이라는 프로시저 선언에는 중첩된 프로시저가 있습니다.

```
procedure DoSomething(S:string);
var
  X, Y:Integer;

  procedure NestedProc(S:string);
  begin
    ⋮
  end;

begin
   ⋮
  NestedProc(S);
   ⋮
end;
```

중첩 루틴의 유효 범위는 루틴이 선언된 프로시저 또는 함수로 제한됩니다. 예제에서 *NestedProc*은 *DoSomething* 내에서만 호출할 수 있습니다.

중첩 루틴의 실제 예제를 보려면 *DateTimeToString* 프로시저, *ScanDate* 함수 및 *SysUtils* 유닛에 있는 루틴을 살펴보십시오.

매개변수

대부분의 프로시저 및 함수 헤더에는 *매개변수 목록*이 있습니다. 예를 들어, 헤더에는 다음이 있습니다.

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

매개변수 목록은 (X:Real; Y: Integer}입니다.

매개변수 목록은 세미콜론으로 구분하고 괄호로 묶은 일련의 매개변수입니다. 각 선언은 매개변수 이름을 콤마로 구분하며, 대부분의 선언 다음에는 콜론과 타입 식별자가 오며, = 기호와 기본값이 오는 경우도 있습니다. 매개변수 이름은 반드시 유효한 식별자여야 합니다. 모든 선언 앞에는 예약어 **var**, **const**, **out** 가운데 하나가 옵니다. 예를 들면, 다음과 같습니다.

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

매개변수 목록은 호출되면 루틴에 전달해야 하는 매개변수 번호, 순서 및 타입을 할당합니다. 매개변수가 없는 루틴이라면 선언에서 식별자 목록과 괄호를 생략하십시오.

```
procedure UpdateRecords;
begin
 ⋮
end;
```

프로시저 또는 함수 몸체 내에서, 매개변수 이름(위 첫 번째 예제의 *X* 및 *Y*)은 지역 변수로 사용할 수 있습니다. 프로시저 또는 함수 몸체의 지역 선언 섹션에서 매개변수 이름을 재선언하지 마십시오.

매개변수 의미론

매개변수는 다음과 같은 방법으로 분류됩니다.

- 모든 매개변수는 *값*, *변수*, *상수* 또는 *출력*으로 분류합니다. 값 매개변수가 기본이며 예약어 **var**, **const** 및 **out**은 각각 변수, 상수 및 출력 매개변수를 나타냅니다.
- 값 매개변수는 항상 *타입이 할당*되어 있으며, 상수, 변수 및 출력 매개변수는 타입을 가지거나 또는 *타입을 가지지 않습니다*.
- 배열 매개변수에는 특수한 규칙이 적용됩니다. 6-14 페이지의 "배열 매개변수"를 참조하십시오.

파일과 파일이 포함된 구조 타입 인스턴스는 변수(**var**) 매개변수로만 전달할 수 있습니다.

값 및 변수 매개변수

대부분의 매개변수는 값 매개변수(기본값) 또는 변수(**var**) 매개변수입니다. 값 매개변수는 *값에 의해 전달*되는 반면, 변수 매개변수는 *참조에 의해 전달*됩니다. 예를 들어 다음과 같은 함수를 생각해 보십시오.

```
function DoubleByValue(X:Integer}: Integer;     // X is a value parameter
begin
  X := X * 2;
  Result := X;
end;

function DoubleByRef(var X:Integer}: Integer;   // X is a variable parameter
begin
  X := X * 2;
  Result := X;
end;
```

이 함수는 동일한 결과를 반환하지만, 두 번째 함수인 *DoubleByRef*만 매개변수로 전달된 변수의 값을 변경할 수 있습니다. 다음 함수를 호출한다고 가정해 보십시오.

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

코드 실행이 끝난 다음, *DoubleByValue*로 전달된 변수 *I*는 처음에 할당한 값과 같은 값을 가집니다. 그러나 *DoubleByRef*로 전달된 변수*V*는 다른 값을 가지게 됩니다.

값 매개변수는 프로시저 또는 함수 호출에서 전달된 값으로 초기화된 지역 변수처럼 동작합니다. 변수를 값 매개변수로 전달하면 프로시저 또는 함수는 값을 복사하여 값 복사본을 만듭니다. 이 복사본을 변경해도 원래 변수에는 아무런 영향을 주지 않으며 프로그램 실행이 호출자로 반환되면 변경 내용을 잃어버립니다.

반면, 변수 매개변수는 복사본이 아니라 포인터처럼 동작합니다. 프로그램 실행이 호출자로 반환되고 매개변수 이름이 유효 범위를 벗어난 후에 함수 또는 프로시저의 몸체 내에 있는 매개변수를 변경하면 그 변경 내용은 계속 유지됩니다.

같은 변수를 둘 이상의 **var** 매개변수로 전달해도, 복사본은 만들어지지 않습니다. 이는 다음 예제에서 설명합니다.

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

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

이 코드가 실행된 다음, *I*의 값은 3이 됩니다.

루틴 선언이 **var** 매개변수를 할당하면 루틴을 호출할 때 할당할 수 있는 표현식, 즉 변수, 타입이 할당된 상수(**{$J+}** 상태에서), 역참조(dereference) 포인터 또는 인덱스된 변수를 루틴으로 전달해야 합니다. 이전 예제에서 `DoubleByValue(7)`는 적합하지만, `DoubleByRef(7)`는 오류를 일으킵니다.

`DoubleByRef(MyArray[I])` 같이 **var** 매개변수에 전달된 인덱스 및 포인터 역참조의 값은 루틴 실행 전에 한 번 계산됩니다.

상수 매개변수

상수(**const**) 매개변수는 지역 상수 또는 읽기 전용 변수와 유사합니다. 프로시저 또는 함수의 몸체 내에서 상수 매개변수에 값을 할당할 수 없다는 것과 다른 루틴에 값을 **var** 매개변수로 전달할 수 없다는 것만 제외하면 상수 매개변수는 값 매개변수와 유사합니다. 그러나 객체 참조를 상수 매개변수로 전달하면 객체 속성을 수정할 수 있습니다.

const를 사용하면 컴파일러는 구조 타입 매개변수와 문자열 타입 매개변수에 대한 코드를 최적화할 수 있습니다. 또한 의도하지 않게 참조에 의해 매개변수를 다른 루틴으로 전달하는 것을 막아줍니다.

다음은 *SysUtils* 유닛의 *CompareStr* 함수에 대한 헤더입니다.

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

S1 및 *S2*는 *CompareStr*의 몸체에서 수정되지 않기 때문에 상수 매개변수로 선언될 수 있습니다.

출력 매개변수

out 매개변수는 변수 매개변수 같이 참조에 의해 전달됩니다. 그러나 **out** 매개변수가 있는 참조 변수의 초기 값은 루틴으로 전달할 때 루틴에 의해 버려집니다. **out** 매개변수는 출력용으로만 사용합니다. 즉, 함수 또는 프로시저에게 어디에 출력을 저장할 지 알려주지만, 입력에 대한 내용을 제공하지는 않습니다.

예를 들어, 다음과 같은 프로시저 헤더를 생각해 보십시오.

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

*GetInfo*를 호출하려면 반드시 *SomeRecordType* 타입 변수를 전달해야 합니다.

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

그러나 *GetInfo* 프로시저로 데이터를 전달하는 데 *MyRecord*를 사용하지 않습니다. *MyRecord*는 단지 컨테이너로, *GetInfo*가 생성하는 정보를 저장합니다. *GetInfo*를 호출하면 프로그램 제어가 프로시저로 전달하기 전에 *MyRecord*가 사용한 메모리를 해제합니다.

Out 매개변수는 종종 COM 및 CORBA 같은 분산 객체 모델과 함께 사용됩니다. 초기화되지 않은 변수를 함수 또는 프로시저로 전달할 때에도 **out** 매개변수를 사용해야 합니다.

타입이 할당되지 않은 매개변수

var, **const** 및 **out** 매개변수를 선언할 때 타입 선언을 생략할 수 있습니다. 그러나, 값 매개변수는 반드시 타입이 있어야 합니다. 예를 들면, 다음과 같습니다.

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

모든 타입의 매개변수를 사용하는 *TakeAnything*라는 프로시저를 선언합니다. 이 루틴을 호출할 때 숫자 또는 타입이 할당되지 않은 숫자 상수를 전달할 수 없습니다.

프로시저 또는 함수 몸체 내에서 타입이 할당되지 않은 매개변수는 어떤 타입과도 호환되지 않습니다. 타입이 할당되지 않은 매개변수를 연산하려면 타입 변환해야 합니다. 일반적으로, 컴파일러는 타입이 할당되지 않은 매개변수의 연산이 유효한지 확인할 수 없습니다.

다음 예에서는 특정 바이트 수의 두 변수를 비교하는 *Equal*이라는 함수에서 타입이 할당되지 않은 매개변수를 사용합니다.

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

다음과 같이 선언했다고 가정해 보십시오.

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

*Equal*에 대해 다음을 호출할 수 있습니다.

```
Equal(Vec1, Vec2, SizeOf(TVector))           // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N)       // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5) // compare first 5 to last 5 elements of Vec1
Equal(Vec1[1], P, 4)                         // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

문자열 매개변수

짧은 문자열 매개변수를 갖는 루틴을 선언할 경우 매개변수 선언에 길이 할당자를 포함할 수 없습니다. 즉, 다음 선언은

```
procedure Check(S: string[20]};   // syntax error
```

컴파일 오류를 일으킵니다. 그러나

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

이 선언은 유효합니다. 특수 식별자 *OpenString*은 다양한 길이의 짧은 문자열 매개변수를 갖는 루틴을 선언하는 데 사용할 수 있습니다.

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

{$H−}** 및 **{$P+} 컴파일러 지시문을 모두 사용하면 예약어 **string**은 매개변수 선언에서 *OpenString*과 같습니다.

짧은 문자열, *OpenString*, **$H** 및 **$P**는 역 호환성을 위해서만 지원됩니다. 새로운 코드에서는 긴 문자열을 사용하여 이러한 문제를 피할 수 있습니다.

배열 매개변수

배열 매개변수가 있는 루틴을 선언할 때 매개변수 선언에 인덱스 타입 할당자를 포함할 수 없습니다. 즉, 다음 선언은

```
procedure Sort(A: array[1..10] of Integer};   // syntax error
```

컴파일 오류를 일으킵니다.그러나

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

이 선언은 유효합니다. 그러나 대부분의 경우에 *개방 타입 배열 매개변수*를 사용하는 것이 더 낫습니다.

개방형 (open) 배열 매개변수

개방형 배열 매개변수를 사용하면 크기가 다른 배열을 프로시저 또는 함수로 전달할 수 있습니다. 개방형 배열 매개변수를 사용하는 루틴을 정의하려면 매개변수 선언에서 array[*X..Y*] of *type*이 아닌 array of *type*을 사용하십시오. 예를 들면, 다음과 같습니다.

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

크기에 상관없는 문자 배열을 사용하고 정수를 반환하는 *Find*라는 함수를 선언합니다.

참고 개방형 배열 매개변수의 구문은 동적 배열 타입의 구문과 유사하지만 똑같지는 않습니다. 위의 예제는 동적 배열을 비롯한 *Char* 요소의 배열을 갖는 함수를 만듭니다. 동적 배열인 매개변수를 선언하려면 타입 식별자를 할당해야 합니다.

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

동적 배열에 대한 자세한 내용은 5−19 페이지의 "동적 배열"을 참조하십시오.

루틴 몸체와 함께 개방형 배열 매개변수에는 다음 규칙이 적용됩니다.

- 개방형 배열 매개변수는 항상 인덱스가 0부터 시작합니다. 첫 번째 요소는 0이고 두 번째 요소는 1입니다. 표준 *Low* 및 *High* 함수는 각각 0과 *길이*−1을 반환합니다. *SizeOf* 함수는 루틴으로 전달한 실제 배열 크기를 반환합니다.
- 요소를 사용해야만 개방형 배열 매개변수에 액세스할 수 있습니다. 전체 개방 타입 배열 매개변수에 대해 할당할 수 없습니다.
- 개방형 배열 매개변수 또는 타입이 할당되지 않은 **var** 매개변수로만 다른 프로시저 및 함수로 전달할 수 있습니다. *SetLength*로 전달할 수 없습니다.
- 배열 대신 개방형 배열 매개변수의 기본 타입 변수를 전달할 수 있습니다. 길이 1의 배열로 간주될 것입니다.

개방형 배열 값 매개변수로 배열을 전달하면 컴파일러는 루틴의 스택 프레임에서 배열의 로컬 복사본을 만듭니다. 큰 배열을 전달하여 스택이 오버플로되지 않도록 하십시오.

다음 예에서는 실수 배열에 있는 각 요소에 0을 할당하는 *Clear* 프로시저와 실수 배열에 있는 요소의 합을 계산하는 *Sum* 함수를 정의하는 데 개방형 배열 매개변수를 사용합니다.

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

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

개방형 배열 매개변수를 사용하는 루틴을 호출할 때 *개방형 배열 생성자*를 개방 타입 배열 매개변수에 전달할 수 있습니다. 6−18 페이지의 "개방형 배열 생성자"를 참조하십시오.

가변 개방형 배열 매개변수

가변 개방형 배열 매개변수를 사용하면 타입이 다른 표현식 배열을 단일 프로시저나 함수에 전달할 수 있습니다. 가변 개방형 배열 매개변수가 있는 루틴을 정의하려면 매개변수의 타입으로 **array of const**를 할당하십시오. 따라서

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

이질적 배열에서 연산을 수행할 수 있는 *DoSomething*이라는 프로시저를 선언합니다.

array of const 생성자는 **array of** *TVarRec*와 같습니다. *시스템* 유닛에서 선언된 *TVarRec*는 정수, 부울, 문자, 실수, 문자열, 포인터, 클래스, 클래스 참조, 인터페이스,

가변 타입 등의 값을 가질 수 있는 가변 부분이 있는 레코드를 나타냅니다. *TVarRec*의 *VType* 필드는 배열에 있는 각 요소의 타입을 나타냅니다. 일부 타입은 값이 아니라 포인터로 넘겨집니다. 특히 긴 문자열은 *포인터*로 전달되고 **string**으로 타입 변환되어야 합니다. 자세한 내용은 *TVarRec*의 온라인 도움말을 참조하십시오.

다음 예제에서는 함수로 전달된 각 요소의 문자열 표현을 만들고 그 결과를 단일 문자열로 연결시키는 함수에서 가변 개방 타입 배열 매개변수를 사용합니다. 이 함수로 호출한 문자열 처리 루틴은 *SysUtils*에서 정의합니다.

```
function MakeStr(const Args: array of const):string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I:Integer;
begin
  Result := '';
  for I := 0 to 66 do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
        vtInt64:      Result := Result + IntToStr(VInt64^);
    end;
end;
```

개방 타입 배열 생성자(6-18 페이지의 "개방형 배열 생성자" 참조)를 사용하여 이 함수를 호출할 수 있습니다. 예를 들면, 다음과 같습니다.

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

"test100 T3.14159TForm"이라는 문자열을 반환합니다.

기본 매개변수

프로시저 또는 함수 헤더에서 기본 매개변수 값을 지정할 수 있습니다. 기본값은 타입이 할당된 **const** 및 값 매개변수에 대해서만 사용할 수 있습니다. 기본값을 할당하려면 매개변수 선언의 끝에 '= 상수 표현식' 형식으로 사용하십시오. 이 매개변수 타입은 상수 표현식에 할당할 수 있는 타입이어야 합니다.

예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

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

이것은 다음 프로시저를 호출하는 것과 똑같습니다.

```
FillArray(MyArray);
```

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

여러 매개변수 선언의 기본값을 한 번에 할당할 수 없습니다. 따라서

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

이것은 적합합니다. 그러나,

```
function MyFunction(X, Y:Real = 3.5}: Real;   // syntax error
```

이것은 적합하지 않습니다.

기본값이 있는 매개변수는 반드시 매개변수 목록의 끝에 나와야 합니다. 즉, 처음 선언된 기본값 다음에 오는 모든 매개변수에는 또한 기본값이 있어야 합니다. 따라서 다음 선언은 오류가 있습니다.

```
procedure MyProcedure(I:Integer = 1; S: string};   // syntax error
```

프로시저 타입에서 지정된 기본값은 실제 루틴에서 할당된 기본값을 오버라이드합니다. 따라서, 다음과 같이 선언했다고 가정해 보십시오.

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

다음과 같은 문장에서

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

결과적으로 (*N*, 1.0) 값이 *Resizer*로 전달됩니다.

기본 매개변수는 상수 표현식에 의해 지정할 수 있는 값으로 제한됩니다.(5−41 페이지의 "상수 표현식" 참조) 따라서 동적 배열 타입, 프로시저 타입, 클래스 타입, 클래스 참조 타입, 인터페이스 타입 등의 매개변수는 기본값으로 **nil** 이외의 값은 가질 수 없습니다. 레코드 타입, 가변 타입, 파일 타입, 정적 배열 타입, 객체 타입 등의 매개변수는 기본값을 절대 가질 수 없습니다.

기본 매개변수 값이 있는 루틴 호출에 대한 자세한 내용은 6−18 페이지의 "프로시저 및 함수 호출"을 참조하십시오.

기본 매개변수 및 오버로드된 루틴

오버로드된 루틴에서 기본 매개변수 값을 사용하려면 모호한 매개변수 시그너처를 피하십시오. 예를 들어, 다음 소스를 보면

```
procedure Confused(I: Integer}; overload;
 f
procedure Confused(I: Integer; J: Integer = 0}; overload;
 f
Confused(X);    //  Which procedure is called?
```

사실상 어느 프로시저도 호출되지 않았습니다. 이 코드는 컴파일 오류를 일으킵니다.

forward 선언문과 인터페이스 선언문의 기본 매개변수

forward 선언문이 루틴에 있거나 루틴이 유닛의 인터페이스 섹션에 나타나면 기본 매개변수 값은 **forward** 선언문이나 인터페이스 선언문에서 지정해야 합니다. 이런 경우에 기본값은 정의(구현) 선언에서 생략될 수 있습니다. 그러나 정의 선언에 기본값이 있으면 해당 기본값은 **forward** 선언문이나 인터페이스 선언문에 있는 기본값과 정확하게 일치해야 합니다.

프로시저 및 함수 호출

프로시저 또는 함수를 호출할 때 프로그램 제어는 호출이 발생한 지점에서 루틴의 몸체로 넘깁니다. 한정자를 사용하거나 사용하지 않고 루틴의 선언 이름이나 루틴을 가리키는 절차 변수를 사용하여 호출할 수 있습니다. 두 경우 모두 루틴이 매개변수와 함께 선언되면 호출 시 선언한 매개변수 목록 순서와 타입으로 이 루틴의 매개변수 목록에 해당하는 매개변수를 전달해야 합니다. 루틴으로 전달된 매개변수는 *실제 매개변수(actual parameter)*라고 하고 루틴의 선언에 있는 매개변수는 *형식 매개변수(formal parameter)*라고 합니다.

루틴을 호출할 때는 다음 내용에 유의하십시오.

- 타입이 있는 **const** 및 값 매개변수를 전달하는 데, 해당 형식 매개변수(formal parameter)의 타입은 값 매개변수에 할당할 수 있는 타입이어야 합니다.
- **var** 및 **out** 매개변수를 전달하는데 사용하는 표현식은 타입이 할당되지 않은 형식매개변수(formal parameter) 외에는 해당 타입 매개변수와 동일한 타입을 가져야 합니다.
- **var** 및 **out** 매개변수를 전달하는데에는 할당할 수 있는 표현식만 사용할 수 있습니다.
- 루틴의 형식 매개변수에 타입이 없으면 분수 값이 있는 숫자 및 실수 상수를 실제 매개변수로 사용할 수 없습니다.

기본 매개변수 값을 사용하는 루틴을 호출하면 처음 승인한 기본값 다음에 오는 모든 실제 매개변수는 또한 기본값을 사용해야 합니다. 다음과 같은 구조의 `SomeFunction(,,X)` 호출은 적합하지 않습니다.

모든 기본 매개변수를 루틴으로 전달할 경우 괄호를 생략할 수 있습니다. 예를 들어, 프로시저가 다음과 같을 경우,

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

다음 호출과 동일합니다.

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

개방형 배열 생성자

개방형 배열 생성자를 사용하면 함수 및 프로시저 호출 내에서 배열을 직접 생성할 수 있습니다. 개방 타입 배열은 개방형 배열 매개변수 또는 가변 개방형 배열 매개변수로만 전달할 수 있습니다.

개방 타입 배열 생성자는 집합 생성자처럼 콤마로 구분하고 괄호로 묶은 일련의 표현식입니다. 예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

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

Add 프로시저를 다음과 같이 호출할 수 있습니다.

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

이것은 다음과 동일합니다.

```
var Temp:array[0.0.3] of Integer;
 ⋮
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

개방 타입 배열 생성자는 값 또는 **const** 매개변수로만 전달할 수 있습니다. 배열 매개변수의 기본 타입은 생성자의 표현식에 할당 호환 가능한 타입이어야 합니다. 가변 개방 타입 배열 매개변수의 경우 표현식은 다른 타입일 수 있습니다.

7

클래스 및 객체

클래스 또는 *클래스 타입*은 *필드*, *메소드* 및 *속성*으로 구성된 구조를 정의합니다. 클래스 타입의 인스턴스를 *객체*라고 합니다. 클래스의 필드, 메소드 및 속성을 클래스의 *컴포넌트* 또는 *멤버*라고 합니다.

- 필드는 본래 객체의 일부인 변수입니다. 레코드 필드와 마찬가지로 클래스 필드는 클래스의 각 인스턴스에 있는 데이터 항목을 나타냅니다.
- 메소드는 클래스와 연결된 프로시저나 함수입니다. 대부분의 메소드는 클래스의 인스턴스인 객체에서 작동합니다. *클래스 메소드*라고 하는 일부 메소드는 클래스 타입 자체에서 작동합니다.
- 속성은 객체(종종 필드에 저장됨)와 연결된 데이터에 대한 인터페이스입니다. 속성에는 데이터를 읽고 수정하는 방법을 결정하는 *액세스 지정자*가 있습니다. 객체 외부에 있는 프로그램의 다른 부분에서 속성은 대부분 필드처럼 나타납니다.

객체는 해당 클래스 타입에 의해 결정되는 동적으로 할당된 메모리 블록입니다. 각 객체에는 클래스에서 정의된 모든 필드의 고유한 복사본이 있지만 클래스의 모든 인스턴스는 동일한 메소드를 공유합니다. 객체는 *생성자*와 *소멸자*라는 특정 메소드에 의해 생성되고 소멸됩니다.

클래스 타입의 변수는 객체를 참조하는 포인터입니다. 따라서 동일한 객체를 두 개 이상의 변수로 참조할 수 있습니다. 다른 포인터와 마찬가지로 클래스 타입 변수는 **nil** 값을 가질 수 있습니다. 하지만 가리키는 객체에 액세스하기 위해 클래스 타입 변수를 명시적으로 역참조할 필요는 없습니다. 예를 들어, `SomeObject.Size := 100`은 *SomeObject*가 참조하는 객체의 *Size* 속성에 100이라는 값을 지정합니다. 그러나 이것을 `SomeObject^.Size := 100` 타입으로 쓰지는 *않습니다*.

클래스 타입

클래스 타입은 선언하고 이름을 지정한 후에야 인스턴스화할 수 있습니다. 클래스 타입을 변수 선언에서 정의할 수 없습니다. 프로시저나 함수 선언이 아닌 프로그램이나 유닛의 가장 외부 유효 범위(scope)에서만 클래스를 선언하십시오.

클래스 타입 선언은 다음과 같은 형태를 가집니다.

```
type className = class (ancestorClass)
  memberList
 end;
```

여기서 *className*은 유효한 식별자이고, (*ancestorClass*)는 옵션이며, *memberList*는 필드, 메소드, 속성 같은 클래스의 멤버를 선언합니다. (*ancestorClass*)를 생략하면 새 클래스는 이미 정의된 *TObject* 클래스에서 직접 상속됩니다. (*ancestorClass*)가 있고 *memberList*가 비어 있는 경우 end를 생략할 수 있습니다. 클래스 타입 선언에 클래스에 의해 구현된 *인터페이스*의 목록을 포함할 수도 있습니다. 10-4 페이지의 "인터페이스 구현"을 참조하십시오.

메소드는 몸체 없이 함수나 프로시저 헤더로 클래스 선언에 나타납니다. 각 메소드에 대한 선언은 프로그램 내의 어디에서나 정의할 수 있습니다.

예를 들어, *클래시스* 유닛에서 *TMemoryStream* 클래스 선언은 다음과 같습니다.

```
type
TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity:Longint;
    procedure SetCapacity(NewCapacity:Longint);
  protected
    function Realloc(var NewCapacity:Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count:Longint): Longint; override;
  end;
```

*TMemoryStream*은 *클래시스* 유닛의 *TStream*의 자손으로 대부분의 해당 멤버를 상속하지만, 소멸자 메소드인 *Destroy*를 비롯한 몇몇 메소드 및 속성을 재정의하거나 새로 정의합니다. 생성자인 *Create*는 변경되지 않고 *TObject*에서 상속하므로 재선언되지 않습니다. 각 멤버는 *private*, *protected* 또는 *public*으로 선언되어 있습니다. 이 클래스에 *published* 멤버는 없습니다. 이 용어에 대한 설명은 7-4 페이지의 "클래스 멤버의 가시성"을 참조하십시오.

위와 같이 선언했다면, 다음과 같이 *TMemoryStream*의 인스턴스를 만들 수 있습니다.

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

상속 및 유효 범위(scope)

클래스를 선언할 때 해당 직계 조상을 지정할 수 있습니다. 예를 들면, 다음과 같습니다.

```
type TSomeControl = class(TControl);
```

이 구문은 *TControl*의 자손인 *TSomeControl*이라는 클래스를 선언합니다. 클래스 타입은 해당 직계 조상으로부터 모든 멤버를 자동으로 상속받습니다. 각 클래스는 새 멤버를 선언할 수 있고 상속된 멤버를 재정의할 수 있지만 조상에서 정의된 멤버를 제거할 수는 없습니다. 따라서 *TSomeControl*에는 *TControl*과 *TControl*의 각 조상에서 정의된 멤버가 모두 들어 있습니다.

멤버의 식별자 유효 범위(scope)는 멤버가 선언된 위치에서 시작하여 클래스 선언 끝까지 계속되며 클래스의 모든 자손과 클래스 및 클래스 자손에서 정의된 모든 메소드 블록까지 확장됩니다.

TObject 및 TClass

시스템 유닛에서 선언된 *TObject* 클래스는 다른 모든 클래스의 최고 조상입니다. *TObject*는 기본 생성자 및 소멸자를 비롯한 최소한의 메소드만을 정의합니다. *TObject*와 함께 *시스템* 유닛은 클래스 참조 타입인 *TClass*를 선언합니다.

```
TClass = class of TObject;
```

*TObject*에 대한 자세한 내용은 온라인 도움말을 참조하십시오. 클래스 참조 타입에 대한 자세한 내용은 7-23 페이지의 "클래스 참조"를 참조하십시오.

클래스 타입 선언에서 조상을 지정하지 않으면 클래스는 *TObject*를 직접 상속받습니다. 따라서

```
type TMyClass = class
 f
end;
```

위 선언은 아래의 선언과 같습니다.

```
type TMyClass = class(TObject)
 f
end;
```

가독성을 위해 후자 형태를 사용하는 것이 좋습니다.

클래스 타입의 호환성

클래스 타입의 조상은 클래스 타입에서 지정할 수 있는 타입입니다. 따라서 클래스 타입의 변수는 모든 자손 타입의 인스턴스를 참조할 수 있습니다. 예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure)
  TSquare = class(TRectangle);
var
  Fig:TFigure;
```

Fig 변수는 *TFigure*, *TRectangle* 및 *TSquare* 타입의 값으로 지정될 수 있습니다.

객체 타입

클래스 타입을 대체하기 위해 다음 구문을 사용하여 *객체 타입*을 선언할 수 있습니다.

```
type objectTypeName = object (ancestorObjectType)
 memberList
end;
```

여기서 *objectTypeName*은 유효한 식별자이고, (*ancestorObjectType*)은 옵션이며, *memberList*는 필드, 메소드 및 속성을 선언합니다. (*ancestorObjectType*)을 생략하면 새 타입에 상속되는 조상이 없습니다. 객체 타입은 published 멤버를 가질 수 없습니다.

객체 타입은 *TObject*의 자손이 아니므로 기본 제공 생성자, 소멸자 또는 기타 메소드를 제공하지 않습니다. *New* 프로시저를 사용하여 객체 타입의 인스턴스를 만들 수 있고 만든 인스턴스를 *Dispose* 프로시저로 소멸시키거나, 레코드에서처럼 객체 타입 변수를 선언할 수 있습니다.

객체 타입은 역 호환성을 위해서만 지원되므로 사용하지 않는 것이 좋습니다.

클래스 멤버의 가시성

모든 클래스 멤버에는 **private**, **protected**, **public**, **published** 또는 **automated** 예약어 중 하나로 *가시성*이라는 속성(attribute)을 나타낼 수 있습니다. 예를 들면, 다음과 같습니다.

```
published property Color:TColor read GetColor write SetColor;
```

이 구문은 *Color*라는 published 속성을 선언합니다. 가시성은 최소한으로 액세스할 수 있는 private, 중간 수준으로 액세스할 수 있는 protected, 최대한으로 액세스할 수 있는 public, published, automated를 사용하여 멤버가 액세스할 수 있는 범위 및 방법을 결정합니다.

멤버의 선언이 해당 가시성 지정자 없이 나타나는 경우 멤버는 바로 앞 멤버와 동일한 가시성을 가집니다. 클래스가 {$M+} 상태에서 컴파일되거나 {$M+} 상태에서 컴파일된 클래스로부터 파생되는 경우 클래스 선언의 맨 처음에 나타나고 지정된 가시성이 없는 멤버는 기본값 published를 갖게 되고, 그외는 public을 갖게 됩니다.

가독성을 위해 private 멤버를 모아 두고, protected 멤버도 모아 두고 나머지도 속성별로 모아 두는 것이 좋습니다. 그러면 각 가시성 예약어는 한 번만 나타나며 선언의 새 "섹션" 시작을 표시합니다. 일반적인 클래스 선언은 다음과 같이 나타납니다.

```
type
  TMyClass = class(TControl)
  private
   : { private declarations here}
  protected
   : { protected declarations here }
  public
   : { public declarations here }
  published
   : { published declarations here }
  end;
```

자손 클래스를 재선언하면 자손 클래스 멤버의 가시성을 높일 수는 있지만 가시성을 낮출 수는 없습니다. 예를 들어, protected 속성은 자손 클래스에서 public이 될 수 있지만 private이 될 수는 없습니다. 또한, published 멤버는 자손 클래스에서 public이 될 수 없습니다. 자세한 내용은 7-21 페이지의 "속성 오버라이드 및 재선언"을 참조하십시오.

Private, protected 및 public 멤버

private 멤버는 해당 클래스가 선언된 유닛이나 프로그램 외부에서 볼 수 없습니다. 즉, 다른 모듈에서 private 메소드를 호출할 수 없고 private 필드나 속성을 다른 모듈에서 읽거나 쓸 수 없습니다. 동일한 모듈에 관련 클래스 선언을 두어 private 멤버에 대한 액세스 가능성을 더 크게 만들지 않고도 private 멤버 간에 서로 액세스할 수 있도록 할 수 있습니다.

protected 멤버는 자손 클래스가 선언된 모듈에 관계 없이 해당 클래스가 선언된 모듈과 모든 자손 클래스에서 볼 수 있습니다. 즉, protected 메소드를 호출할 수 있고 protected 멤버가 선언된 클래스에서 물려받은 클래스에 속한 모든 메소드 정의문에서 protected 필드나 속성을 읽거나 쓸 수 있습니다. 파생 클래스의 구현에만 사용할 수 있는 멤버는 일반적으로 protected로 선언합니다.

public 멤버는 해당 클래스를 참조할 수 있는 모든 곳에서 볼 수 있습니다.

Published 멤버

Published 멤버는 public 멤버와 동일한 가시성을 가집니다. 차이점으로는 published 멤버에서 *런타임 타입 정보*(RTTI)가 생성된다는 것입니다. RTTI를 사용하여 응용 프로그램에서 객체의 필드 및 속성을 동적으로 쿼리하고 해당 메소드를 찾을 수 있습니다. 또한 RTTI를 사용하여 폼 파일을 저장하고 로드할 때 속성 값에 액세스할 수 있고 Object Inspector에서 속성을 표시하며 *이벤트 핸들러*라는 특정 메소드를 *이벤트*라는 특정 속성과 연결할 수 있습니다.

Published 속성은 특정 데이터 타입에 제한됩니다. 순서, 문자열, 클래스, 인터페이스 및 메소드 포인터 타입은 published로 선언될 수 있습니다. 그러므로 집합 타입은 0과 31 사이의 순서값을 가집니다. 즉, 집합 타입은 바이트, 워드 또는 더블 워드여야 합니다. *Real48* 이외의 실수 타입은 published로 선언될 수 있습니다. 배열 타입의 속성(아래에서 설명하는 *배열 속성*과 다름)은 published로 선언될 수 없습니다.

일부 속성은 published로 선언할 수 있더라도 스트리밍 시스템이 완벽하게 지원하지는 않습니다. 이러한 속성에는 레코드 타입, published로 선언할 수 있는 모든 타입의 배열 속성(7-19 페이지의 "배열 속성" 참조), 익명 값을 포함하는 열거 타입(5-7 페이지의 "순서가 명시적으로 할당된 열거 타입" 참조)의 속성이 들어 있습니다. 이러한 종류의 속성을 published로 선언하는 경우 Object Inspector에서 해당 속성을 제대로 표시할 수 없거나 객체를 디스크에 스트림할 때 속성 값이 유지되지 않습니다.

모든 메소드는 published가 될 수 있지만 클래스는 오버로드된 두 개 이상의 메소드를 같은 이름의 published로 만들 수 없습니다. 필드는 클래스나 인터페이스 타입인 경우에만 published가 될 수 있습니다.

클래스는 **{$M+}** 상태에서 컴파일되거나 **{$M+}** 상태에서 컴파일된 클래스의 자손이 아니라면 published 멤버를 가질 수 없습니다. published 멤버가 있는 대부분의 클래스는 **{$M+}** 상태에서 컴파일된 *TPersistent*에서 파생되므로 **$M** 지시어를 사용할 필요가 없습니다.

Automated 멤버

Automated 멤버는 public 멤버와 동일한 가시성을 가집니다. 다른 점이라면 automated 멤버의 경우 Automation 서버에 필요한 *Automation 타입 정보*가 생성된다는 것입니다. Automated 멤버는 일반적으로 Windows 클래스에서만 나타나고 Linux 프로그래밍에는 권장되지 않습니다. **automated** 예약어는 역 호환성을 위해 유지됩니다. *ComObj* 유닛의 *TAutoObject* 클래스는 **automated**를 사용하지 않습니다.

automated로 선언된 메소드와 속성에는 다음과 같은 제한 사항이 따릅니다.

- 모든 속성, 배열 속성 매개변수, 메소드 매개변수 및 함수 결과의 타입은 automated 가능해야 합니다. automated 가능 타입은 *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool* 및 모든 인터페이스 타입입니다.
- 메소드 선언에서는 기본 **register** 호출 규칙을 사용해야 합니다. 메소드 선언은 가상(virtual)일 수 있지만 동적(dynamic)일 수는 없습니다.
- 속성 선언은 액세스 지정자(**read** 및 **write**)를 포함할 수 있지만 다른 지정자(**index**, **stored**, **default** 및 **nodefault**)는 포함할 수 없습니다. 액세스 지정자는 기본 **register** 호출 규칙을 사용하는 메소드 식별자를 열거해야 하며 필드 식별자는 허용되지 않습니다.
- 속성 선언에서는 타입을 지정해야 합니다. 속성 오버라이드는 허용되지 않습니다.

automated 메소드나 속성의 선언은 **dispid** 지시어를 포함할 수 있습니다. **dispid** 지시어에 이미 사용된 ID를 지정하면 오류가 발생합니다.

Windows에서 dispid 지시어는 멤버에 대한 Automation 디스패치 ID를 지정하는 정수 상수의 앞에 있어야 합니다. 그렇지 않으면 컴파일러는 클래스 및 해당 조상의 메소드나 속성으로 사용된 최대 디스패치 ID보다 큰 디스패치 ID를 멤버에게 지정합니다. Automation (Windows만 해당)에 대한 자세한 내용은 10-10 페이지의 "Automation 객체(Windows만 해당)"를 참조하십시오.

Forward 선언 및 상호 종속 클래스

클래스 타입 선언이 **class**와 세미콜론으로 끝나는 경우 즉,

```
type className = class;
```

이 구문처럼 **class** 다음에 조상이나 클래스 멤버가 없으므로 *forward 선언*이라고 합니다. Forward 선언은 동일한 타입 선언 섹션 내에서 동일한 클래스의 *선언을 정의*함으로써 해결되어야 합니다. 즉, forward 선언과 forward 선언 정의 사이는 다른 타입 선언만 올 수 있습니다.

Forward 선언은 상호 종속 클래스를 허용합니다. 예를 들면, 다음과 같습니다.

```
type
  TFigure = class;  // forward declaration
  TDrawing = class
    Figure:TFigure;
     ⋮
  end;

  TFigure = class  // defining declaration
    Drawing: TDrawing;
     ⋮
  end;
```

클래스 멤버를 선언하지 않은 *TObject*에서 파생한 타입의 완전한 선언과 forward 선언을 혼동하지 마십시오.

```
type
  TFirstClass = class;           // this is a forward declaration

  TSecondClass = class           // this is a complete class declaration
    end;

  TThirdClass = class(TObject);  // this is a complete class declaration
```

필드

필드는 객체에 속하는 변수와 같습니다. 필드는 클래스 타입을 비롯한 어떤 타입도 될 수 있습니다. 즉, 필드는 객체 참조를 가질 수 있습니다. 필드는 일반적으로 private입니다.

클래스의 필드 멤버를 정의하려면 변수를 선언하는 것처럼 필드를 선언합니다. 모든 필드는 속성이나 메소드 선언 이전에 선언해야 합니다. 예를 들어, 다음 선언으로 *TObject*에서 상속 받는 메소드가 아닌 *TNumber*라는 클래스를 만듭니다. 이 클래스의 유일한 멤버는 *Int*라는 정수 필드입니다.

```
type TNumber = class
  Int:Integer;
end;
```

필드는 정적으로 바인드됩니다. 즉, 이 필드에 대한 참조는 컴파일 시에 만들어집니다. 이것이 의미하는 내용을 알려면 다음 코드를 참조하십시오.

```
type
  TAncestor = class
    Value:Integer;
  end;

  TDescendant = class(TAncestor)
    Value: string;  // hides the inherited Value field
  end;

var
  MyObject:TAncestor;

begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!';  // error
  TDescendant(MyObject).Value := 'Hello!';  // works!
end;
```

*MyObject*가 *TDescendant*의 인스턴스를 가지고 있지만 *TAncestor*로 선언됩니다. 그러므로 컴파일러는 *MyObject.Value*를 *TAncestor*에 선언된 정수 필드에 대해 참조하는 것처럼 해석합니다. 하지만 두 필드 모두 *TDescendant* 객체에 있으며 상속된 *Value*는 새로운 값에 의해 숨겨져 타입 변환을 통해 액세스할 수 있습니다.

메소드

메소드는 클래스와 연결된 프로시저나 함수입니다. 메소드에 대한 호출은 메소드가 작동하는 객체를 지정합니다. 클래스 메소드의 경우에는 클래스를 지정합니다. 예를 들면, 다음과 같습니다.

```
SomeObject.Free
```

*SomeObject*의 *Free* 메소드를 호출합니다.

메소드 선언과 구현

클래스 선언 내에서 메소드는 프로시저 및 함수 헤더로 나타나며 **forward** 선언처럼 동작합니다. 클래스 선언 다음의 동일한 모듈 내에서 각 메소드는 선언을 정의함으로써 구현되어야 합니다. 예를 들어, *TMyClass*의 선언이 *DoSomething*이라는 메소드를 포함한다고 가정해 보십시오.

```
type
  TMyClass = class(TObject)
     ⋮
    procedure DoSomething;
     ⋮
  end;
```

*DoSomething*의 선언 정의는 모듈 뒷 부분에 나타나야 합니다.

```
procedure TMyClass.DoSomething;
begin
 ⋮
end;
```

클래스는 유닛의 인터페이스나 구현 섹션에서 선언될 수 있지만 클래스 메소드의 선언 정의는 구현 섹션에 있어야 합니다.

선언 정의의 헤더에서 메소드 이름은 항상 해당 메소드가 속한 클래스 이름으로 한정됩니다. 헤더는 클래스 선언의 매개변수 목록을 반복할 수 있으며, 매개변수 목록을 반복하는 경우 매개변수의 순서, 타입 및 이름이 정확하게 일치해야 합니다. 메소드가 함수인 경우에도 반드시 반환값이 있어야 합니다.

메소드 선언은 다른 함수나 프로시저와 함께 사용되지 않는 특정 지시어를 포함할 수 없습니다. 지시어는 선언 정의가 아닌 클래스 선언에만 나타나야 하며 다음과 같은 순서가 되어야 합니다.

```
reintroduce; overload; binding; calling convention; abstract; warning
```

여기서 *binding*은 **virtual**, **dynamic** 또는 **override**이고, *calling convention*은 **register**, **pascal**, **cdecl**, **stdcall** 또는 **safecall**이며, *warning*는 **platform**, **deprecated** 또는 **library**입니다.

Inherited

예약어 **inherited**는 다형의 동작을 구현하는 데 특별한 역할을 수행합니다. 다형의 동작은 그 뒤에 오는 식별자 유무에 관계 없이 메소드 정의문에서 발생할 수 있습니다.

inherited 다음에 메소드 멤버의 이름이 나오면, 이것은 일반적인 메소드 호출 또는 속성이나 필드의 참조를 나타냅니다. 한 가지 다른 점은 이 메소드를 찾을 때, 메소드가 속한 클래스의 바로 위 조상 클래스부터 찾기 시작한다는 점입니다. 예를 들면, 다음과 같습니다.

```
inherited Create(...);
```

이것은 메소드 정의에 사용하며 상속된 *Create*를 호출합니다.

inherited 다음에 식별자가 나타나지 않으면 이것은 메소드와 동일한 이름을 가지는 상속된 메소드를 참조합니다. 이러한 경우, **inherited**는 명시적인 매개변수는 없지만, 해당 메소드에 전달된 매개변수와 동일한 매개변수를 상속된 메소드로 전달합니다. 예를 들면, 다음과 같습니다.

```
inherited;
```

이 구문은 생성자의 구현에서 자주 나타납니다. 자손에게 전달한 매개변수와 동일한 매개변수로 상속된 생성자를 호출합니다.

Self

메소드의 구현에서 *Self* 식별자는 메소드를 호출한 객체를 참조합니다. 예를 들어, *클래시스* 유닛에서 *TCollection*의 *Add* 메소드 구현은 다음과 같습니다.

```
function TCollection.Add: TCollectionItem;
begin
  Result := FItemClass.Create(Self);
end;
```

Add 메소드는 항상 *TCollectionItem* 자손인 *FItemClass* 필드가 참조하는 클래스에서 *Create* 메소드를 호출합니다. *TCollectionItem.Create*는 *TCollection* 타입의 단일 매개변수를 가지므로 *Add*는 *Add*가 호출된 *TCollection* 인스턴스에 매개변수를 전달합니다. 이것은 다음 코드에서 설명합니다.

```
var MyCollection:TCollection
 ⋮
MyCollection.Add  // MyCollection is passed to the TCollectionItem.Create method
```

*Self*는 여러가지 이유로 유용합니다. 예를 들어, 클래스 타입에서 선언된 멤버 식별자는 클래스 메소드 중 하나의 블록에서 재선언될 수 있습니다. 이러한 경우, 원래 멤버 식별자를 `Self.`*Identifier*로 액세스할 수 있습니다.

클래스 메소드의 *Self*에 대한 내용은 7-25 페이지의 "클래스 메소드"를 참조하십시오.

메소드 바인딩

메소드는 *정적 (static)*, *가상 (virtual)* 또는 *동적 (dynamic)*일 수 있습니다. 기본값은 static입니다. 가상 메소드와 동적 메소드는 *오버라이드*될 수 있고 *추상화*될 수 있습니

다. 이는 한 클래스 타입의 변수가 자손 클래스 타입의 값을 가질 때 결정됩니다. 이에 따라 메소드가 호출될 때 어떤 구현이 활성화되는지 결정됩니다.

정적 메소드

메소드는 기본적으로 정적입니다. 정적 메소드가 호출될 때, 메소드 호출에 사용된 클래스나 객체 변수의 컴파일 시 선언된 타입에 따라 활성화할 구현이 결정됩니다. 다음 예제에서 *Draw*는 정적 메소드입니다.

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

위와 같이 선언했다면, 다음 코드는 정적 메소드 호출의 효과를 보여줍니다. *Figure.Draw*에 대한 두 번째 호출에서 *Figure* 변수는 *TRectangle* 클래스의 객체를 참조하지만 *Figure* 변수의 선언된 타입이 *TFigure*이기 때문에 *TFigure*의 *Draw*의 구현이 호출됩니다.

```
var
  Figure:TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw;  // calls TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;  // calls TFigure.Draw
  TRectangle(Figure).Draw;  // calls TRectangle.Draw
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;  // calls TRectangle.Draw
  Rectangle.Destroy;
end;
```

가상 메소드 및 동적 메소드

메소드를 가상(virtual) 또는 동적(dynamic)으로 만들려면 해당 선언에 **virtual**이나 **dynamic** 지시어를 포함하면 됩니다. 정적 메소드와는 달리 가상 메소드와 동적 메소드는 자손 클래스에서 *오버라이드*될 수 있습니다. 오버라이드된 메소드가 호출될 때 변수의 선언된 타입이 아닌 메소드 호출에 사용된 클래스나 객체의 실제(런타임) 타입에 따라 활성화할 구현이 결정됩니다.

메소드를 오버라이드하려면 **override** 지시어를 사용하여 해당 메소드를 재선언합니다. **override** 선언은 조상 선언과 순서, 매개변수 타입 및 결과 타입(있는 경우)이 일치해야 합니다.

다음 예제에서 *TFigure*에서 선언된 *Draw* 메소드는 두 개의 자손 클래스에서 오버라이드합니다.

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

위와 같이 선언했다면, 다음 코드는 실제 타입이 런타임 시 변하는 변수를 통해 가상 메소드 호출의 효과를 보여줍니다.

```
var
  Figure:TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;  // calls TRectangle.Draw
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;  // calls TEllipse.Draw
  Figure.Destroy;
end;
```

가상 메소드와 동적 메소드만이 오버라이드됩니다. 그러나 모든 메소드를 *오버로드*할 수 있습니다. "메소드 오버로드"를 참조하십시오.

가상 메소드와 동적 메소드 비교

가상 메소드와 동적 메소드는 유사합니다. 가상 메소드와 동적 메소드는 런타임 시 메소드 호출 디스패칭의 구현에서만 서로 다릅니다. 가상 메소드는 속도를 최적화하는 반면, 동적 메소드는 코드 크기를 최적화합니다.

일반적으로, 가상 메소드는 다형적인 동작을 구현하기 위한 가장 효율적인 방법입니다. 기초 클래스가 응용 프로그램에서 여러 자손 클래스가 상속되면서 경우에 따라서 오버라이드되는 오버라이드가 가능한 메소드들을 많이 선언할 때 동적 메소드가 유용합니다.

오버라이드 및 숨기기

메소드 선언에서 동일한 메소드 식별자와 매개변수 시그너처를 상속된 메소드로 지정하면서 **override**를 포함하지 않을 경우, 새로운 선언은 오버라이드하지 않고 상속된 메소드만을 *숨깁니다*. 두 메소드는 모두 메소드 이름이 정적으로 바운딩된 자손 클래스에 있습니다. 예를 들면, 다음과 같습니다.

```
type
  T1 = class(TObject)
    procedure Act; virtual;
  end;
  T2 = class(T1)
    procedure Act;  // Act is redeclared, but not overridden
  end;

var
```

```
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act;  // calls T1.Act
end;
```

Reintroduce

reintroduce 지시어는 이전에 선언된 가상 메소드 숨기기에 대한 컴파일러 경고를 표시하지 않습니다. 예를 들면, 다음과 같습니다.

```
procedure DoSomething; reintroduce;  // the ancestor class also has a DoSomething
method
```

새로운 메소드를 사용하여 상속된 가상 메소드를 숨길 때 reintroduce를 사용합니다.

추상 메소드

추상 메소드는 선언된 클래스에서 구현된 부분이 없는 가상 메소드나 동적 메소드입니다. 추상 메소드는 자손 클래스에서 구현합니다. 추상 메소드는 virtual 메소드 또는 dynamic 메소드 다음에 abstract 지시어를 사용하여 선언해야 합니다. 예를 들면, 다음과 같습니다.

```
procedure DoSomething; virtual; abstract;
```

메소드가 오버라이드된 클래스나 클래스의 인스턴스에서만 추상 메소드를 호출할 수 있습니다.

메소드 오버로드

overload 지시어를 사용하여 메소드를 재선언할 수 있습니다. 이러한 경우, 재선언된 메소드가 조상과 다른 매개변수 시그너처를 가지는 경우 재선언된 메소드를 숨기지 않고 상속된 메소드를 오버로드합니다. 자손 클래스의 메소드를 호출하면 호출한 루틴의 매개변수와 일치하는 구현이 활성화됩니다.

가상 메소드를 오버로드하는 경우 자손 클래스에서 가상 메소드를 재선언할 때 reintroduce 지시어를 사용합니다. 예를 들면, 다음과 같습니다.

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;
  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
   ⋮
  SomeObject := T2.Create;
  SomeObject.Test('Hello!');  // calls T2.Test
  SomeObject.Test(7);         // calls T1.Test
```

클래스 내에서 이름이 동일한 여러 개의 오버로드된 메소드를 published로 사용할 수 없습니다. 런타임 타입 정보를 유지하려면 각 published 멤버에 대한 고유 이름이 필요합니다.

```
type
  TSomeClass = class
  published
    function Func(P:Integer):Integer;
    function Func(P: Boolean): Integer  // error
     ⋮
```

read 또는 write 속성 지정자 역할을 수행하는 메소드를 오버로드할 수 없습니다.

오버로드된 메소드의 구현은 클래스 선언에서 매개변수 목록을 반복해야 합니다.오버로드에 대한 자세한 내용은 6-8 페이지의 "프로시저 및 함수 오버로드"를 참조하십시오.

생성자

생성자는 인스턴스 객체를 만들고 초기화하는 특수한 메소드입니다. 생성자 선언은 프로시저 선언처럼 보이지만 constructor로 시작합니다. 예를 들면, 다음과 같습니다.

```
constructor Create;
constructor Create(AOwner:TComponent};
```

생성자는 기본 register 호출 규칙을 사용해야 합니다. 선언에서 반환 값을 지정하지 않아도 생성자는 자신이 생성하는 객체에 대한 참조를 반환합니다.

클래스는 생성자를 두 개 이상 가질 수 있지만 대부분의 클래스는 하나만 있습니다. 일반적으로 *Create* 생성자를 호출하는 것이 일반적입니다.

객체를 만들려면 클래스 타입에서 생성자 메소드를 호출합니다. 예를 들면, 다음과 같습니다.

```
MyObject := TMyClass.Create;
```

이 구문은 힙(heap)에 새 객체를 위한 저장소를 할당하고, 모든 순서 필드 값을 영(0)으로 설정하며, 모든 포인터와 클래스 타입 필드에 nil을 지정하고, 모든 문자열 필드를 비어있게 합니다. 생성자 구현에 지정된 다른 동작은 다음에 실행되며, 일반적으로 객체는 생성자에 전달된 매개변수로 초기화됩니다. 마지막으로 생성자는 새로 할당되고 초기화된 객체에 대한 참조를 반환합니다. 반환 값 타입은 생성자 호출에서 지정된 클래스 타입과 동일합니다.

클래스 참조에서 호출된 생성자 실행 중 예외가 발생하는 경우 *Destroy* 소멸자는 정상적으로 생성되지 못한 객체를 소멸하기 위해 자동으로 호출됩니다.

클래스 참조가 아닌 객체 참조를 사용하여 생성자를 호출할 때 생성자는 객체를 만들지 않습니다. 그 대신, 생성자는 생성자 구현에 있는 문장만을 실행한 다음 객체에 대한 참조를 반환합니다. 생성자는 일반적으로 예약어 inherited와 함께 객체 참조에서 호출되어 상속된 생성자를 실행합니다.

클래스 타입과 클래스 타입 생성자에 대한 예제는 다음과 같습니다.

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender:TObject);
    procedure BrushChanged(Sender:TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;

constructor TShape.Create(Owner:TComponent);
begin
  inherited Create(Owner);  // Initialize inherited parts
  Width := 65;  // Change inherited properties
  Height := 65;
  FPen := TPen.Create;  // Initialize new fields
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

생성자의 첫 번째 동작은 일반적으로 상속된 생성자를 호출하여 객체의 상속된 필드를 초기화하는 것입니다. 그런 다음 생성자는 자손 클래스의 필드를 초기화합니다. 생성자는 항상 새로운 객체에 할당된 저장소를 지우기 때문에 모든 필드 값은 처음에 영(순서 타입), **nil**(포인터 타입 및 클래스 타입), 비어 있음(문자열 타입) 또는 *할당되지 않음*(가변 타입)이 됩니다. 따라서 영(0)이 아니거나 비어 있지 않은 값이 아니라면 생성자의 구현에서 필드를 초기화할 필요가 없습니다.

클래스 타입 식별자를 통해 호출된 경우 **virtual**로 선언된 생성자는 정적 생성자와 동일합니다. 그러나 클래스 참조 타입과 함께 사용하면 가상 생성자는 타입이 컴파일 시에 알려지지 않는 객체 구문인 다형의 객체 구문을 허용합니다. 7-23 페이지의 "클래스 참조"를 참조하십시오.

소멸자

소멸자는 호출한 객체의 메모리를 해제하는 특수한 메소드입니다. 소멸자 선언은 프로시저 선언처럼 보이지만 **destructor**로 시작합니다. 예를 들면, 다음과 같습니다.

```
destructor Destroy;
destructor Destroy; override;
```

소멸자는 기본 **register** 호출 규칙을 사용해야 합니다. 클래스는 소멸자를 두 개 이상 가질 수 있지만 각 클래스는 상속된 *Destroy* 메소드를 오버라이드하고 다른 소멸자는 선언하지 않는 것이 좋습니다.

소멸자를 호출하려면 인스턴스 객체를 참조해야 합니다. 예를 들면, 다음과 같습니다.

```
MyObject.Destroy;
```

소멸자가 호출되면 소멸자 구현에서 지정된 동작이 처음으로 실행됩니다. 일반적으로 이러한 동작은 포함된 객체를 소멸하고 객체에 의해 할당된 리소스를 해제하는 작업으로 구성됩니다. 소멸자가 호출되면 객체에 할당된 저장소가 없어집니다.

소멸자 구현에 대한 예제는 다음과 같습니다.

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

소멸자 구현의 마지막 동작은 일반적으로 상속된 소멸자를 호출하여 객체의 상속된 필드를 소멸하는 것입니다.

객체를 생성하는 중 예외가 발생하는 경우 *Destroy*가 자동으로 호출되어 정상적으로 생성되지 않은 객체를 제거합니다. 이는 불완전하게 생성된 객체를 제거하는 코드가 *Destroy*에 구현되어 있어야 한다는 것을 의미합니다. 다른 동작을 실행하기 전에 생성자는 새 객체의 필드를 영(0) 또는 빈 값으로 설정하기 때문에 불완전하게 생성된 객체의 클래스 타입 및 포인터 타입 필드는 항상 **nil**입니다. 그러므로 소멸자는 클래스 타입 또는 포인터 타입 필드에서 작동하기 전에 **nil** 값을 확인해야 합니다. *Destroy*가 아닌 *TObject*에서 정의된 *Free* 메소드를 호출하면 객체를 소멸하기 전에 **nil** 값을 쉽게 확인할 수 있습니다.

메시지 메소드

메시지 메소드는 동적으로 디스패치된 메시지에 대한 응답을 구현합니다. 메시지 메소드 구문은 모든 플랫폼에서 지원됩니다. VCL은 메시지 메소드를 사용하여 Windows 메시지에 응답할 수 있습니다. CLX는 메시지 메소드를 사용하여 시스템 이벤트에 응답할 수 없습니다.

메시지 메소드는 메소드 선언에서 **message** 지시어를 포함하여 만들어지며 메시지 ID를 지정하는 1과 49151 사이의 정수 상수 앞에 옵니다. VCL 컨트롤의 메시지 메소드에서 정수 상수는 해당 레코드 타입과 함께 *Messages* 유닛에서 정의된 Windows 메시지 ID 중 하나일 수 있습니다. 메시지 메소드는 단일 **var** 매개변수를 갖는 프로시저여야 합니다.

예를 들면, Windows에서는 다음과 같습니다.

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ⋮
  end;
```

예를 들어, Linux나 크로스 플랫폼 프로그래밍에서 다음과 같이 메시지를 처리할 수 있습니다.

```
const
  ID_REFRESH = $0001;
type
  TTextBox = class(TCustomControl)
  private
    procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
    ⋮
  end;
```

메시지 메소드는 상속된 메시지 메소드를 오버라이드하기 위해 override 지시어를 포함할 필요가 없습니다. 실제로 메시지 메소드는 동일한 메소드 이름이나 매개변수 타입을 해당 메시지가 오버라이드하는 메소드로 지정할 필요가 없습니다. 단지 메시지 ID에 따라 메소드가 응답할 메시지 및 메시지가 오버라이드인지 여부가 결정됩니다.

메시지 메소드 구현

메시지 메소드의 구현은 다음 예제에서와 같이 상속된 메시지 메소드를 호출할 수 있습니다(Windows용).

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;
```

Linux나 크로스 플랫폼 프로그래밍에서는 다음과 동일한 예제를 사용하게 됩니다.

```
procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
  if Chr(Message.Code) = #13 then
    ...
  else
    inherited;
end;
```

inherited 문은 클래스 계층에서 역으로 검색하고 현재 메소드와 동일한 ID를 갖는 첫 번째 메시지 메소드를 호출하여 메시지 레코드를 자동으로 전달합니다. 조상 클래스가 특정 ID에 대한 메시지 메소드를 구현하지 않는 경우 inherited는 *TObject*에서 원래 정의된 *DefaultHandler* 메소드를 호출합니다.

*TObject*의 *DefaultHandler* 구현은 아무 동작도 실행하지 않고 반환됩니다. *DefaultHandler*를 오버라이드하여 클래스 자신의 기본 메시지 처리를 구현할 수 있습니다. Windows에서 VCL 컨트롤의 *DefaultHandler* 메소드는 Windows *DefWindowProc* 함수를 호출합니다.

메시지 디스패칭

메시지 핸들러는 직접 호출되는 일이 거의 없습니다. 그 대신, *TObject*에서 상속된 *Dispatch* 메소드를 사용하여 메시지가 객체에 디스패치됩니다.

```
procedure Dispatch(var Message);
```

*Dispatch*에 전달된 *Message* 매개변수는 첫 번째 항목에 메시지 ID가 들어 있는 *Cardinal* 타입의 필드인 레코드여야 합니다.

*Dispatch*는 클래스 계층에서 역으로 검색하고(호출된 객체의 클래스에서 시작) 전달된 ID의 첫 번째 메시지 메소드를 호출합니다. 지정 ID에 대한 메시지 메소드가 없으면 *Dispatch*는 *DefaultHandler*를 호출합니다.

속성

필드와 마찬가지로 속성은 객체의 속성(attribute)을 정의합니다. 그러나 필드가 내용을 확인하고 변경할 수 있는 저장소 위치인 반면, 속성은 해당 데이터를 읽고 수정하는 특정 동작을 연결합니다. 속성(attribute)은 객체 속성에 대한 액세스를 제어하며 속성을 계산할 수 있게 해줍니다.

속성 선언은 이름과 타입을 지정하며 하나 이상의 액세스 지정자를 포함합니다. 속성 선언의 구문은 다음과 같습니다.

```
property propertyName[indexes]: type index integerConstant specifiers;
```

여기서

- *propertyName*은 유효한 식별자입니다.
- [*indexes*]는 옵션이고 세미콜론으로 구분되는 일련의 매개변수 선언의 순서를 나타냅니다. 각 매개변수 선언은 $identifier_1$, ..., $identifier_n$: *type*과 같은 형태를 가집니다. 자세한 내용은 7-19 페이지의 "배열 속성"을 참조하십시오.
- *type* 은 이미 정의되었거나 이전에 선언된 데이터 타입이어야 합니다. 즉, `property Num 0..9 ...`와 같은 속성 선언은 잘못되었습니다.
- `index` *integerConstant* 절은 옵션입니다. 자세한 내용은 7-20 페이지의 "Index 지정자"를 참조하십시오.
- *specifiers*는 일련의 **read**, **write**, **stored**, **default**(또는 **nodefault**) 및 **implements** 지정자입니다. 모든 속성 선언에는 적어도 하나의 **read** 또는 **write** 지정자가 있습니다. **implements**에 대한 내용은 10-6 페이지의 "위임(Delegation)으로 인터페이스 구현"을 참조하십시오.

속성은 속성의 액세스 지정자에 의해 정의됩니다. 필드와 달리 속성은 **var** 매개변수로 전달할 수 없으며 **@** 연산자를 속성에 사용할 수 없습니다. 이는 속성이 항상 메모리에 있는 것이 아니기 때문입니다. 그러나, 속성은 데이터베이스에서 값을 검색하거나 임의 값을 생성하는 경우에는 **read** 메소드를 가질 수 있습니다.

속성 액세스

모든 속성에는 **read** 지정자, **write** 지정자, 또는 둘 다 있습니다. 이러한 지정자를 *액세스 지정자*라고 하며 다음과 같은 형태를 가집니다.

```
read fieldOrMethod
write fieldOrMethod
```

여기서 *fieldOrMethod*는 속성과 동일한 클래스나 조상 클래스에서 선언된 필드나 메소드의 이름입니다.

- *fieldOrMethod*가 동일한 클래스에서 선언되는 경우 속성 선언 전에 선언해야 합니다. 조상 클래스에서 선언되는 경우에는 자손 클래스에서 볼 수 있어야 합니다. 즉, 다른 유닛에서 선언된 조상 클래스의 private 필드나 메소드일 수 없습니다.
- *fieldOrMethod*가 필드인 경우 속성과 동일한 타입이어야 합니다.
- *fieldOrMethod*가 메소드인 경우 오버로드될 수 없습니다. 또한, published 속성의 액세스 메소드는 기본 **register** 호출 규칙을 사용해야 합니다.
- **read** 지정자에서 *fieldOrMethod*가 메소드인 경우 결과 타입이 속성 타입과 동일하고, 매개변수가 없는 함수이어야 합니다.
- **write** 지정자에서 *fieldOrMethod*가 메소드인 경우 단일 값 또는 속성과 같은 타입의 **const** 매개변수를 가지는 프로시저여야 합니다.

예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
property Color:TColor read GetColor write SetColor;
```

GetColor 메소드는 다음과 같이 선언해야 합니다.

```
function GetColor:TColor;
```

SetColor 메소드는 다음 중 하나로 선언해야 합니다.

```
procedure SetColor(Value:TColor);
procedure SetColor(const Value:TColor};
```

*SetColor*의 매개변수 이름이 *Value*일 필요는 없습니다.

표현식에서 속성이 참조되면 **read** 지정자에 나열된 필드나 메소드를 사용하여 속성 값을 읽습니다. 지정문에서 속성이 참조되면 **write** 지정자에 나열된 필드나 메소드를 사용하여 속성 값을 씁니다.

아래 예제에서는 *Heading*이라는 published 속성을 사용하여 *TCompass*라는 클래스를 선언합니다. *Heading* 값은 *FHeading* 필드를 통해 읽고 *SetHeading* 프로시저를 통해 씁니다.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading:THeading;
    procedure SetHeading(Value:THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    ⋮
  end;
```

위와 같이 선언했다면, 문장은 다음과 같습니다.

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

위 문장은 다음 문장에 해당합니다.

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

TCompass 클래스에서 어떤 동작도 *Heading* 속성과 연결되지 않으면, **read** 연산은 *FHeading* 필드에 저장된 값을 읽어 옵니다. 반면, *Heading* 속성에 값을 할당하면 *FHeading* 필드에 새 값을 저장하고 다른 동작을 수행하는 *SetHeading* 메소드에 대한 호출로 번역됩니다. 예를 들어, *SetHeading*은 다음과 같이 구현됩니다.

```
procedure TCompass.SetHeading(Value:THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;  // update user interface to reflect new value
  end;
end;
```

선언에 **read** 지정자만을 포함하는 속성은 *읽기 전용* 속성이고, 선언에 **write** 지정자만을 포함하는 속성은 *쓰기 전용* 속성입니다. 읽기 전용 속성에 값을 할당하거나 표현식에 쓰기 전용 속성을 사용하면 오류가 발생합니다.

배열 속성

*배열 속성*은 인덱싱된 속성입니다. 배열 속성은 목록에 있는 항목, 컨트롤의 자식 컨트롤 및 비트맵의 픽셀 등을 나타낼 수 있습니다.

배열 속성의 선언은 인덱스의 이름 및 타입을 지정하는 매개변수 목록을 포함합니다. 예를 들면, 다음과 같습니다.

```
property Objects[Index:Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y:Integer]: TColor read GetPixel write SetPixel;
property Values[const Name:string): string read GetValue write SetValue;
```

매개변수 선언을 괄호 대신 대괄호로 묶는 것을 제외하면 인덱스 매개변수 목록의 형식은 프로시저 매개변수 목록이나 함수 매개변수 목록의 형식과 같습니다. 순서 타입 인덱스만을 사용할 수 있는 배열과 달리 배열 속성은 모든 타입의 인덱스를 허용합니다.

배열 속성에서 액세스 지정자는 필드가 아닌 메소드를 나열해야 합니다. **read** 지정자의 메소드는 속성의 인덱스 매개변수 목록에 나열된 매개변수의 수와 타입을 동일한 순서로 가지는 함수여야 합니다. **write** 지정자의 메소드는 속성의 인덱스 매개변수 목록에 나열된 매개변수의 수와 타입을 동일한 순서로 가지는 프로시저이고, 속성과 동일한 타입의 추가 값이나 **const** 매개변수여야 합니다.

예를 들어, 위에 나온 배열 속성의 액세스 메소드를 다음과 같이 선언할 수 있습니다.

```
function GetObject(Index:Integer):TObject;
function GetPixel(X, Y:Integer):TColor;
function GetValue(const Name: string):문자열
procedure SetObject(Index:Integer; Value:TObject);
procedure SetPixel(X, Y:Integer; Value:TColor);
procedure SetValue(const Name, Value:string};
```

배열 속성은 속성 식별자를 인덱싱하여 액세스합니다. 예를 들어, 다음과 같은 문장이 있습니다.

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

위의 문장은 다음에 해당합니다.

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

Linux에서 위에 나온 예제의 'C:\DELPHI\BIN' 대신 '/usr/local/bin'과 같은 경로를 사용할 수 있습니다.

배열 속성의 정의는 **default** 지시어 앞에 올 수 있습니다. 이러한 경우 배열 속성은 클래스의 기본 속성이 됩니다. 예를 들면, 다음과 같습니다.

```
type
  TStringArray = class
  public
    property Strings[Index:Integer]: string ...; default;
    ⋮
  end;
```

클래스가 기본 속성을 가지는 경우 *object.property*[*index*]와 동등한 *object*[*index*] 약어를 사용하여 해당 기본 속성에 액세스할 수 있습니다. 예를 들어, 위와 같은 선언에서 StringArray.Strings[7]은 StringArray[7]로 약어화할 수 있습니다. 클래스는 기본 속성을 하나만 가질 수 있습니다. 컴파일러에 따라 항상 객체의 기본 속성이 정적으로 결정되므로 자손 클래스에서 기본 속성을 변경하거나 숨기면 예기치 않은 동작이 일어날 수 있습니다.

Index 지정자

index 지정자를 사용하여 여러 속성이 다른 값을 나타내면서 동일한 액세스 메소드를 공유할 수 있습니다. index 지정자는 −2147483647과 2147483647 사이의 정수 상수 앞에 오는 **index** 지시어로 구성됩니다. 속성이 index 지정자를 가지는 경우 해당 **read** 및 **write** 지정자는 필드가 아닌 메소드를 나열해야 합니다. 예를 들면, 다음과 같습니다.

```
type
  TRectangle = class
  private
    FCoordinates:array[0.0.3] of Char;
    function GetCoordinate(Index:Integer):Longint;
    procedure SetCoordinate(Index:Integer; Value:Longint);
  public
    property Left:Longint index 0 read GetCoordinate write SetCoordinate;
    property Top:Longint index 1 read GetCoordinate write SetCoordinate;
    property Right:Longint index 2 read GetCoordinate write SetCoordinate;
    property Bottom:Longint index 3 read GetCoordinate write SetCoordinate;
    property Coordinates[Index:Integer]: Longint read GetCoordinate write
SetCoordinate;
    ⋮
  end;
```

index 지정자가 있는 속성의 액세스 메소드는 *정수* 타입의 추가 value 매개변수를 가져야합니다. value 매개변수는 **read** 함수에서는 맨 마지막 매개변수여야 하고, **write** 프로시저에서는 속성 값을 지정하는 매개변수 앞의 끝에서 두 번째 매개변수여야 합니다. 프로그램에서 속성에 액세스할 때 속성의 정수 상수는 액세스 메소드에 자동으로 전달됩니다.

위와 같이 선언했다면, *Rectangle*이 *TRectangle* 타입인 경우에 다음과 같습니다.

```
Rectangle.Right := Rectangle.Left + 100;
```

이 문장은 다음 문장과 동일합니다.

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

저장소 지정자

옵션인 **stored**, **default** 및 **nodefault** 지시어를 *저장소 지정자*라고 합니다. 저장소 지정자는 프로그램 동작에 아무런 영향을 주지는 않지만 런타임 타입 정보(RTTI)가 유지되는 방식을 제어합니다. 특히, 저장소 지정자는 폼 파일의 published 속성 값을 저장할지 여부를 결정합니다.

stored 지시어는 *True*, *False*, *부울* 필드 이름 또는 *부울* 값을 반환하는 매개변수가 없는 메소드 이름 앞에 와야 합니다. 예를 들면, 다음과 같습니다.

```
property Name: TComponentName read FName write SetName stored False;
```

속성이 **stored** 지시어를 갖지 않는 경우 `stored True`가 지정된 것처럼 처리됩니다.

default 지시어는 속성과 동일한 타입의 상수 앞에 와야 합니다. 예를 들면, 다음과 같습니다.

```
property Tag: Longint read FTag write FTag default 0;
```

값을 새로 지정하지 않고 상속된 **default** 값을 오버라이드하려면 **nodefault** 지시어를 사용합니다. 집합 기본 타입의 상위 및 하위 경계가 0과 31 사이의 순서 값을 가진다고 가정하면 **default** 및 **nodefault** 지시어는 순서 타입과 집합 타입에 대해서만 지원됩니다. 이러한 속성이 **default** 또는 **nodefault** 없이 선언되는 경우 **nodefault**가 지정된 것으로 간주됩니다. 실수, 포인터, 문자열에는 각각 암시적 **default** 값인 0, **nil** 및 ''(빈 문자열)이 있습니다.

컴포넌트 상태를 저장할 때 컴포넌트의 published 속성에 대한 저장소 지정자를 확인합니다. 속성의 현재 값이 해당 **default** 값과 다르고(또는 **default** 값이 없는 경우) **stored** 지정자가 *True*이면 속성 값이 저장됩니다. 그렇지 않으면 속성 값이 저장되지 않습니다.

참고 저장소 지정자는 배열 속성에 대해 지원되지 않습니다. **default** 지시어는 배열 속성 선언에서 사용될 때 다른 의미를 가집니다. 7-19 페이지의 "배열 속성"을 참조하십시오.

속성 오버라이드 및 재선언

타입을 지정하지 않는 속성 선언을 *속성 오버라이드*라고 합니다. 속성 오버라이드를 사용하여 속성의 상속된 가시성이나 지정자를 변경할 수 있습니다. 가장 간단한 오버라이드는 상속된 속성 식별자 앞에 오는 **property** 예약어만으로 구성되며 이 형태는 속성의

가시성을 변경하는 데 사용됩니다. 예를 들어, 조상 클래스가 속성을 protected로 선언하면 파생된 클래스는 해당 속성을 클래스의 public 또는 published 섹션으로 재선언할 수 있습니다. 속성 오버라이드는 **read**, **write**, **stored**, **default** 및 **nodefault** 지시어를 포함할 수 있으며 이러한 지시어는 상속된 해당 지시어를 오버라이드합니다. 오버라이드는 상속된 액세스 지정자를 교체하고 필요한 지정자를 추가하거나 속성의 가시성을 높일 수 있지만 액세스 지정자를 제거하거나 속성의 가시성을 줄일 수는 없습니다. 오버라이드는 상속된 인터페이스를 제거하지 않고 구현된 인터페이스 목록에 추가할 수 있는 **implements** 지시어를 포함할 수 있습니다.

다음 선언은 속성 오버라이드의 사용법을 보여줍니다.

```
type
  TAncestor = class
   ⋮
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    ⋮
  end;
type
  TDerived = class(TAncestor)
   ⋮
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    ⋮
  end;
```

*Size*의 오버라이드는 **write** 지정자를 추가하여 속성을 수정할 수 있도록 합니다. *Text*와 *Color*의 오버라이드를 통해 속성의 가시성을 protected에서 published로 변경할 수 있습니다. 또한 *Color*의 속성 오버라이드는 속성 값이 *clBlue*가 아닌 경우 속성이 저장되도록 지정합니다.

타입 식별자를 포함하는 속성의 재선언은 상속된 속성을 오버라이드하지 않고 숨깁니다. 이는 새로운 속성이 상속된 속성과 동일한 이름으로 만들어진다는 것을 의미합니다. 타입을 지정하는 모든 속성 선언은 완전한 선언이어야 하고, 따라서 최소한 하나의 액세스 지정자를 포함해야 합니다.

속성이 파생된 클래스에서 숨겨져 있거나 오버라이드되어 있는지 여부에 관계 없이 속성 룩업(look up)은 항상 *정적*입니다. 즉, 객체를 식별하는 데 사용된 변수의 선언(컴파일 타임) 타입에 따라 해당 속성 식별자의 해석이 결정됩니다. 따라서 다음 코드를 실행한 후에 값을 읽거나 *MyObject.Value*에 지정하면 *MyObject*가 *TDescendant*의 인스턴스를 가지고 있어도 *Method1*이나 *Method2*가 호출됩니다. 그러나 *MyObject*를 *TDescendant*로 타입 변환하여 자손 클래스의 속성과 해당 액세스 지정자에 액세스할 수 있습니다.

```
type
  TAncestor = class
     ⋮
```

```
    property Value: Integer read Method1 write Method2;
  end;
  TDescendant = class(TAncestor)
    ⋮
    property Value: Integer read Method3 write Method4;
  end;
var MyObject:TAncestor;
 ⋮
MyObject := TDescendant.Create;
```

클래스 참조

클래스의 인스턴스인 객체가 아닌 클래스 자체에서 작업이 실행되는 경우가 종종 있습니다. 예를 들어, 클래스 참조를 사용하여 생성자 메소드를 호출할 때 이런 경우가 발생합니다. 클래스 이름을 사용하면 항상 특정 클래스를 참조할 수 있지만 클래스를 값으로 갖는 변수나 매개변수를 선언해야 하는 경우에는 *클래스 참조 타입*이 필요합니다.

클래스 참조 타입

종종 *메타클래스*라고 하는 클래스 참조 타입의 구문은 다음과 같습니다.

```
class of type
```

여기서 *type*은 임의의 클래스 타입입니다. *type* 식별자 자체는 타입이 `class of` *type* 값을 나타냅니다. $type_1$이 $type_2$의 조상인 경우 `class of` $type_1$을 `class of` $type_2$에 지정할 수 있습니다. 따라서

```
type TClass = class of TObject;
var AnyObj:TClass;
```

이 구문은 모든 클래스에 대한 참조를 가질 수 있는 *AnyObj*라는 변수를 선언합니다. 클래스 참조 타입의 정의는 변수 선언이나 매개변수 목록에 직접 사용할 수 없습니다. 모든 클래스 참조 타입의 변수에 **nil** 값을 지정할 수 있습니다.

클래스 참조 타입 사용의 사용 방법을 알려면 *Classes* 유닛의 *TCollection* 생성자 선언을 참조하십시오.

```
type TCollectionItemClass = class of TCollectionItem;
  ⋮
constructor Create(ItemClass: TCollectionItemClass};
```

이 선언에서와 같이 *TCollection* 인스턴스 객체를 만들려면 *TCollectionItem*에서 상속받는 클래스의 이름을 생성자에 전달해야 합니다.

클래스 참조 타입은 컴파일 시 실제 타입을 알 수 없는 클래스 또는 객체에서 클래스 메소드나 가상 생성자를 호출할 때 유용합니다.

생성자 및 클래스 참조

클래스 참조 타입의 변수를 사용하여 생성자를 호출할 수 있습니다. 그러면 컴파일 시 타입을 알 수 없는 객체 구문을 허용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
  const ControlName: string; X, Y, W, H:Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;
```

CreateControl 함수는 클래스 참조 매개변수에게 만들 컨트롤 종류를 지시하도록 요청합니다. CreateControl 함수는 이 매개변수를 사용하여 클래스의 생성자를 호출합니다. 클래스 타입 식별자는 클래스 참조 값을 나타내므로 *CreateControl*에 대한 호출을 통해 인스턴스를 만들 클래스의 식별자를 지정할 수 있습니다. 예를 들면, 다음과 같습니다.

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

클래스 참조를 사용하여 호출한 생성자는 일반적으로 가상(virtual)입니다. 호출에 의해 활성화되는 생성자 구현은 클래스 참조의 런타임 타입에 따라 다릅니다.

클래스 연산자

모든 클래스는 객체 및 객체의 직계 조상의 클래스에 대한 참조를 각각 반환하는 *ClassType* 및 *ClassParent*라는 *TObject* 메소드로부터 상속됩니다. 두 메소드는 더 특정한 타입으로 타입 변환할 수 있는 *TClass* 타입(여기서 `TClass = class of TObject`) 값을 반환합니다. 또한 모든 클래스는 호출된 객체가 지정된 클래스의 자손인지 여부를 테스트하는 *InheritsFrom*이라는 메소드를 상속받습니다. 이러한 메소드는 **is** 및 **as** 연산자에 의해 사용되며 연산자를 직접 호출하는 경우는 거의 없습니다.

is 연산자

동적 타입 검사를 수행하는 **is** 연산자는 객체의 실제 런타임 클래스를 확인하는 데 사용됩니다. 다음 표현식에서

object `is` *class*

이 구문은 *object*가 *class*나 자손 클래스의 인스턴스인 경우 *True*를 반환합니다. 그외에는 *False*를 반환합니다. *object*가 **nil**이면 결과는 *False*가 됩니다. *object*의 선언된 타입이 *class*와 관련이 없으면, 즉 타입이 다르고 object가 class의 자손 또는 조상이 아니면 오류가 발생합니다. 예를 들면, 다음과 같습니다.

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

이 문장은 변수가 참조하는 객체가 *TEdit*의 인스턴스이거나 자손임을 먼저 확인한 다음, *TEdit*로 변수를 타입 변환합니다.

as 연산자

as 연산자는 확인된 타입 변환을 실행합니다. 다음 표현식에서

object as *class*

이 구문은 *class*에서 제공한 타입과 함께 *object*와 동일한 객체에 대한 참조를 반환합니다. 런타임 시 *object*는 *class* 또는 자손 클래스의 인스턴스나 **nil**이어야 합니다. 그렇지 않을 경우 예외가 발생합니다. *object*의 선언된 타입이 *class*와 관련이 없으면, 즉 타입이 다르고 object가 class의 자손 또는 조상이 아니면 컴파일 오류가 발생합니다. 예를 들면, 다음과 같습니다.

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

연산자 우선 순위 규칙상 종종 **as** 타입 변환에 괄호를 사용해야 합니다. 예를 들면, 다음과 같습니다.

```
(Sender as TButton).Caption := '&Ok';
```

클래스 메소드

클래스 메소드는 객체 대신 클래스에서 작동하는 메소드이며 생성자는 제외합니다. 클래스 메소드 정의는 **class** 예약어로 시작해야 합니다. 예를 들면, 다음과 같습니다.

```
type
  TFigure = class
  public
    class function Supports(Operation:string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ⋮
  end;
```

클래스 메소드의 선언 정의도 **class**로 시작해야 합니다. 예를 들면, 다음과 같습니다.

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  ⋮
end;
```

클래스 메소드의 정의 선언에서 *Self* 식별자는 메소드가 호출된 클래스를 나타내거나 메소드가 정의된 클래스의 자손 클래스를 나타냅니다. 클래스 *C*에서 메소드를 호출하면 *Self*는 class of *C* 타입입니다. 따라서 *Self*를 사용하여 필드, 속성 및 기본(객체) 메소드에 액세스할 수 없지만 Self를 사용하여 생성자 및 다른 클래스 메소드를 호출할 수는 있습니다.

클래스 메소드는 클래스 참조나 객체 참조를 통해 호출할 수 있습니다. 클래스 메소드가 객체 참조를 통해 호출될 때 객체 클래스는 *Self* 값이 됩니다.

예외

프로그램의 정상적인 실행 중 오류나 기타 프로그램의 정상적인 실행을 방해하는 이벤트가 발생하면 *예외*가 발생합니다. 예외는 정상적인 프로그램 로직과 오류 처리를 분리할 수 있도록 *예외 핸들러*로 제어를 넘겨줍니다. 예외는 객체이므로 상속을 통해 계층으로 그룹화할 수 있으며 기존 코드에 영향을 주지 않고 새로운 예외를 만들 수 있습니다. 예외는 오류 메시지와 같은 정보를 발생한 위치에서 처리된 위치로 전달할 수 있습니다.

애플리케이션에서 *SysUtils* 유닛을 사용할 때 모든 런타임 오류는 자동으로 예외로 변환됩니다. 메모리 부족, 0으로 나누기, 일반 보호 오류 등으로 애플리케이션을 종료시키는 오류를 발견하여 처리할 수 있습니다.

예외를 사용하는 시기

예외는 프로그램을 중단하거나 조건문을 이상하게 계산하지 않고 런타임 오류를 원만하게 처리할 수 있게 해줍니다. 그러나 오브젝트 파스칼의 예외 처리 메커니즘은 복잡하여 자칫 문제 해결이 비효율적일 수 있으므로 상황을 잘 판단하여 사용해야 합니다. 어떤 이유로도 예외는 발생할 수 있으며 어떤 블록이든지 **try...except** 또는 **try...finally** 문을 사용하여 예외 처리할 수 있지만 실제로 특별한 경우에 예외 처리를 사용하는 것이 가장 좋습니다.

예외 처리는 드물게 발생하거나 판단하기 어려운 오류에 적합하지만, 결과가 프로그램 작동 중지와 같은 갑작스런 오류가 발생하거나, **if...then** 문으로 테스트하기 복잡하거나, 운영 체제에서 발생한 예외나 제어할 수 없는 소스 코드를 갖는 루틴에서 발생한 예외에 응답해야 하는 경우에는 적합하지 않습니다. 예외는 일반적으로 하드웨어, 메모리, I/O 및 운영 체제 오류에서 사용됩니다.

경우에 따라 조건문은 오류를 테스트하기 위해 가장 좋은 방법입니다. 예를 들어, 파일을 열기 전에 파일이 존재하는지 확인한다고 가정해 보십시오. 다음과 같은 방법으로 할 수 있습니다.

```
try
  AssignFile(F, FileName);
  Reset(F);  // raises an EInOutError exception if file is not found
except
  on Exception do ...
end;
```

그러나 다음을 사용하여 예외 처리의 오버헤드를 피할 수도 있습니다.

```
if FileExists(FileName) then  // returns False if file is not found; raises no
exception
begin
  AssignFile(F, FileName);
  Reset(F);
end;
```

소스 코드의 어느 곳에서나 *Assertions*으로 부울 조건을 테스트할 수 있습니다. *Assert* 문이 실패하면 프로그램이 중단되거나(*SysUtils* 유닛을 사용하는 경우) *EAssertionFailed* 예외가 발생합니다. Assertions는 발생을 예측할 수 없는 조건을

테스트하기 위해서만 사용되어야 합니다. 자세한 내용은 *Assert* 표준 프로시저의 온라인 도움말을 참조하십시오.

예외 타입 선언

예외 타입은 다른 클래스와 마찬가지로 선언됩니다. 실제로 클래스의 인스턴스를 예외로 사용할 수 있지만 *SysUtils*에 정의된 *Exception* 클래스에서 예외를 파생하는 것이 바람직합니다.

상속을 통해 예외를 패밀리로 그룹화할 수 있습니다. 예를 들어, 다음과 같은 *SysUtils* 선언은 산술 오류의 예외 타입 패밀리를 정의합니다.

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

위와 같이 선언했다면, *EInvalidOp*, *EZeroDivide*, *EOverflow* 및 *EUnderflow*를 처리하는 단일 *EMathError* 예외 핸들러를 정의할 수 있습니다.

예외 클래스는 경우에 따라 오류에 대한 추가 정보를 전달하는 필드, 메소드 및 속성을 정의합니다. 예를 들면, 다음과 같습니다.

```
type EInOutError = class(Exception)
    ErrorCode:Integer;
  end;
```

예외 발생 및 처리

예외 객체를 만들려면 **raise** 문 내에서 예외 클래스의 생성자를 호출합니다. 예를 들면, 다음과 같습니다.

```
raise EMathError.Create;
```

일반적으로 **raise** 문의 형태는 다음과 같습니다.

```
raise object at address
```

여기서 *object* 및 **at** *address*는 모두 옵션입니다. *object*를 생략하면 이 문장에서 현재 예외가 재발생합니다. 7-30 페이지의 "예외 재발생"을 참조하십시오. *address*가 지정되면 일반적으로 프로시저나 함수에 대한 포인터를 가리킵니다. 오류가 실제로 발생한 위치보다 앞선 스택의 위치에서 예외를 발생시키려면 이 옵션을 사용합니다.

*발생*된 예외(**raise** 문에서 참조된 예외)는 특정 예외 처리 로직에 의해 처리됩니다. **raise** 문은 일반적인 방법으로 제어를 반환하지 않습니다. 대신, 특정 클래스의 예외를 처리할 수 있는 가장 내부 예외 핸들러로 제어를 전송합니다. 내부 핸들러는 가장 최근에 들어왔지만 아직 빠져나가지 않은 **try...except** 블록 내에 있는 핸들러입니다.

예를 들어, 아래 함수는 문자열을 정수로 변환하여 결과 값이 지정된 범위를 벗어나는 경우 *ERangeError* 예외를 발생시킵니다.

```
function StrToIntRange(const S: string; Min, Max:Longint):Longint;
begin
  Result := StrToInt(S);  // StrToInt is declared in SysUtils
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

CreateFmt 메소드가 **raise** 문에서 호출되었다는 점에 유의하십시오. *예외*와 예외의 자손은 예외 메시지와 컨텍스트 ID를 만들 수 있는 다른 방법을 제공하는 특별한 생성자를 가집니다. 자세한 내용은 온라인 도움말을 참조하십시오.

발생한 예외는 처리된 후 자동으로 소멸됩니다. 발생한 예외를 절대 수동으로 제거하지 마십시오.

참고 유닛의 초기화 섹션에서 예외가 발생하면 의도한 결과가 나타나지 않을 수도 있습니다. 일반적인 예외 지원은 *SysUtils* 유닛에서 제공되며 초기화되어야만 지원됩니다. 초기화 중에 예외가 발생하는 경우 *SysUtils*를 비롯한 초기화된 모든 유닛이 완료되고 예외가 재발생합니다. 그런 다음 일반적으로 프로그램이 중단되고 예외를 발견하여 처리합니다.

Try...except 문

예외는 **try...except** 문 내에서 처리됩니다. 예를 들면, 다음과 같습니다.

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

이 문에서는 *Y*를 *Z*로 나눕니다. 그러나 *EZeroDivide*(0으로 나누기) 예외가 발생하면 *HandleZeroDivide*라는 루틴을 호출합니다.

try...except 문의 구문은 다음과 같습니다.

try *statements* except *exceptionBlock* end

여기서 *statements*는 세미콜론으로 구분되는 일련의 문장이고 *exceptionBlock*은 다음 중 하나입니다.

- 일련의 다른 문장.
- 일련의 예외 핸들러. 경우에 따라 다음 문장이 뒤에 옵니다.

  else *statements*

예외 핸들러는 다음과 같은 형태를 가집니다.

on *identifier*: *type* do *statement*

여기서 *identifier*:은 옵션이고(*identifier*는 유효한 식별자) *type*은 예외를 나타내는 데 사용되는 타입이며 *statement*는 임의의 문장입니다.

try...except 문은 첫 *statements* 목록의 문장을 실행합니다. 예외가 발생하지 않으면 예외 블록(*exceptionBlock*)은 무시되고 제어가 프로그램의 다음 부분으로 넘어갑니다.

statements 목록의 **raise** 문에 의해 또는 *statements* 목록에서 호출된 프로시저나 함수에 의해 첫 *statements* 목록의 실행 중 예외가 발생하면 예외 "처리"를 시도합니다.

- 예외 블록의 핸들러 중 하나가 예외와 일치하는 경우 첫 번째 핸들러로 제어가 전달됩니다. 핸들러의 *type*이 예외 클래스 또는 해당 클래스의 조상인 예외 핸들러가 예외와 "일치"하는 핸들러입니다.
- 일치하는 핸들러가 없으면 **else** 절의 *statement*으로 제어가 넘어갑니다.
- 예외 블록이 예외 핸들러가 없는 일련의 문장인 경우 목록의 첫 번째 문장으로 제어가 넘어갑니다.

위 조건 중 어떤 것과도 만족하지 않는 경우 가장 최근에 들어왔지만 아직 빠져나가지 않은 **try...except** 문의 예외 블록에서 계속 찾습니다. 적절한 핸들러, **else** 절 또는 문장 목록이 없는 경우 **try...except** 문의 다음 상위 단계로 계속 찾아 나갑니다. 가장 외부 **try...except** 문에 도달해도 예외가 처리되지 않는 경우 프로그램은 종료됩니다.

예외가 처리될 때 스택은 **try...except** 문이 들어 있는 프로시저나 함수를 역으로 추적하며 실행될 예외 핸들러, **else** 절 또는 문장 목록으로 제어가 전달됩니다. 이러한 과정에서 예외가 처리되는 **try...except** 문에 들어간 다음 나오는 모든 프로시저와 함수 호출은 제거됩니다. 그런 다음 예외 객체는 *Destroy* 소멸자에 대한 호출을 통해 자동으로 소멸되며 제어는 **try...except** 문의 다음 문장으로 전달됩니다. *Exit*, *Break* 또는 *Continue* 표준 프로시저를 호출하여 예외 핸들러에 제어를 그대로 남겨두어도 예외 객체는 자동으로 소멸됩니다.

아래 예제에서 첫 번째 예외 핸들러는 0으로 나누기 예외를 처리하고 두 번째 예외 핸들러는 오버플로 예외를 처리하며 마지막 예외 핸들러는 다른 모든 산술 예외를 처리합니다. *EMathError*는 다른 두 예외 클래스의 조상이므로 예외 블록의 마지막에 나타나며 처음에 나타나는 경우에는 다른 두 핸들러가 호출되지 않습니다.

```
try
  ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

예외 핸들러는 예외 클래스의 이름 앞에 식별자를 지정할 수 있습니다. 이는 식별자가 **on...do**에 뒤에 나오는 문장을 실행하는 중의 예외 객체를 나타내도록 선언합니다. 식별자의 유효 범위(scope)는 해당 문장으로 제한됩니다. 예를 들면, 다음과 같습니다.

```
try
  ⋮
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

예외 블록이 **else** 절을 지정하는 경우 **else** 절은 블록의 예외 핸들러에 의해 처리되지 않는 예외를 처리합니다. 예를 들면, 다음과 같습니다.

```
try
 ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

여기서 else 절은 *EMathError*가 아닌 예외를 처리합니다.

그러나 예외 핸들러가 없는 예외 블록은 문장 목록으로만 구성되고 모든 예외를 처리합니다. 예를 들면, 다음과 같습니다.

```
try
 ⋮
except
  HandleException;
end;
```

여기서 *HandleException* 루틴은 **try**와 **except** 사이의 문을 실행한 결과로 발생하는 예외를 처리합니다.

예외 재발생

예약어 **raise**가 뒤에 오는 객체 참조 없이 예외 블록에 나오는 경우 블록에 의해 처리되는 예외를 발생합니다. 이를 통해 예외 핸들러가 제한된 방법으로 오류에 응답한 다음 예외를 *재발생*시킬 수 있습니다. 예외 재발생은 프로시저나 함수가 예외 발생 후 클린업 해야 하지만 완전히 예외를 처리하지 못할 경우 유용합니다.

예를 들어, *GetFileList* 함수는 *TStringList* 객체를 할당한 다음 해당 객체를 지정 검색 경로와 일치하는 파일 이름으로 채웁니다.

```
function GetFileList(const Path:string): TStringList;
var
  I:Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
    begin
      Result.Add(SearchRec.Name);
      I := FindNext(SearchRec);
    end;
  except
    Result.Free;
    raise;
  end;
end;
```

*GetFileList*는 *TStringList* 객체를 만든 다음 *SysUtils*에서 정의된 *FindFirst*와 *FindNext* 함수를 사용하여 객체를 초기화합니다. 예를 들어, 검색 경로가 잘못되었거나 문자열 목록을 채울 메모리가 부족하여 초기화가 실패하는 경우 호출자가 문자열 목록의 존재 여부를 아직 알지 못하므로 *GetFileList*는 새로운 문자열 목록을 없애야 합니다. 이러한 이유로 문자열 목록의 초기화는 **try...except** 문에서 실행됩니다. 예외가 발생하면 문장의 예외 블록은 문자열 목록을 없앤 다음 예외를 재발생시킵니다.

중첩 예외

예외 핸들러에서 실행된 코드는 자체적으로 예외를 발생시키고 처리할 수 있습니다. 이러한 예외가 예외 핸들러 내에서 처리되는 한 원래 예외에 영향을 주지 않습니다. 그러나 예외 핸들러에서 발생한 예외가 원래 핸들러를 벗어나 상위 단계로 핸들러를 찾아 나가기 때문에 원래 예외는 소실됩니다. 아래의 *Tan* 함수가 이를 보여줍니다.

```
type
  ETrigError = class(EMathError);

function Tan(X: Extended):Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
  end;
end;
```

*Tan*의 실행 중 *EMathError* 예외가 발생하면 예외 핸들러는 *ETrigError*를 발생시킵니다. *Tan*은 *ETrigError*에 핸들러를 제공하지 않으므로 원래 핸들러를 벗어나 상위 단계로 핸들러로 예외가 전달되기 때문에 *EMathError* 예외는 소멸됩니다. 호출자에게는 *Tan* 함수가 *ETrigError* 예외를 발생시킨 것처럼 나타납니다.

Try...finally 문

간혹 연산의 특정 부분이 완료되었는지, 연산이 예외에 의해 중단되었는지 여부를 확인하고자 할 경우가 있습니다. 예를 들어, 루틴이 리소스를 제어할 때는 루틴이 비정상적으로 종료되더라도 리소스가 확실히 해제되도록 하는 것이 중요합니다. 이러한 상황에서 **try...finally** 문을 사용할 수 있습니다.

다음 예제는 파일을 열어 처리하는 코드가 실행 중 오류가 발생하더라도 어떻게 파일을 최종적으로 닫을 수 있는지를 보여줍니다.

```
Reset(F);
try
  ⋮  // process file F
finally
  CloseFile(F);
end;
```

try...finally 문의 구문은 다음과 같습니다.

try *statementList*$_1$ finally *statementList*$_2$ end

여기서 각 *statementList*는 세미콜론으로 구분된 일련의 문장입니다. **try...finally** 문은 $statementList_1$ (**try** 절)의 문장을 실행합니다. $statementList_1$이 예외없이 종료되면 $statementList_2$ (**finally** 절)가 실행됩니다. $statementList_1$의 실행 중 예외가 발생하면 $statementList_2$로 제어가 전달되고, $statementList_2$가 실행을 끝내면 예외가 재발생합니다. *Exit*, *Break* 또는 *Continue* 프로시저를 호출하여 제어가 $statementList_1$에 남아 있더라도 $statementList_2$는 자동으로 실행됩니다. 따라서 **finally** 절은 **try** 절이 어떻게 종료되든지 상관없이 항상 실행됩니다.

예외가 발생했지만 **finally** 절에서 처리되지 않으면 **try...finally** 문 외부로 예외가 전달되며 **try** 절에서 이미 발생된 예외는 소실됩니다. 그러므로 **finally** 절은 지역적으로 발생한 예외를 모두 처리하여 다른 예외의 전달을 방해하지 않아야 합니다.

표준 예외 클래스 및 루틴

SysUtils 유닛은 *ExceptObject*, *ExceptAddr* 및 *ShowException*을 비롯한 예외 처리를 위한 몇몇 표준 루틴을 선언합니다. *SysUtils* 및 다른 유닛 또한 *Exception*에서 파생된(*OutlineError* 제외) 수십 개의 예외 클래스를 포함합니다.

Exception 클래스는 상황에 맞는 온라인 설명서를 위한 오류 설명 및 컨텍스트 ID를 전달하는 데 사용할 수 있는 *Message* 및 *HelpContext*라는 속성을 가집니다. 이 클래스는 다른 방법으로 설명 및 컨텍스트 ID를 지정할 수 있도록 다양한 생성자 메소드도 정의합니다. 자세한 내용은 온라인 도움말을 참조하십시오.

8

표준 루틴 및 I/O

이 장에서는 텍스트 및 파일 I/O에 대해 설명하고 표준 라이브러리 루틴을 요약합니다. 이 장에서 다루는 대부분의 프로시저 및 함수는 *시스템* 유닛에 정의되어 있으며, 암시적으로 모든 애플리케이션과 함께 컴파일됩니다. 여기에 나열되지 않은 것은 컴파일러에서 기본 제공되지만 *시스템* 유닛에 있는 것처럼 처리됩니다.

일부 표준 루틴은 *SysUtils* 같은 유닛에 정의되어 있습니다. 이 유닛을 프로그램에서 사용하려면 유닛을 **uses** 절에 나열해야 합니다. 그러나 *시스템* 유닛은 **uses** 절에 나열할 수 없으며, *시스템* 유닛을 수정하거나 명시적으로 다시 빌드할 수 없습니다.

나열된 루틴에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

파일 입력 및 출력

아래 표에는 입력 및 출력 루틴이 나열되어 있습니다.

표 8.1 입력과 출력 프로시저 및 함수

프로시저 또는 함수	설명
Append	내용을 추가하기 위해 기존 텍스트 파일을 엽니다.
AssignFile	외부 파일의 이름을 파일 변수에 할당합니다.
BlockRead	타입이 지정되지 않은 파일에서 하나 이상의 레코드를 읽습니다.
BlockWrite	타입이 지정되지 않은 파일에 하나 이상의 레코드를 씁니다.
ChDir	현재 디렉토리를 변경합니다.
CloseFile	열려 있는 파일을 닫습니다.
Eof	파일의 파일 끝(end-of-file) 상태에 이르렀을 경우 True값을 반환합니다.
Eoln	텍스트 파일의 줄 끝(end-of-line) 상태에 이르렀을 경우 True값을 반환합니다.
Erase	외부 파일을 지웁니다.
FilePos	타입이 지정된 파일이나 타입이 지정되지 않은 파일의 현재 위치를 반환합니다.
FileSize	파일의 현재 크기를 반환합니다. 텍스트 파일에 대해서는 사용하지 않습니다.
Flush	출력 텍스트 파일의 버퍼를 플러시합니다.

표 8.1 입력과 출력 프로시저 및 함수 (계속)

프로시저 또는 함수	설명
GetDir	지정된 드라이브의 현재 디렉토리를 반환합니다.
IOResult	마지막으로 수행된 I/O 함수의 상태인 정수값을 반환합니다.
MkDir	하위 디렉토리를 만듭니다.
Read	파일에서 하나 이상의 값을 하나 이상의 변수로 읽어 들입니다.
Readln	*Read*와 동일하지만 텍스트 파일의 한 줄만 읽어 들입니다.
Rename	외부 파일의 이름을 재지정합니다.
Reset	기존 파일을 엽니다.
Rewrite	새 파일을 만들어 엽니다.
RmDir	빈 하위 디렉토리를 제거합니다.
Seek	타입이 지정된 파일이나 타입이 지정되지 않은 파일의 현재 위치를 지정된 컴포넌트로 이동시킵니다. 텍스트 파일에서는 사용하지 않습니다.
SeekEof	텍스트 파일의 파일 끝(end-of-file) 상태를 반환합니다.
SeekEoln	텍스트 파일의 줄 끝(end-of-line) 상태를 반환합니다.
SetTextBuf	I/O 버퍼를 텍스트 파일에 할당니다.
Truncate	현재 파일 위치에서 타입이 지정된 파일이나 타입이 지정되지 않은 파일을 잘라냅니다.
Write	파일에 하나 이상의 값을 씁니다.
Writeln	*Write*와 동일하지만 텍스트 파일에 한 줄을 쓰고 줄 끝에 eoln(end-of-line)을 표시합니다.

파일 변수는 타입이 파일 타입인 변수입니다. 파일에는 *타입이 지정된 파일*, *텍스트 파일* 및 *타입이 지정되지 않은 파일* 세 가지가 있습니다. 파일 타입 선언 구문은 5-24 페이지의 "파일 타입(File types)"에서 다룹니다.

AssignFile 프로시저 호출을 통해 파일 변수를 외부 파일과 연결해야만 파일 변수를 사용할 수 있습니다. 외부 파일은 일반적으로 명명된 디스크 파일입니다. 외부 파일로는 키보드나 디스플레이 같은 장치가 될 수도 있습니다. 외부 파일은 파일에 기록된 정보를 저장하거나 파일에서 읽은 정보를 제공합니다.

외부 파일과 일단 연결되면 파일 변수는 입력 또는 출력을 할 수 있도록 "열려야" 합니다. 기존 파일은 *Reset* 프로시저로 열 수 있고 새 파일은 *Rewrite* 프로시저로 만들거나 열 수 있습니다. *Reset*으로 연 텍스트 파일은 읽기 전용이고 *Rewrite*나 *Append*로 연 텍스트 파일은 쓰기 전용입니다. 타입이 지정된 파일과 타입이 지정되지 않은 파일은 *Reset* 또는 *Rewrite*으로 열었든 혹은 그렇지 않든 상관없이 항상 읽고 쓸 수 있습니다.

모든 파일은 일련의 컴포넌트들의 연속이고 각 컴포넌트는 파일의 컴포넌트 타입이나 레코드 타입을 가지고 있습니다. 컴포넌트는 0부터 번호가 매겨집니다.

일반적으로 파일은 순차적으로 액세스됩니다. 즉, 컴포넌트를 표준 프로시저 *Read*를 사용하여 읽거나 표준 프로시저 *Write*를 사용하여 쓰면 현재 파일 위치는 다음 번호의 파일 컴포넌트로 이동합니다. 현재 파일 위치를 지정된 컴포넌트로 이동시키는 표준 프로시저 *Seek*를 통해 타입이 지정된 파일과 타입이 지정되지 않은 파일을 임의 액세스할 수도 있습니다. 표준 함수 *FilePos* 및 *FileSize*는 현재 파일 위치와 파일 크기를 알아 내는 데 사용할 수 있습니다.

프로그램이 파일 처리를 완료하면 표준 프로시저 *CloseFile*을 사용하여 해당 파일을 닫아야 합니다. 파일을 닫아야 연결된 외부 파일이 업데이트됩니다. 그런 다음 파일 변수는 다른 외부 파일에 연결될 수 있습니다.

기본적으로, 표준 I/O 프로시저 및 함수 호출에 대해서는 자동으로 오류를 검사하며 오류가 발생하면 예외가 발생합니다. 예외 처리를 할 수 없으면 프로그램이 종료됩니다. **{$I+}** 및 **{$I−}** 컴파일러 지시어를 사용하면 자동 검사 기능을 켜고 끌 수 있습니다. I/O 검사 기능이 꺼져 있으면, 즉 프로시저 또는 함수 호출이 **{$I−}** 상태에서 컴파일되면 I/O 오류는 예외를 발생시키지 않습니다. I/O 작동 결과를 검사하려면 표준 함수 *IOResult*를 대신 호출해야 합니다.

오류를 지우려면 *IOResult* 함수를 호출해야 합니다. 오류를 지우지 않고, 현재 상태가 **{$I+}**이면 다음 번 I/O 함수 호출은 *IOResult* 오류와 함께 실패합니다.

텍스트 파일

이 단원에서는 표준 타입 *텍스트*의 파일 변수를 사용하여 I/O를 요약 설명합니다.

텍스트 파일이 열리면 외부 파일은 특별한 방법으로 해석됩니다. 줄 단위로 서식이 지정된 일련의 문자를 나타내는 것으로 간주됩니다. 각 줄은 줄 끝(end−of−line) 표시(캐리지 리턴 문자, 그 뒤에 라인 피드 문자가 올 수도 있음)가 되어 있습니다. *텍스트* 타입은 `file of Char` 타입과 다릅니다.

텍스트 파일에는 *Char* 타입이 아닌 값을 읽고 쓸 수 있는 *Read*와 *Write*라는 특별한 형태가 있습니다. 이 값들은 문자 표현으로, 그리고 문자 표현에서 값으로 자동으로 바뀝니다. 예를 들면, *I*가 *Integer* 타입 변수라면 `Read(F, I)`는 일련의 숫자를 읽고 이를 십진수 정수로 해석한 다음 *I*에 저장합니다.

*Input*과 *Output*이라는 두 개의 표준 텍스트 파일 변수가 있습니다. 표준 파일 변수 *Input*은 운영 체제의 표준 입력(일반적으로, 키보드)에 연결된 읽기 전용 파일입니다. 표준 파일 변수 *Output*은 운영 체제의 표준 출력(일반적으로, 디스플레이 장치)에 연결된 출력 전용 파일입니다. 애플리케이션이 실행되기 전에 *Input*과 *Output*은 다음 문장이 실행된 것처럼 자동으로 열립니다.

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

참고 텍스트 I/O는 콘솔 애플리케이션, 즉 Project Options 대화 상자의 Linker 페이지에서 "Generate console application" 옵션을 선택했거나 **−cc** 명령줄 컴파일러 옵션으로 컴파일된 애플리케이션에서만 사용할 수 있습니다. 콘솔이 아닌 GUI 애플리케이션에서 *Input*이나 *Output*을 사용하여 읽거나 쓰려고 하면 I/O 오류가 생깁니다.

텍스트 파일에 사용할 수 있는 몇몇 표준 I/O 루틴은 매개변수로 명시적으로 주어진 파일 변수가 없어도 됩니다. 파일 매개변수가 생략되면 프로시저나 함수가 입력용인지 출력용인지에 따라서 *Input* 또는 *Output*이 기본값으로 설정됩니다. 예를 들면, `Read(X)`는 `Read(Input, X)`에 해당되며 `Write(X)`는 `Write(Output, X)`에 해당됩니다.

텍스트 파일에서 작동하는 입력 또는 출력 루틴 가운데 하나를 호출할 때 반드시 *AssignFile*을 사용하여 파일을 외부 파일에 연결해야 하고 *Reset*, *Rewrite* 또는 *Append*를 사용하여

열어야 합니다. *Reset*으로 연 파일을 출력용 프로시저나 함수에 전달하면 예외가 발생합니다. *Rewrite* 또는 *Append*로 연 파일을 또한 입력용 프로시저 또는 함수에 전달하면 예외가 발생합니다.

타입이 지정되지 않은 파일

타입이 지정되지 않은 파일은 타입이나 구조에 상관 없이 디스크 파일에 직접 액세스하는 데 주로 사용되는 저수준 I/O 채널입니다. 타입이 지정되지 않은 파일은 **file**로만 선언됩니다. 예를 들면, 다음과 같습니다.

```
var DataFile:file;
```

타입이 지정되지 않은 파일에서 *Reset* 및 *Rewrite* 프로시저를 사용하면 데이터 전송에 사용된 레코드 크기를 지정할 수 있는 추가 매개변수를 가질 수 있습니다. 전통적으로 레코드 크기는 기본적으로 128바이트입니다. 레코드 크기 1은 모든 파일의 정확한 크기를 반영하는 유일한 값입니다. (레코드 크기가 1이면 어떠한 부분 레코드도 불가능합니다.)

Read 및 *Write*를 제외하고 타입이 지정된 파일 표준 프로시저 및 함수 모두는 타입이 지정되지 않은 파일에서도 사용할 수 있습니다. *Read* 및 *Write* 대신에 *BlockRead* 및 *BlockWrite*라는 두 프로시저는 고속 데이터 전송에 사용됩니다.

텍스트 파일 장치 드라이버

사용자의 프로그램에서 고유한 텍스트 파일 장치 드라이버를 정의할 수 있습니다. 텍스트 파일 장치 드라이버는 오브젝트 파스칼의 파일 시스템과 일부 장치 사이의 인터페이스를 완벽하게 구현하는 네 가지 함수의 집합니다.

각 장치 드라이버를 정의하는 네 개의 함수는 *Open*, *InOut*, *Flush* 및 *Close*입니다. 각 함수의 함수 헤더는 다음과 같습니다.

```
function DeviceFunc(var F: TTextRec}:Integer;
```

여기서 *DeviceFunc*는 함수의 이름, 즉 *Open*, *InOut*, *Flush* 또는 *Close*입니다. *TTextRec* 타입에 대한 자세한 내용은 온라인 도움말을 참조하십시오. 장치 인터페이스 함수의 반환값은 *IOResult*에 의해 반환된 값이 됩니다. 반환값이 0이면 연산은 성공적입니다.

장치 인터페이스 함수를 특정 파일과 연결하려면 사용자 지정 *Assign* 프로시저를 작성해야 합니다. *Assign* 프로시저는 반드시 네 개의 장치 인터페이스 함수의 주소를 텍스트 파일 변수에서 네 개의 함수 포인터에 지정해야 합니다. 또한 이 프로시저는 *fmClosed* "매직" 상수를 *Mode* 필드에, 텍스트 파일 버퍼 크기를 *BufSize*에, 텍스트 파일 버퍼에 대한 포인터를 *BufPtr*에 저장해야 하고 *Name* 문자열을 지워야 합니다.

예를 들어, *DevOpen*, *DevInOut*, *DevFlush* 및 *DevClose*라는 장치 인터페이스 함수가 있다면 *Assign* 프로시저는 다음과 같을 것입니다.

```
procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
  begin
```

```
      Mode := fmClosed;
      BufSize := SizeOf(Buffer);
      BufPtr := @Buffer;
      OpenFunc := @DevOpen;
      InOutFunc := @DevInOut;
      FlushFunc := @DevFlush;
      CloseFunc := @DevClose;
      Name[0] := #0;
    end;
  end;
```

장치 인터페이스 함수는 파일 레코드의 *UserData* 필드를 사용하여 개별 정보를 저장할 수 있습니다. 이 필드는 제품 파일 시스템에 의해 수정할 수 없습니다.

장치 함수

텍스트 파일 장치 드라이버를 구성하는 함수는 다음과 같습니다.

Open 함수

Open 함수는 *Reset*, *Rewrite* 및 *Append* 표준 프로시저에 의해 호출되어 장치와 연결된 텍스트 파일을 엽니다. 입력 시 *Mode* 필드에는 *fmInput*, *fmOutput* 또는 *fmInOut*이 포함되어 *Open* 함수가 *Reset*, *Rewrite* 또는 *Append*에서 호출되었는지 여부를 나타냅니다.

Open 함수는 *Mode* 값에 따라 입력 또는 출력을 할 수 있도록 파일을 준비합니다. *Mode*가 *fmInOut*을 지정했으면(*Open*이 *Append*에 의해 호출되었음을 나타냄) *Open*이 반환되기 전에 *fmOutput*으로 변경되어야 합니다.

*Open*은 항상 다른 장치 인터페이스 함수보다 먼저 호출됩니다. 이러한 이유로 *AssignDev*는 *OpenFunc* 필드만 초기화하고 나머지 *Open*을 구성하는 벡터는 초기화하지 않습니다. *Mode*에 따라 *Open*은 입력용 함수 또는 출력용 함수에 대한 포인터를 설정할 수 있습니다. 이것으로 인해 현재 모드로부터 *InOut*, *Flush* 함수 및 *CloseFile* 프로시저는 생략됩니다.

InOut 함수

InOut 함수는 장치에 입력 또는 출력이 필요할 때마다 *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln* 및 *CloseFile* 표준 루틴에 의해 호출됩니다.

*Mode*가 *fmInput*이면 *InOut* 함수는 *BufSize* 문자를 `BufPtr^`에 읽어 오고 *BufEnd*에서 읽은 문자 수를 반환합니다. 또한 *BufPos*에 0을 저장합니다. *InOut* 함수가 입력 요청의 결과로 *BufEnd*에서 0을 반환하면 파일에 대한 *Eof*는 *True*가 됩니다.

*Mode*가 *fmOutput*이면 *InOut* 함수는 `BufPtr^`로부터 *BufPos* 문자를 쓰고 *BufPos*에 0을 반환합니다.

Flush 함수

Flush 함수는 각각의 *Read*, *Readln*, *Write* 및 *Writeln* 끝에서 호출됩니다. 이 함수는 텍스트 파일 버퍼를 옵션으로 플러시할 수 있습니다.

*Mode*가 *fmInput*이면 *Flush* 함수는 *BufPos* 및 *BufEnd*에 0을 저장하여 버퍼의 나머지 읽지 않은 문자를 플러시할 수 있습니다. 이 기능은 거의 사용되지 않습니다.

*Mode*가 *fmOutput*이면 *Flush* 함수는 장치에 쓰여진 텍스트가 장치에 바로 나타나도록 하는 *InOut* 함수와 똑같이 버퍼 내용을 쓸 수 있습니다. *Flush*가 아무 일도 하지 않으면 버퍼가 꽉 차거나 파일이 닫힐 때까지 텍스트는 장치에 나타나지 않습니다.

Close 함수

Close 함수는 *CloseFile* 표준 프로시저에 의해 호출되어 장치와 연결된 텍스트 파일을 닫습니다. 해당 프로시저가 열려는 파일이 이미 열려 있으면 *Reset*, *Rewrite* 및 *Append* 프로시저는 *Close*를 호출합니다. *Mode*가 *fmOutput*이면 *Close*를 호출하기 전에 파일 시스템은 *InOut* 함수를 호출하여 모든 문자가 장치에 기록되었는지 확인합니다.

Null 종료 문자열 처리

오브젝트 파스칼의 확장 구문을 사용하면 *Read*, *Readln*, *Str* 및 *Val* 표준 프로시저를 인덱스가 0부터 시작하는 문자 배열에 적용할 수 있고 *Write*, *Writeln*, *Val*, *AssignFile* 및 *Rename* 표준 프로시저를 인덱스가 0부터 시작하는 문자 배열 및 문자 포인터에 적용할 수 있습니다. 또한 Null 종료 문자열 처리를 위해 다음 함수가 제공됩니다. Null 종료 문자열에 대한 자세한 내용은5-13 페이지의 "Null 종료 문자열 사용"을 참조하십시오.

표 8.2 Null 종료 문자열 함수

함수	설명
StrAlloc	특정 크기의 문자 버퍼를 힙에 할당합니다.
StrBufSize	*StrAlloc* 또는 *StrNew*를 사용하여 할당된 문자 버퍼의 크기를 반환합니다.
StrCat	두 문자열을 연결합니다.
StrComp	두 문자열을 비교합니다.
StrCopy	문자열을 복사합니다.
StrDispose	*StrAlloc* 또는 *StrNew*를 사용하여 할당된 문자 버퍼를 해제합니다.
StrECopy	문자열을 복사하고 문자열 끝에 대한 포인터를 반환합니다.
StrEnd	문자열 끝에 대한 포인터를 반환합니다.
StrFmt	하나 이상의 값을 문자열 내로 서식화합니다.
StrIComp	대소문자를 구별하지 않고 두 문자열을 비교합니다.
StrLCat	결과 문자열의 주어진 최대 길이로 두 문자열을 연결합니다.
StrLComp	주어진 최대 길이로 두 문자열을 비교합니다.
StrLCopy	주어진 최대 길이까지 문자열을 복사합니다.
StrLen	문자열의 길이를 반환합니다.
StrLFmt	주어진 최대 길이로 하나 이상의 값을 문자열 내로 서식화합니다.
StrLIComp	대소문자를 구별하지 않고 주어진 최대 길이에 대해 두 문자열을 비교합니다.
StrLower	문자열을 소문자로 변환합니다.
StrMove	한 문자열의 문자 블록을 다른 문자열로 이동합니다.

표 8.2 Null 종료 문자열 함수 (계속)

함수	설명
StrNew	문자열을 힙에 할당합니다.
StrPCopy	파스칼 문자열을 Null 종료 문자열에 복사합니다.
StrPLCopy	주어진 최대 길이로 파스칼 문자열을 Null 종료 문자열에 복사합니다.
StrPos	문자열 내에서 특정 부분 문자열이 처음 나타나는 곳의 포인터를 반환합니다.
StrRScan	문자열 내에서 특정 문자가 마지막으로 나타나는 곳의 포인터를 반환합니다.
StrScan	문자열 내에서 특정 문자가 처음 나타나는 곳의 포인터를 반환합니다.
StrUpper	문자열을 대문자로 변환합니다.

표준 문자열 처리 함수에는 문자의 로케일 특정 정렬을 구현하고 멀티바이트를 사용할 수 있는 처리 함수가 있습니다. 멀티바이트 함수의 이름은 *Ansi*로 시작합니다. 예를 들면, *StrPos*의 멀티바이트 버전은 *AnsiStrPos*입니다. 멀티바이트 문자 지원은 운영 체제에 따라 다르며 현재 로케일을 기반으로 합니다.

와이드 문자 문자열

시스템 유닛은 Null 종료 와이드 문자 문자열을 1바이트나 2바이트의 긴 문자열로 상호간에 변환하여 사용할 수 있는 *WideCharToString*, *WideCharLenToString* 및 *StringToWideChar* 함수를 제공합니다.

와이드 문자 문자열에 대한 자세한 내용은 5-13 페이지의 "확장 문자 집합 정보"를 참조하십시오.

기타 표준 루틴

아래의 표는 Borland 제품 라이브러리에서 자주 사용되는 프로시저 및 함수를 보여줍니다. 이 표는 표준 루틴의 전체 목록은 아닙니다. 이 루틴 및 기타 루틴에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

표 8.3 기타 표준 루틴

프로시저 또는 함수	설명
Abort	오류를 보고하지 않고 프로세스를 끝냅니다.
Addr	지정된 객체에 대한 포인터를 반환합니다.
AllocMem	메모리 블록을 할당하고 각 바이트를 0으로 초기화합니다.
ArcTan	특정 숫자의 아크탄젠트를 계산합니다.
Assert	부울 표현식이 *True*인지 테스트합니다.
Assigned	할당되지 않은 **nil** 포인터 타입 또는 프로시저 타입 변수를 테스트합니다.
Beep	컴퓨터 스피커를 사용하여 표준 경고음을 발생합니다.
Break	**for**, **while** 또는 **repeat** 문을 빠져나갑니다.
ByteToCharIndex	문자열에 지정된 바이트를 포함하는 문자의 위치를 반환합니다.
Chr	지정된 값에 해당하는 아스키 문자를 반환합니다.
Close	파일 변수와 외부 파일 사이의 연결을 종료합니다.

표 8.3 기타 표준 루틴 (계속)

프로시저 또는 함수	설명
CompareMem	두 메모리 이미지의 바이너리 비교를 수행합니다.
CompareStr	대소문자를 구별하여 문자열을 비교합니다.
CompareText	순서값에 의해 문자열을 비교합니다. 대소문자는 구별하지 않습니다.
Continue	**for**, **while** 또는 **repeat** 문의 다음 반복에 대한 제어를 반환합니다.
Copy	문자열의 부분 문자열이나 동적 배열의 세그먼트를 반환합니다.
Cos	각도의 코사인을 계산합니다.
CurrToStr	통화 변수를 문자열로 변환합니다.
Date	현재 날짜를 반환합니다.
DateTimeToStr	*TDateTime* 타입의 변수를 문자열로 변환합니다.
DateToStr	*TDateTime* 타입의 변수를 문자열로 변환합니다.
Dec	순서 변수 값을 하나 감소시킵니다.
Dispose	동적 변수에 할당된 메모리를 해제합니다.
ExceptAddr	현재 예외가 발생한 주소를 반환합니다.
Exit	현재 프로시저를 빠져나갑니다.
Exp	X의 지수 값을 계산합니다.
FillChar	인접한 바이트를 지정된 값으로 채웁니다.
Finalize	동적으로 할당된 변수의 초기화를 해제합니다.
FloatToStr	부동 소수점 값을 문자열로 변환합니다.
FloatToStrF	지정된 서식을 사용하여 부동 소수점 값을 문자열로 변환합니다.
FmtLoadStr	리소스 서식 문자열을 사용하여 서식화된 출력을 반환합니다.
FmtStr	일련의 배열에서 서식화된 문자열을 모읍니다.
Format	서식 문자열 및 일련의 배열에서 문자열을 모읍니다.
FormatDateTime	날짜 및 시간 값을 서식화합니다.
FormatFloat	부동 소수점 값을 서식화합니다.
FreeMem	동적 변수를 해제합니다.
GetMem	블록 주소에 대한 포인터와 동적 변수를 만듭니다.
GetParentForm	지정된 컨트롤이 있는 폼 또는 속성 페이지를 반환합니다.
Halt	프로그램의 비정상적인 종료를 초기화합니다.
Hi	표현식의 상위 바이트를 부호없는 값으로 반환합니다.
High	타입, 배열 또는 문자열의 범위에서 최상위 값을 반환합니다.
Inc	순서 변수 값을 하나 증가시킵니다.
Initialize	동적으로 할당된 변수를 초기화합니다.
Insert	문자열의 지정된 곳에 부분 문자열을 삽입합니다.
Int	실수의 정수 부분을 반환합니다.
IntToStr	정수를 문자열로 변환합니다.
Length	문자열 또는 배열의 길이를 반환합니다.
Lo	표현식의 하위 바이트를 부호없는 값으로 반환합니다.
Low	타입, 배열 또는 문자열의 범위에서 최하위 값을 반환합니다.
LowerCase	아스키 문자열을 소문자로 변환합니다.

표 8.3 기타 표준 루틴 (계속)

프로시저 또는 함수	설명
MaxIntValue	정수 배열에서 가장 큰 부호있는 값을 반환합니다.
MaxValue	배열에서 가장 큰 부호있는 값을 반환합니다.
MinIntValue	정수 배열에서 가장 작은 부호있는 값을 반환합니다.
MinValue	배열에서 가장 작은 부호있는 값을 반환합니다.
New	동적 변수를 새로 만들어 지정된 포인터로 이를 참조합니다.
Now	현재 날짜와 시간을 반환합니다.
Ord	순서 타입 표현식의 순서값을 반환합니다.
Pos	문자열 내에서 지정된 부분 문자열의 첫 번째 문자의 인덱스를 반환합니다.
Pred	순서값의 바로 앞 값을 반환합니다.
Ptr	지정된 주소를 포인터로 변환합니다.
Random	지정된 범위 내에서 임의의 수를 생성합니다.
ReallocMem	동적 변수를 다시 할당합니다.
Round	실수 값을 가장 근접하게 반올림 또는 반내림한 수를 반환합니다.
SetLength	문자열 변수 또는 배열의 동적 길이를 설정합니다.
SetString	특정 문자열의 내용 및 길이를 설정합니다.
ShowException	예외 메시지를 주소와 함께 표시합니다.
ShowMessage	서식화되지 않은 문자열과 OK 버튼이 있는 메시지 상자가 표시됩니다.
ShowMessageFmt	서식화된 문자열과 OK 버튼이 있는 메시지 상자가 표시됩니다.
Sin	라디안 각도의 사인 값을 반환합니다.
SizeOf	변수 또는 타입이 차지하는 바이트 수를 반환합니다.
Sqr	숫자의 제곱을 반환합니다.
Sqrt	숫자의 제곱근을 반환합니다.
Str	문자열을 서식화하여 변수에 반환합니다.
StrToCurr	문자열을 통화값으로 변환합니다.
StrToDate	문자열을 날짜 서식(*TDateTime*)으로 변환합니다.
StrToDateTime	문자열을 *TDateTime*으로 변환합니다.
StrToFloat	문자열을 부동 소수점 값으로 변환합니다.
StrToInt	문자열을 정수로 변환합니다.
StrToTime	문자열을 시간 서식(*TDateTime*)으로 변환합니다.
StrUpper	문자열을 대문자로 반환합니다.
Succ	순서값의 다음 값을 반환합니다.
Sum	배열의 요소 합을 반환합니다.
Time	현재 시간을 반환합니다.
TimeToStr	*TDateTime* 타입의 변수를 문자열로 변환합니다.
Trunc	실수를 잘라내어 정수로 만듭니다.
UniqueString	문자열에 참조가 하나만 있도록 합니다. (문자열을 복사하여 단일 참조를 만들 수도 있습니다.)
UpCase	문자를 대문자로 변환합니다.
UpperCase	문자열을 대문자로 반환합니다.

표 8.3 기타 표준 루틴 (계속)

프로시저 또는 함수	설명
VarArrayCreate	가변 타입 배열을 만듭니다.
VarArrayDimCount	가변 타입 배열의 차원 수를 반환합니다.
VarARrayHighBound	가변 타입 배열에서 차원의 높은 경계를 반환합니다.
VarArrayLock	가변 타입 배열을 잠그고 데이터에 대한 포인터를 반환합니다.
VarArrayLowBound	가변 타입 배열에서 차원의 낮은 경계를 반환합니다.
VarArrayOf	일차원 가변 타입 배열을 만들고 채웁니다.
VarArrayRedim	가변 타입 배열의 크기를 다시 조정합니다.
VarArrayRef	전달된 가변 타입 배열에 대한 참조를 반환합니다.
VarArrayUnlock	가변 타입 배열의 잠금을 해제합니다.
VarAsType	가변을 지정된 타입으로 변환합니다.
VarCast	가변 타입을 지정된 타입으로 변환하고 결과를 변수에 저장합니다.
VarClear	가변 타입을 지웁니다.
VarCopy	가변 타입을 복사합니다.
VarToStr	가변 타입을 문자열로 변환합니다.
VarType	지정된 가변 타입의 타입 코드를 반환합니다.

서식 문자열에 대한 자세한 내용은 온라인 도움말의 "Format strings"를 참조하십시오.

II

전문적인 기능

2부에서는 전문적인 랭귀지 기능과 고급 기능에 관해 다룹니다. 각 장에는 다음과 같은 내용이 포함되어 있습니다.

- 9장 "라이브러리 및 패키지"
- 10장 "객체 인터페이스"
- 11장 "메모리 관리"
- 12장 "프로그램 제어"
- 13장 "인라인 어셈블러 코드"

9

라이브러리 및 패키지

동적으로 로드할 수 있는 라이브러리는 Windows의 동적 연결 라이브러리(DLL) 또는 Linux의 공유 객체 라이브러리(shared object library) 파일입니다. 애플리케이션 및 기타 DLL 또는 공유 객체가 호출할 수 있는 루틴 컬렉션입니다. 유닛처럼, 동적으로 로드할 수 있는 라이브러리에는 공유할 수 있는 코드 또는 리소스가 포함됩니다. 그러나 이런 형식의 라이브러리는 별도로 컴파일되는 실행 파일로서 라이브러리를 사용하는 프로그램과 런타임 시 연결됩니다.

독립 실행 파일과 구별하려면 컴파일된 DLL을 포함하는 Windows 파일 확장자를 .DLL로 지정합니다. Linux에서 공유 객체 파일을 포함하는 파일 확장자를 .so로 지정합니다. 오브젝트 파스칼 프로그램은 DLL 또는 다른 랭귀지로 작성된 공유 객체를 호출할 수 있습니다. 다른 랭귀지로 작성된 애플리케이션은 DLL 또는 오브젝트 파스칼로 작성된 공유 객체를 호출할 수 있습니다.

동적으로 로드할 수 있는 라이브러리 호출

운영체제 루틴을 직접 호출할 수 있으나, 런타임에만 애플리케이션에 연결됩니다. 즉, 프로그램 컴파일 시에는 라이브러리가 없어도 됩니다. 루틴을 import하려는 시도에 대한 컴파일 타임 확인도 없습니다.

공유 객체에 정의된 루틴을 호출하기 전에 반드시 루틴을 *import*해야 합니다. 다음 두 가지 방법으로 루틴을 import할 수 있습니다. 하나는 **external** 프로시저 또는 함수 선언이고 다른 하나는 운영체제 직접 호출입니다. 어느 방법을 사용하든지 간에 루틴은 런타임에만 애플리케이션에 연결됩니다.

오브젝트 파스칼은 공유 라이브러리에서 변수 import를 지원하지 않습니다.

정적 로딩 (Static loading)

프로시저 또는 함수를 가져오는 가장 간단한 방법은 **external** 지시어를 사용하여 프로시저나 함수를 선언하는 것입니다. 예를 들면, 다음과 같습니다.

```
Windows 인 경우 : procedure DoSomething; external 'MYLIB.DLL';
Linux 인 경우 :   procedure DoSomething; external 'mylib.so';
```

이 선언이 프로그램에 포함되어 있으면 프로그램이 시작될 때 MYLIB.DLL (Windows) 또는 mylib.so (Linux)가 로드됩니다. 프로그램 실행 동안 식별자 *DoSomething*은 같은 공유 라이브러리에서 항상 같은 엔트리 포인트을 참조합니다.

가져온 루틴의 선언은 이를 호출한 프로그램이나 유닛에 직접 둘 수 있습니다. 그러나 간단하게 유지할 수 있게 하려면 external 선언을 라이브러리와 인터페이스하는 데 필요한 모든 상수와 타입이 포함된 별도의 "import 유닛"으로 모을 수 있습니다. import 유닛을 사용하는 다른 모듈은 그 유닛에서 선언된 모든 루틴을 호출할 수 있습니다.

external 선언에 대한 자세한 내용은 6-6 페이지의 "외부 선언"을 참조하십시오.

동적 로딩 (Dynamic loading)

LoadLibrary, *FreeLibrary* 및 *GetProcAddress*와 같은 OS 라이브러리 함수를 직접 호출하여 라이브러리에 있는 루틴에 액세스할 수 있습니다. Windows에서 이러한 함수는 Windows.pas에서 선언되고 Linux에서는 호환성을 위해 SysUtils.pas에서 구현됩니다. 실제 Linux OS 루틴은 *dlopen*, *dlclose* 및 *dlsym*이며 Kylix의 *Libc* 유닛에서 선언됩니다. 자세한 내용은 man 페이지를 참조하십시오. 이 경우 import된 루틴을 참조하려면 절차 타입 변수를 사용하십시오.

예를 들어, Windows 또는 Linux에서 다음과 같이 작성했다고 가정해 보십시오.

```
uses Windows, ...; {On Linux, replace Windows with SysUtils }

type
  TTimeRec = record
    Second:Integer;
    Minute:Integer;
    Hour:Integer;
  end;

  TGetTime = procedure(var Time:TTimeRec);
  THandle = Integer;

var
  Time:TTimeRec;
  Handle:THandle;
  GetTime:TGetTime;
  ⋮
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;
```

이런 방식으로 루틴을 import하면 *LoadLibrary* 호출이 포함된 코드가 실행될 때까지는 라이브러리가 로드되지 않습니다. 라이브러리는 나중에 *FreeLibrary* 호출로 언로드됩니다. 이렇게 하면 메모리를 보존할 수 있고 일부 라이브러리가 존재하지 않아도 프로그램을 실행할 수 있습니다.

이와 똑같은 예제를 Linux에서도 다음과 같이 작성할 수 있습니다.

```
uses Libc, ...;

type
  TTimeRec = record
    Second:Integer;
    Minute:Integer;
    Hour:Integer;
  end;

  TGetTime = procedure(var Time:TTimeRec);
  THandle = Pointer;

var
  Time:TTimeRec;
  Handle:THandle;
  GetTime:TGetTime;
  ⋮
begin
  Handle := dlopen('datetime.so', RTLD_LAZY);
  if Handle <> 0 then
  begin
    @GetTime := dlsym(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    dlclose(Handle);
  end;
end;
```

이 경우 루틴을 import하면 *dlopen* 호출이 포함된 코드가 실행될 때까지는 공유 객체가 로드되지 않습니다. 공유 객체는 나중에 *dlclose* 호출로 언로드됩니다. 이렇게 하면 메모리를 보존할 수 있고 일부 공유 객체가 존재하지 않아도 프로그램을 실행할 수 있습니다.

동적으로 로드할 수 있는 라이브러리 작성

program대신 **library** 라는 예약어로 시작한다는 것을 제외하면 동적으로 로드할 수 있는 라이브러리의 메인 소스는 프로그램의 메인 소스와 동일합니다.

라이브러리가 명시적으로 export한 루틴만이 다른 라이브러리나 프로그램에서 import하는 데 사용될 수 있습니다. 다음 예제는 *Min* 및 *Max*라는 두 개의 export된 함수가 있는 라이브러리를 보여줍니다.

```
library MinMax;

function Min(X, Y:Integer):Integer; stdcall;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y:Integer):Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min,
  Max;

begin
end.
```

라이브러리를 다른 랭귀지로 작성한 애플리케이션에서 사용하려면 export된 함수의 선언에서 **stdcall**을 지정하는 것이 가장 안전합니다. 다른 랭귀지에서는 오브젝트 파스칼의 기본 **register** 호출 규칙을 지원하지 않을 수도 있습니다.

라이브러리는 여러 유닛에서 만들 수 있습니다. 이 경우 종종 **uses** 절, **exports** 절 및 초기화 코드로 라이브러리 소스 파일의 코드를 줄일 수 있습니다. 예를 들면, 다음과 같습니다.

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors,
  DoneEditors name Done,
  InsertText name Insert,
  DeleteSelection name Delete,
  FormatSelection,
  PrintSelection name Print,
   :
  SetErrorHandler;

begin
  InitLibrary;
end.
```

유닛의 **interface**나 **implementation** 섹션에 **exports** 절을 둘 수 있습니다. **uses** 절에 해당 유닛이 포함된 라이브러리는 자체에 **exports** 절 없이도 유닛의 **exports** 절에 나열된 루틴을 자동으로 export합니다.

export할 수 없는 루틴을 표시하는 지시어 **local**은 플랫폼에 따라 다르며, Windows 프로그래밍에는 아무런 영향을 미치지 않습니다.

Linux에서 **local** 지시어는 라이브러리에 컴파일되었으나 export되지 않은 루틴에 대한 성능 최적화를 제공하기도 합니다. 이 지시어는 독립 프로시저 및 함수에 대해서는 지정할 수 있으나 메소드에 대해서는 지정할 수 없습니다. **local**에서 선언된 루틴은 다음 예제에서 처럼,

```
function Contraband(I:Integer): Integer; local;
```

EBX 레지스터를 새로 고치지 않습니다. 따라서

- 라이브러리에서 export할 수 없습니다.
- 유닛의 **interface** 섹션에서 선언할 수 없습니다.
- 주소를 가질 수 없으며 절차 타입 변수에 지정될 수 없습니다.
- 순수 어셈블러 루틴인 경우 *호출자*가 EBX를 설정하지 않는 한 다른 유닛에서 호출할 수 없습니다.

exports 절

루틴은 다음과 같은 형태를 갖는 **exports** 절에 나열될 때 export할 수 있습니다.

```
exports entry1, ..., entryn;
```

여기서 각 *entry*는 반드시 **exports** 절보다 먼저 선언해야 하는 프로시저 또는 함수 이름으로 구성되며 루틴이 오버로드된 경우에는 매개변수 목록 앞에 옵니다. **name** 지정자는 옵션입니다. 프로시저 또는 함수 이름을 유닛의 이름으로 제한할 수 있습니다.

(역 호환성을 위해 필요하지만 컴파일러가 무시하는 지시어 **resident**를 entry에 포함할 수도 있습니다.)

Windows에서 **index** 지정자는 지시어 **index**로 구성되며 그 뒤에는 1과 2, 147, 483, 647 사이의 숫자 상수가 나옵니다. 효율적인 프로그램을 위해서는 작은 인덱스 값을 사용하십시오. entry에 **index** 지정자가 없으면 export 테이블에 있는 숫자에 자동으로 루틴이 할당됩니다.

참고 역 호환성만을 지원하는 **index** 지정자의 사용은 다른 개발 툴에 대해서 문제를 야기할 수 있습니다.

name 지정자는 지시어 **name**으로 구성되며 문자열 상수가 뒤에 나옵니다. 엔트리에 **name** 지정자가 없으면 루틴은 원래 선언된 이름과 같은 철자와 대소문자로 export합니다. 다른 이름으로 루틴을 내보내려면 **name** 절을 사용하십시오. 예를 들면, 다음과 같습니다.

```
exports
  DoSomethingABC name 'DoSomething';
```

오버로드된 함수나 프로시저를 동적으로 로드할 수 있는 라이브러리에서 export할 때 **exports** 절에서 매개변수 목록을 지정해야 합니다. 예를 들면, 다음과 같습니다.

```
exports
  Divide(X, Y: Integer) name 'Divide_Ints',
  Divide(X, Y: Real) name 'Divide_Reals';
```

Windows에서 오버로드된 루틴의 entry에 **index** 지정자를 포함시키지 마십시오.

exports 절은 프로그램 또는 라이브러리의 선언 부분이나 유닛의 **interface** 또는 **implementation** 섹션에서 여러 번 나타날 수 있습니다. 프로그램에는 **exports** 절을 거의 포함하지 않습니다.

라이브러리 초기화 코드

라이브러리의 블록에 있는 문은 라이브러리의 *초기화 코*드를 구성합니다. 이러한 문장은 라이브러리가 로드될 때마다 한 번 실행됩니다. 또한 window 클래스 등록 및 변수 초기화 같은 작업을 주로 수행합니다. 라이브러리 초기화 코드는 12-4 페이지의 "종료 프로시저"에 설명하는 것처럼 *ExitProc* 변수를 사용하여 종료 프로시저를 설치할 수도 있습니다. 종료 프로시저는 라이브러리가 언로드될 때 실행됩니다.

라이브러리 초기화 코드는 *ExitCode* 변수를 0이 아닌 값으로 설정하여 오류 신호를 낼 수 있습니다. *ExitCode*는 *시스템* 유닛에서 선언되며 기본값을 0으로 설정하여 성공한 초기화를 나타냅니다. 라이브러리의 초기화 코드가 *ExitCode*를 다른 값으로 설정하면 라이브러리는 언로드되며 호출 애플리케이션에는 실패가 통지됩니다. 마찬가지로 초기화 코드의 실행 동안 처리할 수 없는 예외가 발생하면 호출 애플리케이션에는 라이브러리 로드 실패가 통지됩니다.

다음은 초기화 코드와 종료 프로시저가 있는 라이브러리의 예제입니다.

```
library Test;

var
  SaveExit:Pointer;

procedure LibExit;
begin
  ⋮ // library exit code
  ExitProc := SaveExit;  // restore exit procedure chain
end;

begin
  ⋮ // library initialization code
  SaveExit := ExitProc;  // save exit procedure chain
  ExitProc := @LibExit;  // install LibExit exit procedure
end.
```

라이브러리가 언로드되면 종료 프로시저는 *ExitProc*가 **nil**이 될 때까지 *ExitProc*에 저장된 주소를 반복적으로 호출하여 실행됩니다. 라이브러리에서 사용하는 모든 유닛의 초기화 부분은 라이브러리의 초기화 코드보다 먼저 실행되며 유닛의 완료 부분은 라이브러리의 종료 프로시저보다 먼저 실행됩니다.

라이브러리의 전역 변수

공유 라이브러리에서 선언한 전역 변수는 오브젝트 파스칼 애플리케이션에서 가져올 수 없습니다.

라이브러리는 동시에 여러 애플리케이션에서 사용할 수 있습니다. 그러나 각 애플리케이션은 전역 변수와 함께 자체 프로세스 공간에 라이브러리 복사본을 가지고 있습니다. 여러 라이브러리나 라이브러리의 여러 인스턴스에서 메모리를 공유하려면 메모리 맵 파일을 사용해야 합니다. 자세한 내용은 사용 중인 관련 시스템 설명서를 참조하십시오.

라이브러리 및 시스템 변수

시스템 유닛에서 선언한 여러 변수는 프로그램 라이브러리에서 특히 중요하게 다루어집니다. 코드가 애플리케이션에서 실행하는지 라이브러리에서 실행하는지를 결정하려면 *IsLibrary*를 사용하십시오. *IsLibrary*는 애플리케이션에서는 항상 *False*이며 라이브러리에서는 항상 *True*입니다. 라이브러리 수명 동안 *HInstance*에는 인스턴스 핸들이 포함됩니다. *CmdLine*은 라이브러리에서 항상 **nil**입니다.

DLLProc 변수를 사용하면 라이브러리는 운영체제의 라이브러리 엔트리 포인트에 대한 호출을 모니터할 수 있습니다. 이러한 기능은 보통 다중 스레드를 지원하는 라이브러리에서만 사용됩니다. *DLLProc*는 Windows 및 Linux 모두에서 사용할 수 있으나, 각기 다르게 사용됩니다. Windows에서 *DLLProc*는 다중 스레드 애플리케이션에서 사용되고 Linux에서는 라이브러리가 언로드되는 시기를 결정하는 데 사용합니다. 모든 종료 행태에 대해서 종료 프로시저가 아니라 완료 섹션을 사용해야 합니다.(3-5 페이지의 "완료 섹션" 참조)

운영체제 호출을 모니터하려면 다음 예제와 같이 단일 정수 매개변수가 있는 콜백 프로시저를 만드십시오.

```
procedure DLLHandler(Reason:Integer};
```

그리고 프로시저 주소를 *DLLProc* 변수에 지정하십시오. 프로시저가 호출되면 다음 값 가운데 하나를 전달합니다.

DLL_PROCESS_DETACH	명백한 종료 또는 *FreeLibrary* 호출이나 Linux의 경우 *dlclose* 호출의 결과로 라이브러리가 호출 프로세스의 주소 공간에서 분리(detach)됨을 나타냅니다.
DLL_THREAD_ATTACH	현재 프로세스가 새로운 스레드를 만든다는 것을 나타냅니다(Windows만 해당).
DLL_THREAD_DETACH	스레드가 명백하게 종료됨을 나타냅니다(Windows만 해당).

Linux에서는 *Libc* 유닛에서 정의됩니다.

프로시저 몸체에서 프로시저로 넘긴 매개변수에 따라 취할 동작을 지정할 수 있습니다.

라이브러리의 예외 및 런타임 오류

동적으로 로드할 수 있는 라이브러리에서 처리할 수 없는 예외가 발생하면 이는 라이브러리에서 호출자로 전달됩니다. 호출 애플리케이션이나 라이브러리가 오브젝트 파스칼로 작성되었다면 예외는 보통 **try...except** 문으로 처리할 수 있습니다. 호출 애플리케이션이나 라이브러리가 다른 언어로 작성되었다면 예외는 예외 코드 $0EEDFACE를 갖는 운영체제 예외로 처리할 수 있습니다. 운영체제 예외 레코드의 *ExceptionInformation* 배열의 첫 번째 항목에는 예외 주소가 포함되며 두 번째 항목에는 오브젝트 파스칼 예외 객체에 대한 참조가 포함됩니다.

일반적으로, 라이브러리에 예외가 반드시 포함되도록 해야 합니다. Windows에는 Delphi 예외가 OS 예외 모델에 매핑되지만, Linux에는 예외 모델이 없습니다.

라이브러리가 *SysUtils* 유닛을 사용하지 않을 경우 예외 지원은 사용할 수 없습니다. 이 경우 라이브러리에 런타임 오류가 발생하면 호출 애플리케이션은 종료합니다. 라이브러리에서는 오브젝트 파스칼 프로그램에서 호출되었는지를 알 수 없기 때문에 애플리케이션의 종료 프로시저를 호출할 수 없으며 애플리케이션은 중지되고 메모리에서 제거됩니다.

공유 메모리 관리자 (Shared-memory manger : Windows만 해당)

Windows에서 DLL이 긴 문자열이나 동적 배열을 매개변수 또는 직접적인 결과나 레코드나 객체에 중첩된 결과와 같은 함수 결과로 전달하는 루틴을 내보내면 DLL과 클라이언트 애플리케이션이나 DLL은 반드시 *ShareMem* 유닛을 사용해야 합니다. 다른 모듈에서 *Dispose* 또는 *FreeMem* 호출에 의해 해제된 메모리를 하나의 애플리케이션이나 DLL에서 *New* 또는 *GetMem*으로 할당하는 경우에도 마찬가지입니다. *ShareMem*은 항상 모든 프로그램 또는 라이브러리 **uses** 절에 나열된 첫 번째 유닛이어야 합니다.

*ShareMem*은 BORLANDMM.DLL 메모리 관리자용 인터페이스 유닛으로, 모듈에서 동적으로 할당할 수 있는 메모리를 공유할 수 있습니다. BORLANDMM.DLL은 반드시 *ShareMem*을 사용하는 애플리케이션과 DLL로 배치되어야 합니다. 애플리케이션이나 DLL에서 *ShareMem*을 사용하면 메모리 관리자가 BORLANDMM.DLL의 메모리 관리자로 교체됩니다.

Linux는 glibc의 *malloc*을 사용하여 공유 메모리를 관리합니다.

패키지

패키지는 애플리케이션이나 IDE 또는 두 가지를 모두 사용하여 특별히 컴파일된 라이브러리입니다. 패키지를 사용하면 소스 코드에 영향을 미치지 않으면서 코드가 상주할 때를 재배열할 수 있습니다. 이러한 것을 *애플리케이션 분할 작업*(partitioning)이라고 부르기도 합니다.

*런타임 패키지*는 사용자가 애플리케이션을 실행할 때 필요한 기능을 제공합니다. *디자인 타임 패키지*는 IDE에 컴포넌트를 설치하고 사용자 지정 컴포넌트에 대한 특별 속성 편집기를 만드는 데 사용합니다. 단일 패키지는 디자인 타임과 런타임 모두에 사용될 수 있습니다. 디자인 타임 패키지는 종종 **requires** 절의 런타임 패키지를 참조하여 작동합니다.

이 패키지들을 다른 라이브러리와 구별하기 위해 다음과 같은 파일에 저장합니다.

- Windows에서 패키지 파일은 확장자 .bpl (Borland package library)로 끝납니다.
- Linux에서 패키지는 일반적으로 접두사 bpl로 시작하고 확장자 .so를 갖습니다.

보통 패키지는 애플리케이션이 시작되면 정적으로 로드됩니다. 그러나 *SysUtils* 유닛에서 *LoadPackage* 및 *UnloadPackage* 루틴을 사용하여 동적으로 패키지를 로드할 수 있습니다.

참고 애플리케이션에서 패키지를 사용할 경우, 각 패키지 유닛의 이름은 이를 참조하는 모든 소스 파일의 **uses** 절에 나타나야 합니다. 패키지에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

패키지 선언 및 소스 파일

각 패키지는 별도의 소스 파일에서 선언되며, 오브젝트 파스칼 코드가 포함된 다른 파일과의 혼돈을 피하기 위해 확장자 .dpk 로 저장해야 합니다. 패키지 소스 파일에는 형식, 데이터, 프로시저 또는 함수 선언이 들어 있지 않습니다. 그 대신 다음 내용이 들어 있습니다.

- 패키지 *이름*.
- 새 패키지에서 *필요로 하는* 다른 패키지의 목록. 이 패키지들은 새 패키지와 연결되어 있습니다.
- 컴파일될 때 패키지에 *포함*되거나 바인딩되는 유닛 파일의 목록. 패키지는 본래 컴파일된 패키지의 기능을 제공하는 소스 코드 유닛에 대한 랩퍼입니다.

패키지 선언은 다음과 같은 형태를 가집니다.

```
package packageName;
  requiresClause;
  containsClause;
end.
```

여기서 *packageName*은 유효한 식별자입니다. *requiresClause* 및 *containsClause*는 모두 옵션입니다. 예를 들어, 다음 코드는 *DATAX* 패키지를 선언합니다.

```
package DATAX;
requires
  baseclx,
  visualclx;
  contains Db, DBLocal, DBXpress, ... ;
end.
```

requires 절은 선언된 패키지에서 사용하는 기타 외부 패키지를 나열합니다. 지시어 **requires**로 구성되며 그 뒤에는 콤마로 구분되는 패키지 이름 목록이 오며 그 뒤에 세미콜론이 옵니다. 패키지가 다른 패키지를 참조하지 않으면 패키지에는 **requires** 절이 필요하지 않습니다.

contains 절은 패키지로 컴파일되고 바인딩되는 유닛 파일을 식별합니다. contains 절은 지시어 **contains**로 구성되며 그 뒤에는 콤마로 구분되는 유닛 이름 목록이 오며 그 뒤에 세미콜론이 옵니다. 모든 유닛 이름 다음에는 예약어 **in**이 오며 디렉토리 경로가 있거나 없는 소스 파일 이름은 작은 따옴표 안에 옵니다. 디렉토리 경로는 절대 경로이거나 상대 경로일 수 있습니다. 예를 들면, 다음과 같습니다.

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';
```

참고 패키지를 사용하는 클라이언트는 패키지 유닛에서 **threadvar**로 선언된 스레드 로컬 변수에 액세스할 수 없습니다.

패키지 이름 지정

컴파일된 패키지에는 여러 생성 파일이 있습니다. 예를 들어, *DATAX*라는 패키지에 대한 소스 파일은 DATAX.dpk이고 여기서 컴파일러는 다음과 같은 실행 파일과 바이너리 이미지를 생성합니다.

- Windows인 경우: DATAX.bpl 및 DATAX.dcp
- Linux인 경우: bplDATAX.so (Linux) 및 DATAX.dcp.

*DATAX*는 다른 패키지의 **requires** 절에 있는 패키지를 참조하거나 애플리케이션에 있는 패키지를 사용할 때 사용합니다. 패키지 이름은 프로젝트 내에서 고유해야 합니다.

requires 절

requires 절에는 현재 패키지에서 사용하는 기타 외부 패키지가 나열되어 있습니다. 유닛 파일의 **uses** 절과 같은 기능을 합니다. **requires** 절에 나열된 외부 패키지는 외부 패키지에 포함된 유닛 가운데 하나와 현재 패키지를 모두 사용하는 모든 애플리케이션에 자동으로 컴파일 될 때 연결됩니다.

패키지에 포함된 유닛 파일이 다른 패키지 유닛을 참조하면 다른 패키지는 첫 번째 패키지의 **requires** 절에 포함되어 있어야 합니다. 다른 패키지가 **requires** 절에 생략되면 컴파일러는 .dcu (Windows) 또는 .dpu (Linux) 파일에서 참조된 유닛을 로드합니다.

순환 패키지 참조 피하기

패키지는 **requires** 절에 순환 참조를 포함할 수 없습니다. 이것은 다음을 의미합니다.

- 패키지는 자신의 **requires** 절에 자신을 참조할 수 없습니다.
- 참조 체인은 해당 체인에 있는 패키지를 참조하지 않고 종료되어야 합니다. 패키지 *A*가 패키지 *B*를 요청하면 패키지 *B*는 패키지 *A*를 요청할 수 없습니다. 패키지 *A*가 패키지 *B*를 요청하고 패키지 *B*가 패키지 *C*를 요청하면 패키지 *C*는 패키지 *A*를 요청할 수 없습니다.

중복 패키지 참조

컴파일러는 패키지의 **requires** 절에 있는 중복 참조를 무시합니다. 하지만 프로그래밍 명확성과 가독성을 위해 중복 복제를 제거해야 합니다.

contains 절

contains 절은 패키지에 포함될 유닛 파일을 식별합니다. **contains** 절에 파일 이름 확장자를 포함시키지 마십시오.

불필요한 소스 코드 uses 사용 피하기

패키지는 다른 패키지의 **contains** 절이나 유닛의 **uses** 절에 나열할 수 없습니다.

패키지의 **contains** 절에 직접 포함되거나 그러한 유닛의 **uses** 절에 간접 포함된 모든 유닛은 컴파일 시 패키지에 포함됩니다. 패키지에 직간접적으로 포함된 유닛은 해당 패키지의 **requires** 절에서 참조한 다른 패키지에 포함될 수 없습니다.

유닛은 같은 애플리케이션에서 사용한 둘 이상의 패키지에 직간접적으로 포함될 수 없습니다.

패키지 컴파일

패키지는 보통 패키지 편집기를 사용하여 생성한 .dpk 파일을 사용하여 IDE에서 컴파일됩니다. 명령줄에서 직접 .dpk 파일을 컴파일할 수도 있습니다. 패키지가 포함된 프로젝트를 만들 때 필요에 따라 패키지가 암시적으로 재컴파일됩니다.

생성되는 파일

다음 표에는 성공적인 패키지 컴파일에 의해 생성되는 파일이 나열되어 있습니다.

표 9.1 컴파일된 패키지 파일

파일 확장자	내용
dcp	패키지 헤더에 포함된 바이너리 이미지 및 패키지에 있는 모든 dcu (Windows) 또는 dpu (Linux) 파일의 연결입니다. 각 패키지마다 dcp 파일이 하나 생성됩니다. dcp에 대한 기본 이름은 dpk 소스 파일의 기본 이름입니다.
dcu (Windows) dpu (Linux)	패키지에 포함된 유닛 파일에 대한 바이너리 이미지입니다. 각 유닛 파일에 대해 필요에 따라 dcu 또는 dpu 파일이 하나 생성됩니다.
Windows인 경우 .bpl Linux인 경우 bpl<package>.so	런타임 패키지입니다. 이 파일은 특별한 Borland 특정 기능을 가진 공유 라이브러리입니다. 패키지에 대한 기본 이름은 dpk 소스 파일의 기본 이름입니다.

여러 컴파일러 지시어와 명령줄 스위치는 패키지 컴파일을 지원하는 데 사용할 수 있습니다.

패키지 특정 컴파일러 지시어

다음 표에는 소스 코드에 삽입할 수 있는 패키지 특정 컴파일러 지시어가 나열되어 있습니다. 자세한 내용은 온라인 도움말을 참조하십시오.

표 9.2 패키지 특정 컴파일러 지시어

지시어	용도
{$IMPLICITBUILD OFF}	나중에 패키지가 암시적으로 다시 컴파일되지 않게 합니다. 저수준 기능을 제공하며 빌드 사이에서 가끔씩 변경되거나, 소스 코드가 배포되지 않는 패키지를 컴파일할 때는 .dpk 파일에서 사용하십시오.
{$G-} 또는 {$IMPORTEDDATA OFF}	가져온 데이터 참조를 만들 수 없게 합니다. 이 지시어는 메모리 액세스 효율은 증가시키지만 유닛이 다른 패키지의 변수를 참조하지 못하게 합니다.
{$WEAKPACKAGEUNIT ON}	온라인 도움말에 설명된 것처럼 유닛을 "약하게" 패키지로 만듭니다.
{$DENYPACKAGEUNIT ON}	유닛을 패키지에 두지 못하게 합니다.

표 9.2 패키지 특정 컴파일러 지시어 (continued)

지시어	용도
{$DESIGNONLY ON}	IDE에 설치하기 위해 패키지를 컴파일합니다.(.dpk 파일 안에 둡니다.)
{$RUNONLY ON}	런 타임에만 패키지를 컴파일합니다.(.dpk 파일 안에 둡니다.)

소스 코드에 {$DENYPACKAGEUNIT ON}을 포함시키면 유닛 파일이 패키지로 만들어지지 않습니다. {$G-} 또는 {IMPORTEDDATA OFF}를 포함시키면 패키지를 다른 패키지와 함께 같은 애플리케이션에서 사용할 수 없습니다.

패키지 소스 코드에 적절한 다른 컴파일러 지시어를 포함시킬 수 있습니다.

패키지 특정 명령줄 컴파일러 스위치

다음 패키지 특정 스위치는 명령줄 컴파일러에서 사용할 수 있습니다. 자세한 내용은 온라인 도움말을 참조하십시오.

표 9.3 패키지 특정 명령줄 컴파일러 스위치

스위치	용도
-$G-	가져온 데이터 참조를 만들 수 없게 합니다. 이 스위치를 사용하면 메모리 액세스 효율은 증가하지만 컴파일된 패키지가 다른 패키지에 있는 변수를 참조할 수 없습니다.
-LE *path*	컴파일된 패키지 파일을 둘 디렉토리를 지정합니다.
-LN *path*	패키지 dcp 파일을 둘 디렉토리를 지정합니다.
-LU*packageName [;packageName2;...]*	애플리케이션에서 사용할 추가 런타임 패키지를 지정합니다. 프로젝트를 컴파일할 때 사용합니다.
-Z	나중에 패키지가 암시적으로 다시 컴파일되지 않게 합니다. 저수준 기능을 제공하며 빌드 사이에서 가끔씩 변경되며 소스 코드가 배포되지 않는 패키지를 컴파일할 때 사용하십시오.

-$G- 스위치를 사용하면 패키지를 다른 패키지와 함께 같은 애플리케이션에서 사용할 수 없습니다.

패키지를 컴파일할 때 적절한 다른 명령줄 옵션을 사용할 수도 있습니다.

10

객체 인터페이스

객체(object) 인터페이스 또는 간략하게 *인터페이스*는 클래스에서 구현할 수 있는 메소드를 정의합니다. 인터페이스는 클래스처럼 선언되지만 직접 인스턴스화될 수 없으며 자체의 메소드 정의를 갖지 않습니다. 그보다는 인터페이스의 메소드 구현을 제공하는 인터페이스를 지원하는 것이 클래스가 하는 역할입니다. 인터페이스 타입 변수는 해당 인터페이스를 구현하는 클래스의 객체를 참조하지만, 인터페이스에 선언된 메소드만 이러한 변수를 사용하여 호출될 수 있습니다.

인터페이스는 의미상의 어려움 없이 다중 상속의 장점을 가집니다. 인터페이스는 분산 객체 모델을 사용하는 데에도 필수적입니다. 인터페이스를 지원하는 사용자 지정 객체는 C++, Java 및 기타 랭귀지로 작성된 객체와 상호 작용할 수 있습니다.

인터페이스 타입

클래스처럼 인터페이스는 프로시저나 함수 선언이 아닌 프로그램이나 유닛의 가장 외부 유효 범위(scope)에서만 선언될 수 있습니다. 인터페이스 타입 선언은 다음과 같은 형태를 가집니다.

```
type interfaceName = interface (ancestorInterface)
  ['{GUID}']
  memberList
end;
```

여기서 (*ancestorInterface*)와 ['{*GUID*}']는 옵션입니다. 대부분의 경우 인터페이스 선언은 클래스 선언과 유사하지만 다음과 같은 제한 사항이 적용됩니다.

- *memberList*는 메소드와 속성만 포함할 수 있습니다. 필드는 인터페이ㄴ스에서 사용할 수 없습니다.
- 인터페이스는 필드를 갖지 않으므로 속성 **read** 및 **write** 지정자는 반드시 메소드여야 합니다.
- 인터페이스의 모든 멤버는 public입니다. 가시성(visibility) 지정자 및 저장소 지정자는 사용할 수 없습니다. 그러나 배열 속성은 **default**로 선언될 수 있습니다.

- 인터페이스는 생성자나 소멸자를 갖지 않습니다. 인터페이스는 메소드를 구현하는 클래스가 없으면 인스턴스화될 수 없습니다.
- 메소드는 **virtual**, **dynamic**, **abstract** 또는 **override**로 선언될 수 없습니다. 인터페이스가 자신의 메소드를 구현하지 않기 때문에 이러한 지정은 아무런 의미가 없습니다.

다음은 인터페이스 선언의 예제입니다.

```
type
  IMalloc = interface(IInterface)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size:Integer):Pointer; stdcall;
    function Realloc(P: Pointer; Size:Integer):Pointer; stdcall;
    procedure Free(P:Pointer}; stdcall;
    function GetSize(P:Pointer):Integer; stdcall;
    function DidAlloc(P:Pointer):Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

일부 인터페이스 선언에서는 **interface** 예약어를 **dispinterface**로 대체하기도 합니다. **dispid**, **readonly** 및 **writeonly** 지시어를 사용하는 이러한 구문은 플랫폼에 특정한 것으로 Linux 프로그래밍에서는 사용하지 않습니다.

IInterface 및 상속

인터페이스는 클래스처럼 조상의 메소드를 모두 상속받습니다. 하지만 인터페이스는 클래스와는 달리 메소드를 *구현*하지는 않습니다. 인터페이스가 상속하는 것은 메소드를 구현하는 *의무* 즉, 인터페이스를 지원하는 모든 클래스에 양도하는 의무입니다.

인터페이스 선언은 조상 인터페이스를 지정할 수 있습니다. 조상을 지정하지 않을 경우, 인터페이스는 *시스템* 유닛에 정의되어 있고 다른 모든 인터페이스의 궁극적인 조상인 *IInterface*의 직계 자손이 됩니다. *IInterface*는 다음 세 가지 메소드를 선언합니다. *QueryInterface*, *_AddRef* 및 *_Release*.

참고 *IInterface*는 *IUnknown과 의미가 같습니다.* 일반적으로 플랫폼과 무관한 애플리케이션에는 *IInterface*를 사용하고 Windows 종속 관계를 포함하는 특정 프로그램에 대해서는 *IUnknown*을 사용합니다.

*QueryInterface*는 객체가 지원하는 다른 인터페이스 사이를 자유롭게 이동할 수 있는 방법을 제공합니다. *_AddRef* 및 *_Release*는 인터페이스 참조에 대한 메모리 수명 관리를 합니다. 이러한 메소드를 구현하는 가장 쉬운 방법은 *시스템* 유닛의 *TInterfacedObject*에서 구현 클래스를 파생시키는 것입니다. 또한 이러한 메소드를 빈 함수로 구현하면 어떤 메소드로도 분배할 수 있습니다. 하지만 COM 객체(Windows만 해당)는 반드시 *_AddRef* 및 *_Release*를 통해 관리해야 합니다.

인터페이스 식별

인터페이스 선언은 멤버 목록 바로 앞에서 대괄호로 묶은 문자열로 표시되는, 전역적으로 고유한 식별자(GUID : Globally Unique Identifier)를 지정할 수 있습니다. 선언에서 GUID 부분은 다음과 같은 형태여야 합니다.

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

여기서 각 *x*는 16진수 숫자(0부터 9 또는 A부터 F까지)입니다. Windows에서는 Type Library 편집기가 자동으로 새 인터페이스의 GUID를 생성합니다. 사용자가 Code Editor에서 *Ctrl+Shift+G*(Linux에서는 *Ctrl+Shift+G* 사용)를 클릭해서 GUID를 만들 수도 있습니다.

GUID는 인터페이스를 고유하게 식별하는 16바이트 바이너리 값입니다. 인터페이스에 GUID가 있으면 인터페이스 쿼리를 사용하여 인터페이스 구현에 대한 참조를 얻을 수 있습니다. 10-10 페이지의 "인터페이스 쿼리"를 참조하십시오.

시스템 유닛에 선언된 *TGUID* 및 *PGUID* 타입은 GUID를 처리하는데 사용됩니다.

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2:Word;
    D3:Word;
    D4:array[0..7] of Byte;
  end;
```

TGUID 타입이 지정된 상수를 선언할 경우 문자열 리터럴을 사용하여 그 값을 지정할 수 있습니다. 예를 들면, 다음과 같습니다.

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

프로시저 및 함수 호출에서 GUID나 인터페이스 식별자는 *TGUID* 타입의 값 또는 상수 매개변수로 사용할 수 있습니다.예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
function Supports(Unknown: IInterface; const IID: TGUID}:Boolean;
```

*Supports*는 다음 중 한 가지 방법으로 호출될 수 있습니다.

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

인터페이스의 호출 규칙

기본 호출 규칙은 **register**이지만, 모듈 간에 공유된 인터페이스 특히, 서로 다른 랭귀지로 작성했을 경우에는 **stdcall**을 사용하여 모든 메소드를 선언해야 합니다. **safecall**을 사용하여 CORBA 인터페이스를 구현합니다. Windows에서는 **safecall**을 사용하면 10-13 페이지의 "이중 인터페이스(Dual interface : Windows만 해당)"에서 설명한 대로 이중 인터페이스의 메소드를 구현할 수 있습니다.

호출 규칙에 대한 자세한 내용은 6-5 페이지의 "호출 규칙"을 참조하십시오.

인터페이스 속성

인터페이스에 선언된 속성은 인터페이스 타입의 표현식을 통해서만 액세스할 수 있으며 클래스 타입 변수를 통해서는 액세스할 수 없습니다. 더구나 인터페이스 속성은 인터페이스가 컴파일된 프로그램 내에서만 가시성이 있습니다. 예를 들어 Windows에서 COM 객체는 속성을 가지지 않습니다.

인터페이스에서 필드를 사용할 수 없기 때문에 속성 **read** 및 **write** 지정자는 반드시 메소드여야 합니다.

Forward 선언

조상, GUID 또는 멤버 목록을 지정하지 않고 **interface** 예약어와 세미콜론으로 끝나는 인터페이스 선언은 *forward 선언*입니다. forward 선언은 반드시 동일한 타입 선언 섹션 안에서 동일한 인터페이스의 *선언 정의*로 해결되어야 합니다. 즉, forward 선언과 forward 선언 정의 사이는 다른 타입 선언 이외에는 올 수 없습니다.

forward 선언은 상호 종속적인 인터페이스를 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index:Integer): IControl;
    ⋮
  end;
  IControl = interface
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
    ⋮
  end;
```

상호 파생된 인터페이스는 사용할 수 없습니다. 예를 들어 *IControl*에서 *IWindow*를 파생시키고 *IWindow*에서 *IControl*을 파생시키는 것은 적합하지 않습니다.

인터페이스 구현

일단 인터페이스가 선언되면 인터페이스가 사용되기 전에 클래스에 인터페이스를 구현해야 합니다. 클래스에 의해 구현된 인터페이스는 클래스 선언에서 클래스의 조상 이름 뒤에 지정됩니다. 이러한 선언문은 다음과 같은 형태를 가집니다.

```
type className = class (ancestorClass, interface1, ..., interfacen)
  memberList
 end;
```

예를 들면, 다음과 같습니다.

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ⋮
  end;
```

IMalloc 및 *IErrorInfo* 인터페이스를 구현하는 *TMemoryManager*라는 클래스를 선언합니다. 클래스가 인터페이스를 구현할 때 클래스는 인터페이스에 선언된 각 메소드를 구현하거나 각 메소드의 구현을 상속해야 합니다.

다음은 *시스템* 유닛의 *TInterfacedObject* 선언입니다.

```
type
TInterfacedObject = class(TObject, IInterface)
protected
    FRefCount:Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef:Integer; stdcall;
    function _Release:Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

*TInterfacedObject*는 *IInterface* 인터페이스를 구현합니다. 그러므로 *TInterfaced Object*는 *IInterface*의 메소드 세 개를 각각 선언하고 구현합니다.

인터페이스를 구현하는 클래스는 기본 클래스로도 사용될 수 있습니다. (위의 첫 번째 예제는 *TMemoryManager*를 *TInterfacedObject*의 직계 자손으로서 선언합니다.) 모든 인터페이스는 *IInterface*로부터 상속되기 때문에 인터페이스를 구현하는 클래스는 *QueryInterface*, *_AddRef* 및 *_Release* 메소드를 구현해야 합니다. *시스템* 유닛의 *TInterfacedObject*는 이러한 메소드를 구현하며, 따라서 이것은 인터페이스를 구현하는 다른 클래스를 파생시키는 편리한 기본 클래스입니다.

인터페이스를 구현할 때 인터페이스의 각 메소드는 각 위치에서 동일한 결과 타입, 동일한 호출 규칙, 동일한 매개변수의 수 및 동일하게 타입이 지정된 매개변수를 갖는 구현 클래스의 메소드에 매핑됩니다. 기본적으로 각 인터페이스 메소드는 구현 클래스에서 동일한 이름의 메소드에 매핑됩니다.

메소드 확인 절

클래스 선언에 *메소드 확인 절*을 포함시켜 기본 이름 기반 매핑을 오버라이드할 수 있습니다. 클래스가 동일하게 명명된 메소드를 갖는 둘 이상의 인터페이스를 구현할 때 메소드 확인 절을 사용하여 이름 충돌을 해결합니다.

메소드 확인 절은 다음과 같은 형태를 가집니다.

```
procedure interface.interfaceMethod = implementingMethod;
```

또는

```
function interface.interfaceMethod = implementingMethod;
```

여기서 *implementingMethod*는 클래스 또는 이 클래스의 조상 클래스에 선언된 메소드입니다. *implementingMethod*는 클래스 선언에서 나중에 선언된 메소드일 수 있지만, 다른 모듈에 선언된 조상 클래스의 private 메소드일 수는 없습니다.

예를 들어 다음과 같은 클래스 선언이 있습니다.

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ⋮
  end;
```

*IMalloc*의 *Alloc*과 *Free* 메소드는 *TMemoryManager*의 *Allocate*과 *Deallocate* 메소드에 매핑됩니다.

메소드 확인 절은 조상 클래스에 의해 도입된 매핑을 대체할 수 없습니다.

상속된 구현 변경

자손 클래스는 구현 메소드를 오버라이드함으로써 특정 인터페이스 메소드가 구현되는 방법을 변경할 수 있습니다. 이렇게 하려면 구현 메소드가 가상이거나 동적이어야 합니다.

클래스는 조상 클래스로부터 상속되는 전체 인터페이스를 재구현할 수도 있습니다. 이것은 자손 클래스의 선언에서 인터페이스를 재열거하는 것과 관련됩니다. 예를 들면, 다음과 같습니다.

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    ⋮
  end;

  TWindow = class(TInterfacedObject, IWindow)  // TWindow implements IWindow
    procedure Draw;
    ⋮
  end;

  TFrameWindow = class(TWindow, IWindow)  // TFrameWindow reimplements IWindow
    procedure Draw;
    ⋮
  end;
```

인터페이스를 재구현하면 동일한 인터페이스에서 상속된 구현을 숨깁니다. 그러므로 조상 클래스의 메소드 확인 절은 재구현된 인터페이스에 아무런 영향을 주지 않습니다.

위임(Delegation)으로 인터페이스 구현

implements 지시어를 사용하여 구현 클래스의 속성에 대한 인터페이스 구현을 위임할 수 있습니다. 예를 들면, 다음과 같습니다.

```
property MyInterface:IMyInterface read FMyInterface implements IMyInterface;
```

IMyInterface 인터페이스를 구현하는 *MyInterface*라고 하는 속성을 선언합니다.

implements 지시어는 속성 선언에서 마지막 지정자여야 하며 쉼표로 구분된 둘 이상의 인터페이스를 열거할 수 있습니다. 위임 속성은 다음과 같아야 합니다.

- 클래스타입이나 인터페이스 타입이어야 합니다.
- 배열 속성이거나 인덱스 지정자를 가질 수 없습니다.
- 반드시 **read** 지정자를 가져야 합니다. 속성이 **read** 메소드를 사용할 경우 해당 메소드는 기본 **register** 호출 규칙을 사용해야 하고 해당 메소드가 가상일 수는 있지만 동적이거나 **message** 지시어를 지정할 수는 없습니다.

참고 위임된 인터페이스를 구현하는 데 사용하는 클래스는 *TAggregatedObject*로부터 파생되어야 합니다.

인터페이스 타입 속성에 위임

위임 속성이 인터페이스 타입일 경우, 해당 인터페이스 또는 해당 인터페이스가 파생된 인터페이스는 해당 속성이 선언된 클래스의 조상 목록에 있어야 합니다. 위임 속성은 **implements** 지시어가 지정한 인터페이스를 메소드 확인 절 없이 완전하게 구현하는 클래스의 객체를 반환해야 합니다. 예를 들면, 다음과 같습니다.

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyClass = class(TObject, IMyInterface)
    FMyInterface:IMyInterface;
    property MyInterface:IMyInterface read FMyInterface implements IMyInterface;
  end;

var
  MyClass:TMyClass;
  MyInterface:IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ...  // some object whose class implements IMyInterface
  MyInterface := MyClass;
  MyInterface.P1;
end;
```

클래스 타입 속성에 위임

위임 속성이 클래스 타입일 경우, 해당 클래스와 그 조상은 상위 클래스와 그 조상을 검색하기 전에 지정된 인터페이스를 구현하는 메소드를 검색합니다. 따라서 속성에 의해 지정된 클래스의 일부 메소드와 속성이 선언된 클래스의 나머지 메소드를 구현하는 것이 가능합니다. 메소드 확인 절은 불확실성을 해결하거나 특정 메소드를 지정하기 위한 일반적인 방법으로 사용될 수 있습니다. 인터페이스는 둘 이상의 클래스 타입 속성에 의해 구현될 수 없습니다. 예를 들면, 다음과 같습니다.

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
  TMyImplClass = class
    procedure P1;
    procedure P2;
```

```
  end;
  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyImplClass: TMyImplClass;
    property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
  end;
procedure TMyImplClass.P1;
  ⋮
procedure TMyImplClass.P2;
  ⋮
procedure TMyClass.MyP1;
  ⋮
var
  MyClass:TMyClass;
  MyInterface:IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;        // calls TMyClass.MyP1;
  MyInterface.P2;        // calls TImplClass.P2;
end;
```

인터페이스 참조

인터페이스 타입 변수를 선언할 경우 해당 변수는 인터페이스를 구현하는 모든 클래스의 인스턴스를 참조할 수 있습니다. 이러한 변수를 사용하여 컴파일 시 해당 인터페이스가 어디에 구현되었는지를 모르더라도 인터페이스 메소드를 호출할 수 있습니다. 그러나 이러한 변수들은 다음과 같은 제한이 따릅니다.

- 인터페이스 타입 표현식은 인터페이스에 선언된 메소드와 속성만 액세스할 수 있으며, 구현 클래스의 다른 멤버는 액세스할 수 없습니다.
- 또한 클래스 또는 상속받은 클래스가 명시적으로 조상 인터페이스를 구현하지 않으면 인터페이스 타입 표현식은 자손 인터페이스를 구현하는 클래스의 객체를 참조할 수 없습니다.

예를 들면, 다음과 같습니다.

```
type
  IAncestor = interface
  end;

  IDescendant = interface(IAncestor)
    procedure P1;
  end;

  TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
  end;
  ⋮
var
```

```
  D: IDescendant;
  A : IAncestor;
begin
  D := TSomething.Create;  // works!
  A := TSomething.Create;  // error
  D.P1;  // works!
  D.P2;  // error
end;
```

이 예제에서,

- *A*는 *IAncestor* 타입의 변수로 선언됩니다. *TSomething*은 구현하는 인터페이스 간의 *IAncestor*를 열거하지 않기 때문에 *TSomething* 인스턴스는 *A*에 지정될 수 없습니다. 그러나 *TSomething*의 선언을 다음과 같이 변경할 경우,

  ```
  TSomething = class(TInterfacedObject, IAncestor, IDescendant)
   ⋮
  ```

  첫 번째 오류는 유효한 할당이 됩니다 .
- *D*는 *IDescendant* 타입의 변수로 선언됩니다. *D*는 *TSomething*의 인스턴스를 참조하는 반면, *P2*는 *IDescendant*의 메소드가 아니기 때문에 이것을 사용하여 *TSomething*의 *P2* 메소드에 액세스할 수 없습니다. 그러나 *D*의 선언을 다음과 같이 변경할 경우,

  ```
  D: TSomething;
  ```

  두 번째 오류는 유효한 메소드 호출이 됩니다 .

인터페이스 참조는 *IInterface*에서 상속된 *_AddRef* 및 *_Release* 메소드에 종속되는 참조 카운팅을 통해 관리됩니다. 객체가 인터페이스를 통해서만 참조될 경우 수동으로 객체를 삭제할 필요가 없습니다. 객체에 대한 마지막 참조가 유효 범위(scope)를 벗어나면 객체는 자동으로 소멸됩니다.

전역 인터페이스 타입 변수는 **nil**로만 초기화될 수 있습니다.

인터페이스 타입 표현식이 객체를 참조할지 여부를 알아보려면 객체를 *Assigned* 표준 함수로 전달하십시오.

인터페이스 할당 호환(Interface assignment-compatibility)

클래스 타입은 클래스에 의해 구현된 어떠한 인터페이스 타입과도 할당 호환됩니다. 인터페이스 타입은 어떠한 조상 인터페이스 타입과도 할당 호환됩니다. **nil** 값은 모든 인터페이스 타입 변수에 할당될 수 있습니다.

인터페이스 타입 표현식은 가변(Varient)에 할당될 수 있습니다. 인터페이스가 *IDispatch* 타입이거나 자손일 경우, 가변은 *varDispatch* 타입 코드를 받습니다. 그렇지 않을 경우 가변은 *varUnknown* 타입 코드를 받습니다.

타입 코드가 *varEmpty*, *varUnknown* 또는 *varDispatch*인 가변은 *IInterface* 변수에 할당될 수 있습니다. 타입 코드가 *varEmpty*이거나 *varDispatch*인 가변은 *IDispatch* 변수에 할당될 수 있습니다.

인터페이스 타입 변환

인터페이스 타입은 변수 및 값 타입 변환에서 클래스 타입과 동일한 규칙을 따릅니다. 클래스 타입 표현식은 클래스가 인터페이스를 구현할 경우 예를 들어 `IMyInterface(SomeObject)` 같은 인터페이스 타입으로 타입 변환 될 수 있습니다.

인터페이스 타입 표현식은 *Variant*로 타입 변환될 수 있습니다. 인터페이스가 *IDispatch* 타입이거나 자손일 경우 결과 가변은 *varDispatch* 타입 코드를 가집니다. 그렇지 않을 경우 결과 가변은 *varUnknown* 타입 코드를 가집니다.

타입 코드가 *varEmpty*, *varUnknown* 또는 *varDispatch*인 가변은 *IInterface*로 타입 변환될 수 있습니다. 타입 코드가 *varEmpty*이거나 *varDispatch*인 가변은 *IDispatch*로 타입 변환될 수 있습니다.

인터페이스 쿼리

as 연산자를 사용하여 확인된 인터페이스 타입 변환을 수행할 수 있습니다. 이것을 *인터페이스 쿼리*라고 하며, 객체의 실제(런타임) 타입에 기반하여 객체 참조나 다른 인터페이스 참조에서 인터페이스 타입 표현식을 계산합니다. 인터페이스 쿼리는 다음과 같은 형태를 가집니다.

object **as** *interface*

여기서 *object*는 인터페이스 타입 또는 가변 타입의 표현식이거나 인터페이스를 구현하는 클래스의 인스턴스를 의미하며, *interface*는 GUID를 사용하여 선언된 인터페이스입니다.

인터페이스 쿼리는 *object*가 **nil**일 경우 **nil**을 반환합니다. 그외에는 인터페이스 쿼리는 *QueryInterface*가 0을 반환하지 않는다는 예외를 발생하고 *interface*의 GUID를 *object*의 *QueryInterface* 메소드로 전달합니다. *QueryInterface*가 0을 반환하여 *object*의 클래스가 *interface*를 구현한다는 것을 나타낼 경우 인터페이스 쿼리는 *object*에 대한 인터페이스 참조를 반환합니다.

Automation 객체(Windows만 해당)

시스템 유닛에 선언된 *IDispatch* 인터페이스를 구현하는 클래스의 객체가 Automation 객체입니다. Automation은 Windows에서만 사용할 수 있습니다.

Dispatch 인터페이스 타입(Windows만 해당)

Dispatch 인터페이스 타입은 Automation 객체가 *IDispatch*를 통해 구현하는 메소드와 속성을 정의합니다. dispatch 인터페이스의 메소드 호출은 런타임 시 *IDispatch*의 *Invoke* 메소드를 통해 라우트되고 클래스는 dispatch 인터페이스를 구현할 수 없습니다.

dispatch 인터페이스 타입 선언은 다음과 같은 형태를 가집니다.

```
type interfaceName = dispinterface
  ['{GUID}']
  memberList
end;
```

여기서 ['{*GUID*}']는 옵션이고 *memberList*는 속성 및 메소드 선언문으로 구성됩니다. Dispatch 인터페이스 선언은 보통의 인터페이스 선언과 유사하지만 dispatch 인터페이스 선언문은 조상을 지정할 수 없습니다. 예를 들면, 다음과 같습니다.

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index:Integer]: OleVariant dispid 0; default;
    function Count:Integer; dispid 1;
    property Item[Index:Integer]: OleVariant dispid 2;
    procedure Remove(Index:Integer}; dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant):Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
  end;
```

Dispatch 인터페이스 메소드 (Windows 만 해당)

dispatch 인터페이스 메소드는 *IDispatch* 구현을 기본으로 하는 *Invoke* 메소드에 대한 호출의 프로토타입입니다. 메소드의 Automation 디스패치 ID를 지정하려면 선언에 정수 상수가 뒤에 오고 이미 사용된 ID가 오류를 발생한다는 것을 지정하는 **dispid** 지시어를 포함시키십시오.

dispatch 인터페이스에 선언된 메소드는 **dispid** 이외의 지시어를 포함할 수 없습니다. 매개변수 및 결과 타입은 반드시 Automation이 가능해야 합니다. 다시 말해서 매개변수 및 결과 타입은 *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool* 또는 인터페이스 타입이어야 합니다.

Dispatch 인터페이스 속성

dispatch 인터페이스의 속성은 액세스 지정자를 포함하지 않습니다. dispatch 인터페이스 속성은 **readonly** 또는 **writeonly**로 선언될 수 있습니다. 속성의 디스패치 ID를 지정하려면 선언에 뒤에 정수 상수를 둔 **dispid** 지시어를 포함시켜서 이미 사용된 ID에서 오류가 발생하도록 지정하십시오. 배열 속성은 **default**로 선언될 수 있습니다. 그 외의 지시어는 dispatch 인터페이스 속성 선언문에 사용될 수 없습니다.

Automation 객체 액세스(Windows만 해당)

Automation 객체에 액세스하려면 가변 타입을 사용합니다. 가변 타입이 Automation 객체를 참조할 경우, 객체의 메소드를 호출할 수 있으며 가변 타입을 통해 객체의 속성을 읽거나 쓸 수 있습니다. 이렇게 하려면 유닛이나 프로그램 또는 라이브러리 중 하나의 **uses** 절에 *ComObj*를 포함해야 합니다.

Automation 객체 메소드 호출은 런타임 시 바인드되고 이전 메소드 선언문을 필요로 하지 않습니다. 이러한 호출의 유효성은 컴파일 시 확인되지 않습니다.

다음 예제는 Automation 메소드 호출을 보여 줍니다. *ComObj*에 정의된 *CreateOleObject* 함수는 Automation 객체에 대한 *IDispatch* 참조를 반환하고 이 함수는 *Word* 가변 타입과 할당 호환됩니다.

```
var
  Word:Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

인터페이스 타입 매개변수를 Automation 메소드에 전달할 수 있습니다.

*varByte*의 요소 타입을 가진 가변 타입 타입 배열은 Automation 컨트롤러와 서버 사이에 바이너리 데이터를 전달하는 기본 설정 메소드입니다. 이러한 배열은 해당 데이터의 변환에 종속되지 않고 *VarArrayLock* 및 *VarArrayUnlock* 루틴을 사용하여 효율적으로 액세스할 수 있습니다.

Automation 객체 메소드 호출 구문

Automation 객체 메소드 호출이나 속성 액세스 구문은 일반적인 메소드 호출이나 속성 액세스 구문과 유사합니다. 하지만 Automation 메소드 호출은 *위치* 및 *명명된* 매개변수를 둘 다 사용할 수 있습니다. (그러나 일부 Automation 서버에서는 명명된 매개변수를 지원하지 않습니다.)

위치 매개변수는 단순히 표현식입니다. 명명된 매개변수 다음에는 **:=** 기호가 있고 그 뒤에 표현식이 오는 매개변수 식별자로 구성됩니다. 위치 매개변수는 메소드 호출에서 명명된 매개변수보다 앞에 와야 합니다. 명명된 매개변수는 어떤 순서로든 지정될 수 있습니다.

일부 Automation 서버를 사용하여 메소드 호출에서 기본 값을 승인하면 매개변수를 생략할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation 메소드 호출 매개변수는 정수 타입, 실수 타입, 문자열 타입, 부울 타입 및 가변 타입일 수 있습니다. 매개변수 표현식이 변수 참조로만 구성되었을 경우, 그리고 변수 참조가 *Byte*, *Smallint*, *Integer*, *Single*, *Double*, *Currency*, *TDateTime*, *AnsiString*, *WordBool*, 또는 *Variant* 타입일 경우 매개변수는 참조에 의해 전달됩니다. 표현식이 이러한 타입 중 하나가 아니거나 변수가 아닐 경우 매개변수는 값에 의해 전달됩니다. 값 매개변수를 예상하는 메소드에 참조에 의해 매개변수를 전달하면 COM이 해당 참조 매개변수에서 값을 페치(fetch)합니다. 참조 매개변수를 예상하는 메소드로 값에 의한 매개변수를 전달하면 오류가 발생합니다.

이중 인터페이스(Dual interface : Windows만 해당)

이중 인터페이스는 Automation을 통해 컴파일 시 바인딩과 런타임 바인딩을 모두 지원하는 인터페이스입니다. 이중 인터페이스는 *IDispatch*의 자손이어야 합니다.

IInterface 및 *IDispatch*로부터 상속된 것을 제외한 모든 이중 인터페이스의 메소드는 **safecall** 규칙을 사용해야 하고, 모든 메소드 매개변수와 결과 타입은 Automation 가능해야 합니다. (Automation 가능 타입은 *Byte*, *Currency*, *Real*, *Double*, *Real48*, *Integer*, *Single*, *Smallint*, *AnsiString*, *ShortString*, *TDateTime*, *Variant*, *OleVariant* 및 *WordBool*입니다.)

11

메모리 관리

이 장에서는 프로그램이 메모리를 사용하는 방법과 오브젝트 파스칼 데이터 타입의 내부 형식에 대해 설명합니다.

메모리 관리자(Windows만 해당)

참고 Linux에서는 메모리 관리를 위해 *malloc* 같은 glibc 함수를 사용합니다. 자세한 내용은 Linux 시스템의 *malloc* man 페이지를 참조하십시오.

Windows 시스템에서는 메모리 관리자가 애플리케이션의 모든 동적 메모리 할당 및 해제를 관리합니다. *New*, *Dispose*, *GetMem*, *ReallocMem* 및 *FreeMem* 표준 프로시저에서는 메모리 관리자를 사용하고 모든 객체와 긴 문자열은 메모리 관리자를 통해 할당됩니다.

Windows에서 메모리 관리자는 중소 타입 블록을 대량으로 할당하는 애플리케이션에 맞게 최적화되어 있으며, 이것은 문자열 데이터를 처리하는 객체 지향 애플리케이션에서 일반적인 방식입니다. *GlobalAlloc*의 구현, *LocalAlloc* 및 Windows에서 전용 힙(heap) 지원과 같은 다른 메모리 관리자는 일반적으로 중소 타입 메모리를 많이 할당하는 프로그램에서는 제대로 실행되지 않으며 만일 직접 사용했을 경우 애플리케이션의 속도가 느려집니다.

성능을 최상으로 올리려면 메모리 관리자가 Win32 가상 메모리 API (*VirtualAlloc* 및 *VirtualFree* 함수)와 직접 인터페이스해야 합니다. 메모리 관리자는 운영 체제의 메모리를 1MB의 주소 공간 섹션에 예약하고 필요하면 메모리를 16KB씩 커밋합니다. 16KB와 1MB 섹션에서 사용되지 않는 메모리를 디커밋하고 릴리스합니다. 더 작은 블록인 경우 커밋된 메모리는 더 작게 할당됩니다.

메모리 관리자 블록은 항상 4바이트로 올림되므로 블록의 크기와 다른 상태 비트가 저장되어 있는 4바이트 헤더가 항상 포함됩니다. 이것은 메모리 관리자 블록이 항상 더블 워드로 정렬되어 있음을 의미하므로 블록을 주소 지정할 때 최적의 CPU 성능을 보장합니다.

메모리 관리자는 현재 할당된 메모리 블록의 수와 현재 할당된 메모리 블록을 모두 합친 크기가 들어 있는 두 개의 상태 변수, *AllocMemCount*와 *AllocMemSize*를 가지고 있습니다. 애플리케이션에서는 이러한 변수를 사용하여 디버깅 시 상태 정보를 표시합니다.

시스템 유닛은 두 개의 프로시저 *GetMemoryManager*와 *SetMemoryManager*를 제공하여 애플리케이션에서 저수준의 메모리 관리자 호출을 인터셉트할 수 있습니다. 또한 *시스템* 유닛은 자세한 메모리 관리자 상태 정보가 들어 있는 레코드를 반환하는 *GetHeapStatus*라는 함수도 제공합니다. 이러한 루틴에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

변수

전역 변수는 애플리케이션 데이터 세그먼트에 할당되어 프로그램이 실행되는 동안 계속 유지됩니다. 프로시저와 함수 내에서 선언된 지역 변수는 애플리케이션의 스택에 저장됩니다. 프로시저나 함수가 호출되면 언제나 지역 변수 집합이 스택에 할당되고, 루틴이 종료되면 할당된 지역 변수는 제거됩니다. 컴파일러 최적화를 수행하면 이전의 변수가 제거됩니다.

참고 Linux에서 스택 크기는 해당 환경에서만 설정됩니다.

Windows에서 애플리케이션의 스택은 *최소 스택 크기*와 *최대 스택 크기* 두 가지 값으로 정의됩니다. 이 값들은 **$MINSTACKSIZE**와 **$MAXSTACKSIZE** 컴파일러 지시문으로 제어되며 기본값은 각각 16,384(16K) 와 1,048,576(1M)입니다. 애플리케이션은 사용 가능한 최소 스택 크기를 갖도록 되어 있으며 애플리케이션의 스택은 최대 스택 크기보다 커질 수 없습니다. 애플리케이션의 최소 스택 요구 사항을 만족하는 메모리가 없는 경우 Windows는 애플리케이션을 시작하려고 할 때 오류를 보고합니다.

Windows 애플리케이션이 최소 스택 크기보다 큰 스택 공간을 필요로 할 경우 추가 메모리는 자동으로 4K 단위로 할당됩니다. 메모리가 부족하거나 전체 스택 크기가 최대 스택 크기를 초과하여 추가 스택 공간 할당에 실패할 경우 *EStackOverflow* 예외가 발생합니다. 스택 오버플로 검사는 자동으로 실행됩니다. 원래 오버플로 검사를 제어하는 **$S** 컴파일러 지시문은 이전 버전과의 호환성을 위해 계속 지원됩니다.

Windows나 Linux에서 *GetMem* 또는 *New* 프로시저로 만든 동적 변수에는 힙이 할당되고 *FreeMem* 또는 *Dispose*를 사용하여 해제할 때까지 계속 유지됩니다.

긴 문자열, 와이드 문자열, 동적 배열, 가변 및 인터페이스는 힙에 할당되지만 이들의 메모리는 자동으로 관리됩니다.

내부 데이터 형식

다음 섹션에서는 오브젝트 파스칼 데이터 타입의 내부 형식에 대해 설명합니다.

정수 타입(Integer types)

정수 타입 변수의 형식은 최소 및 최대 경계에 따라 다릅니다.

- 두 경계가 모두 -128..127(*Shortint*) 범위 내에 있을 경우 변수는 부호있는 바이트로 저장됩니다.
- 두 경계가 모두 0..255(*Byte*) 범위 내에 있을 경우 변수는 부호없는 바이트로 저장됩니다.
- 두 경계가 모두 -32768..32767(*Smallint*) 범위 내에 있을 경우 변수는 부호있는 워드로 저장됩니다.
- 두 경계가 모두 0..65535(*Word*) 범위 내에 있을 경우 변수는 부호없는 워드로 저장됩니다.
- 두 경계가 모두 -2147483648..2147483647(*Longint*) 범위 내에 있을 경우 변수는 부호있는 더블 워드로 저장됩니다.
- 두 경계가 모두 0..4294967295(*Longword*) 범위에 있을 경우 변수는 부호없는 더블 워드로 저장됩니다.
- 그 외의 경우 변수는 부호있는 4배(quadruple)워드(*Int64*)로 저장됩니다.

문자 타입(Character types)

Char, *AnsiChar* 또는 *Char* 타입의 부분범위 타입은 부호없는 바이트로 저장됩니다. *WideChar*는 부호없는 바이트로 저장됩니다.

부울 타입(Boolean types)

Boolean 타입은 *Byte*로 저장되고, *ByteBool*은 *Byte*로 저장되고, *WordBool* 타입은 *Word*로 저장되고 *LongBool*은 *Longint*로 저장됩니다.

*Boolean*은 값 0(*False*) 또는 1(*True*)을 가질 수 있습니다. *ByteBool*, *WordBool* 및 *LongBool* 타입은 값 0(*False*) 또는 0이 아닌 값(*True*)을 가질 수 있습니다.

열거 타입(Enumerated types)

열거 타입의 값이 256개 이하이고, 형식이 기본 설정된 {$Z1} 상태에서 선언되었을 경우, 열거 타입은 부호없는 바이트로 저장됩니다. 열거 타입의 값이 257개 이상의 값을 갖거나 타입이 {$Z2} 상태에서 선언되었을 경우 부호없는 워드로 저장됩니다. 열거 타입이 {$Z4} 상태에서 선언되었을 경우에는 부호를 포함하지 않은 더블 워드로 저장됩니다.

실수 타입(Real types)

실수 타입은 부호(+ 또는 −), *exponent* 및 *significand*를 바이너리 표현으로 저장합니다. 실수값은 다음과 같은 형식을 가집니다.

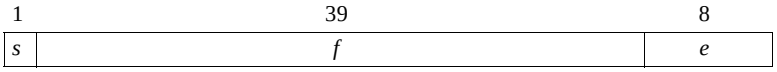
+/- *significand* * $2^{exponent}$

여기서, *significand*는 바이너리 소수점 왼쪽에 1비트를 가집니다. 즉, 0 <= *significand* < 2입니다.

다음 그림에서 최상위 비트는 항상 왼쪽에 있고 최하위 비트는 오른쪽에 있습니다. 다음 그림에서 위에 있는 숫자는 각 필드의 너비(비트 단위)를 나타냅니다. 가장 왼쪽에 있는 항목은 최상위 주소에 저장됩니다. 예를 들어, *Real48* 값에서 *e*는 첫 번째 바이트에 저장되고 *f*는 다음의 5바이트에, *s*는 마지막 바이트의 최상위 비트에 저장됩니다.

Real48 타입 (The Real48 type)

6바이트(48비트)의 *Real48* 숫자는 다음과 같이 세 필드로 나뉩니다.



0 < *e* <= 255일 경우 숫자의 *v* 값은 다음 식으로 구합니다.

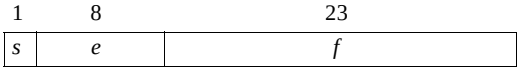
$v = (-1)^s * 2^{(e-129)} * (1.f)$

e = 0이면 *v* = 0입니다.

Real48 타입은 denormal, NaN(Not a Number)및 무한대를 저장할 수 없습니다. NaN과 무한대를 *Real48*에 저장하려고 하면 오버플로 오류가 발생하는 반면, denormal을 *Real48*에 저장하면 0이 됩니다.

Single 타입 (The Single type)

4바이트(32비트)의 *Single* 숫자는 다음과 같이 세 필드로 나뉩니다.



숫자의 *v* 값은 다음 식으로 구합니다.

0 < *e* < 255 이면 $v = (-1)^s * 2^{(e-127)} * (1.f)$

e = 0 이고 *f* <> 0 이면 $v = (-1)^s * 2^{(-126)} * (0.f)$
e = 0 이고 *f* = 0 이면 $v = (-1)^s * 0$
e = 255 이고 *f* = 0 이면 $v = (-1)^s *$ Inf
e = 255 이고 *f* <> 0 이면 *v*는 NaN(Not a Number) 입니다 .

Double 타입 (The Double type)

8바이트(64비트)의 *Double* 숫자는 다음과 같이 세 필드로 나눕니다.

1	11	52
s	*e*	*f*

숫자의 *v* 값은 다음 식으로 구합니다.

$0 < e < 2047$ 이면 $v = (-1)^s * 2^{(e-1023)} * (1.f)$

$e = 0$ 이고 $f <> 0$ 이면 $v = (-1)^s * 2^{(-1022)} * (0.f)$
$e = 0$ 이고 $f = 0$ 이면 $v = (-1)^s * 0$
$e = 2047$ 이고 $f = 0$ 이면 $v = (-1)^s *$ Inf
$e = 2047$ 이고 $f <> 0$ 이면 v는 NaN 입니다 .

Extended 타입 (The Extended type)

10바이트(80비트)의 *Extended* 숫자는 다음과 같이 네 필드로 나눕니다.

1	15	1	63
s	*e*	i	*f*

숫자의 *v* 값은 다음 식으로 구합니다.

$0 < e < 32767$ 이면 $v = (-1)^s * 2^{(e-16383)} * (i.f)$

$e = 32767$ 이면 $f = 0$ 이면 $v = (-1)^s *$ Inf
$e = 32767$ 이면 $f <> 0$ 이면 v는 NaN 입니다 .

Comp 타입 (The Comp type)

8바이트(64비트)의 *Comp* 숫자는 부호있는 64비트 정수로 저장됩니다.

Currency 타입 (The Currency type)

8바이트(64비트)의 *Currency* 숫자는 범위가 긴 부호있는 64비트 정수로 저장됩니다. Currency 숫자는 소수점 4자리를 암시하는 4개의 최하위 숫자를 가지고 있습니다.

포인터 타입(Pointer types)

포인터 타입은 32비트 주소로 4바이트에 저장됩니다. 포인터 값 **nil**은 0으로 저장됩니다.

짧은 문자열 타입(Shont String types)

문자열은 최대 길이에 1 바이트를 더한 크기를 사용합니다. 첫 번째 바이트에는 문자열의 현재 동적 길이가 들어 있고 다음 바이트에는 문자열의 문자가 들어 있습니다.

길이 바이트와 문자는 부호없는 값으로 간주됩니다. 최대 문자열 길이는 255개의 문자에 길이 바이트(**string**[255])를 더한 것입니다.

긴 문자열 타입(Long string types)

긴 문자열 변수는 동적으로 할당된 문자열에 대한 포인터가 들어 있는 4바이트 메모리를 사용합니다. 긴 문자열 변수가 비어 있으면(길이가 0인 문자열) 문자열 포인터는 **nil**이고 어떠한 동적 메모리도 해당 문자열 변수에 연결되지 않습니다. 비어 있지 않은 문자열 값인 경우 문자열 포인터는 32비트 길이 지시자와 32비트 참조 카운트를 비롯한 문자열 값이 들어 있는 동적으로 할당된 메모리 블록을 가리킵니다. 아래 테이블은 긴 문자열 메모리 블록의 레이아웃을 보여줍니다.

표 11.1 긴 문자열 동적 메모리 레이아웃

오프셋	내용
−8	32비트 참조 카운트
−4	바이트 단위의 길이
0..*길이* − 1	문자열
길이	Null 문자

긴 문자열 메모리 블록의 끝에 있는 Null 문자는 컴파일러와 내장 문자열 처리 루틴에 의해 자동으로 유지됩니다. 그러므로 긴 문자열을 Null로 끝나는 문자열로 직접 타입 변환할 수 있습니다.

문자열 상수와 리터럴에 대해 컴파일러는 동적으로 할당된 문자열과 똑같은 레이아웃을 가지는 메모리 블록을 만듭니다. 그러나, 이 메모리 블록의 참조 카운트는 −1입니다. 긴 문자열 변수에 문자열 상수가 지정되면 문자열 포인터에는 해당 문자열 상수에 대해 생성된 메모리 블록의 주소가 지정됩니다. 내장 문자열 처리 루틴은 참조 카운트가 −1인 블록을 수정하지 않습니다.

와이드 문자열 타입(Wide string types)

참고 Linux에서 와이드 문자열은 긴 문자열과 똑같이 구현됩니다.

Windows에서 와이드 문자열 변수는 동적으로 할당된 문자열에 대한 포인터가 들어 있는 4바이트 크기의 메모리를 사용합니다. 와이드 문자열 변수가 비어 있으면(길이가 0인 문자열) 문자열 포인터는 **nil**이고 어떠한 동적 메모리도 해당 문자열 변수에 연결되지 않습니다. 비어 있지 않은 문자열 값인 경우 문자열 포인터는 32비트 길이 지시자를 비롯한 해당 문자열 값이 들어 있는 동적으로 할당된 메모리 블록을 가리킵니다. 아래 표는 Windows에서 와이드 문자열 메모리 블록의 레이아웃을 보여줍니다.

표 11.2 와이드 문자열 동적 메모리 레이아웃 (Windows 만 해당)

오프셋	내용
−4	바이트 단위의 32비트 길이 지시자
0..*길이* − 1	문자열
길이	Null 문자

문자열 길이는 바이트 수이므로 길이는 문자열에 들어 있는 와이드 문자 수의 두 배입니다.

와이드 문자열 메모리 블록의 끝에 있는 Null 문자는 컴파일러와 내장 문자열 처리 루틴에 의해 자동으로 유지됩니다. 그러므로 와이드 문자열을 Null로 끝나는 문자열로 직접 타입 변환할 수 있습니다.

집합 타입(Set types)

집합이란 비트 배열로, 각 비트는 요소가 집합에 속하는지 여부를 나타냅니다. 집합의 최대 요소 수는 256개이므로 집합 하나는 항상 32바이트 이하입니다. 특정 집합이 차지하는 바이트 수는 다음과 같습니다.

(*Max* div 8) - (*Min* div 8) + 1

여기서, *Max*와 *Min*은 해당 집합의 기본 타입의 상위 경계와 하위 경계를 나타냅니다. 특정 요소 *E*의 바이트 수는 다음과 같습니다.

(*E* div 8) - (*Min* div 8)

이 바이트 내의 비트 수는 다음과 같습니다.

E mod 8

여기서, *E*는 요소의 순서 값을 표시합니다. 가능하다면 컴파일러는 집합을 CPU 레지스터에 저장하지만 집합이 일반 *정수* 타입 보다 크거나 집합의 주소를 가지는 코드가 프로그램에 있으면, 집합을 항상 메모리에 저장합니다.

정적 배열 타입(Static array types)

정적 배열은 컴포넌트 타입의 배열 변수들을 일련의 연속적인 변수로 저장합니다. 최하위 인덱스를 갖는 컴포넌트는 최하위 메모리 주소에 저장됩니다. 다차원 배열은 첫 번째로 증가하는 가장 오른쪽 차원에 저장됩니다.

동적 배열 타입(Dynamic array types)

동적 배열 변수는 동적으로 할당된 배열에 대한 포인터가 들어 있는 4바이트 메모리를 사용합니다. 초기화되지 않아서 변수가 비어 있거나 길이가 0인 배열인 경우 포인터는 **nil**이고 어떠한 동적 메모리도 해당 변수에 연결되지 않습니다. 비어 있지 않은 배열인 경우 변수는 32비트 길이 지시자와 32비트 참조 카운트를 비롯한 해당 배열이 들어 있는 동적으로 할당된 메모리 블록을 가리킵니다. 아래 테이블은 동적 배열 메모리 블록의 레이아웃을 보여줍니다.

표 11.3 동적 배열 메모리 레이아웃

오프셋	내용
−8	32비트 참조 카운트
−4	32비트 길이 지시자(요소 수)
0..*길이* * (요소의 크기) − 1	배열 요소

레코드 타입(Record types)

레코드 타입이 기본 설정인 {$A+} 상태에서 선언되고 선언에 packed 변경자가 들어 있지 않으면 해당 타입은 *압축되지 않은(unpacked) 레코드 타입*이고 레코드의 필드는 CPU가 효율적으로 액세스할 수 있도록 정렬됩니다. 각 필드의 타입에 따라 정렬됩니다. 모든 데이터 타입에는 컴파일러가 자동으로 계산하는 고유의 정렬이 있습니다. 정렬은 1, 2, 4, 또는 8이 될 수 있으며 가장 효율적인 액세스를 하기 위하여 해당 타입의 값을 반드시 저장해야 하는 바이트 경계를 나타냅니다. 아래 표는 모든 데이터 타입에 대한 정렬을 보여줍니다.

표 11.4 타입 정렬 마스크

타입	정렬
순서 타입	타입의 크기(1, 2, 4 또는 8)
실수 타입	*Real48 타입*은 2, *Single 타입*은 4, *Double 타입* 및 *Extended 타입*은 8
짧은 문자열 타입	1
배열 타입	배열의 요소 타입과 동일
레코드 타입	레코드에서 필드의 가장 큰 정렬
집합 타입	1, 2 또는 4일 경우는 타입의 크기, 그렇지 않으면 1
그 외 타입	4

압축되지 않은 레코드 타입에서 필드를 적절하게 정렬하려면 컴파일러는 정렬이 2인 필드 앞에 사용하지 않은 바이트를 삽입하고 필요할 경우 정렬이 4인 필드 앞에 최대 3개의 사용하지 않은 바이트를 삽입합니다. 마지막으로 컴파일러는 전체 레코드 크기를 필드의 가장 큰 정렬이 지정한 바이트 경계까지 올림합니다.

레코드 타입이 {$A−} 상태에서 선언되거나 선언에 packed 변경자가 들어 있는 경우 해당 레코드의 필드는 정렬되지 않고 대신 연속적인 오프셋이 지정됩니다. 이러한 압축 레코드의 전체 크기는 단순히 모든 필드의 크기가 됩니다. 데이터 정렬은 변경할 수 있기 때문에 디스크에 기록하려는 레코드 구조나, 메모리에서 버전이 다른 컴파일러를 사용하여 컴파일된 다른 모듈로 넘기려는 레코드 구조는 압축하는 것이 좋습니다.

파일 타입(File types)

파일 타입은 레코드로 표시됩니다. 타입이 지정된 파일과 타입이 지정되지 않은 파일은 다음과 같이 332바이트를 사용합니다.

```
type
  TFileRec = packed record
    Handle:Integer;
    Mode:word;
    Flags:word;
    case Byte of
      0: (RecSize:Cardinal;
      1: (BufSize:Cardinal;
          BufPos:Cardinal;
          BufEnd:Cardinal;
          BufPtr:PChar;
```

```
        OpenFunc:Pointer;
        InOutFunc:Pointer;
        FlushFunc:Pointer;
        CloseFunc:Pointer;
        UserData:array[1..32] of Byte;
        Name:array[0.0.259] of Char;
  end;
```

텍스트 파일은 다음과 같이 460바이트를 차지합니다.

```
type
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle:Integer;
    Mode:word;
    Flags:word;
    BufSize:Cardinal;
    BufPos:Cardinal;
    BufEnd:Cardinal;
    BufPtr:PChar;
    OpenFunc:Pointer;
    InOutFunc:Pointer;
    FlushFunc:Pointer;
    CloseFunc:Pointer;
    UserData:array[1..32] of Byte;
    Name:array[0.0.259] of Char;
    Buffer: TTextBuf;
  end;
```

파일이 열리면 *Handle*은 파일의 핸들을 갖게 됩니다.

Mode 필드는 다음 값 중 하나를 가질 수 있습니다

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

여기서, *fmClosed*는 파일이 닫혀 있음을 나타내고, *fmInput*와 *fmOutput*은 다시 설정되었거나(*fmInput*) 다시 작성된(*fmOutput*) 텍스트 파일을 나타내고, *fmInOut*는 다시 설정되었거나 다시 작성한 타입이 지정된 파일 또는 타입이 지정되지 않은 파일을 나타냅니다. 다른 값은 파일 변수가 할당되지 않아 초기화되지 않음을 나타냅니다.

UserData 필드는 사용자가 작성한 루틴으로 데이터를 저장하는 데 사용할 수 있습니다.

*Name*에는 Null 문자(#0)로 끝나는 일련의 문자로 된 파일 이름이 들어 있습니다.

타입이 지정된 파일과 타입이 지정되지 않은 파일인 경우 *RecSize*에는 바이트 단위의 레코드 길이가 들어 있고 *Private* 필드는 사용되지 않고 저장됩니다.

텍스트 파일인 경우 *BufPtr*은 *BufSize* 바이트의 버퍼에 대한 포인터이고, *BufPos*는 버퍼에서 읽거나 쓸 다음 문자의 인덱스이고, *BufEnd*는 버퍼에서 유효한 문자의 카운트입니다. *OpenFunc*, *InOutFunc*, *FlushFunc* 및 *CloseFunc*는 파일을 제어하는 I/O 루틴에 대한 포인터입니다. 8-5 페이지의 "장치 함수"를 참조하십시오. 플래그는 다음과 같은 줄 바꿈 모양을 결정합니다.

비트 0 해제	LF 줄 바꿈
비트 0 설정	CRLF 줄 바꿈

그 외 모든 플래그 비트는 나중에 사용하기 위해 예약됩니다. *DefaultTextLineBreakStyle* 및 *SetLineBreakStyle*을 참조하십시오.

프로시저 타입

프로시저 포인터는 프로시저나 함수의 엔트리 포인트를 32비트 포인터로 저장됩니다. 메소드 포인터는 메소드의 엔트리 포인트를 32비트 포인터로 저장하고 뒤에 있는 32비트 객체에 대한 포인터를 저장합니다.

클래스 타입

클래스 타입 값은 *객체*라는 클래스의 인스턴스에 대해 32비트 포인터로 저장됩니다. 객체의 내부 데이터 형식은 레코드 타입과 비슷합니다. 객체의 필드는 일련의 연속적인 변수로서 선언된 순서대로 저장됩니다. 필드는 압축되지 않은 레코드 타입처럼 항상 정렬됩니다. 조상 클래스에서 상속된 필드는 후손 클래스에서 새 필드가 정의되기 전에 저장됩니다.

모든 객체의 첫 번째 4바이트 필드는 클래스의 *가상 메소드 테이블*(VMT)에 대한 포인터입니다. 객체당 하나가 아니라 클래스당 하나의 VMT가 있어 클래스 타입을 구분하고 클래스의 유사한 정도에 상관 없이 절대 VMT를 공유하지 않습니다. VMT는 컴파일러가 자동으로 빌드하며 프로그램으로 직접 처리할 수는 없습니다. 생성자 메소드가 만든 객체에 의해 자동으로 저장되는 VMT에 대한 포인터 또한 프로그램에서 직접 처리할 수 없습니다.

다음 표는 VMT의 레이아웃을 보여줍니다. 양수 오프셋에서 VMT는 클래스 타입에서 사용자 정의된 가상 메소드 당 하나의 32비트 메소드 포인터 목록으로, 선언 순서대로 구성됩니다. 각 슬롯에는 해당 가상 메소드의 엔트리 포인트 주소가 들어 있습니다. 이 레이아웃은 C++ v-table 및 COM과 호환됩니다. 음수 오프셋에서 VMT는 오브젝트 파스칼의 구현에 대해 여러 개의 내부 필드가 들어 있습니다. 레이아웃은 향후 오브젝트 파스칼의 구현에서 변경되기 쉽기 때문에 이 정보를 쿼리하려면 애플리케이션에서 *TObject*에 정의된 메소드를 사용해야 합니다.

표 11.5 가상 메소드 테이블 레이아웃

오프셋	타입	설명
-76	*포인터*	가상 메소드 테이블에 대한 포인터(또는 **nil**)
-72	*포인터*	인터페이스 테이블에 대한 포인터(또는 **nil**)
-68	*포인터*	Automation 정보 테이블에 대한 포인터(또는 **nil**)

표 11.5 가상 메소드 테이블 레이아웃 (계속)

오프셋	타입	설명
−64	*포인터*	인스턴스 초기화 테이블에 대한 포인터(또는 **nil**)
−60	*포인터*	타입 정보 테이블에 대한 포인터(또는 **nil**)
−56	*포인터*	필드 정의 테이블에 대한 포인터(또는 **nil**)
−52	*포인터*	메소드 정의 테이블에 대한 포인터(또는 **nil**)
−48	*포인터*	동적 메소드 테이블에 대한 포인터(또는 **nil**)
−44	*포인터*	클래스 이름이 들어 있는 짧은 문자열에 대한 포인터
−40	*카디널*	바이트 단위의 인스턴스 크기
−36	*포인터*	조상 클래스에 대한 포인터에 대한 포인터(또는 **nil**)
−32	*포인터*	*SafecallException* 메소드의 엔트리 포인트에 대한 포인터(또는 **nil**)
−28	*포인터*	*AfterConstruction* 메소드의 엔트리 포인트
−24	*포인터*	*BeforeDestruction* 메소드의 엔트리 포인트
−20	*포인터*	*Dispatch* 메소드의 엔트리 포인트
−16	*포인터*	*DefaultHandler* 메소드의 엔트리 포인트
−12	*포인터*	*NewInstance* 메소드의 엔트리 포인트
−8	*포인터*	*FreeInstance* 메소드의 엔트리 포인트
−4	*포인터*	*Destroy* 소멸자의 엔트리 포인트
0	*포인터*	첫 번째 사용자 정의 가상 메소드의 엔트리 포인트
4	*포인터*	두 번째 사용자 정의 가상 메소드의 엔트리 포인트
⋮	⋮	⋮

클래스 참조 타입

클래스 참조 값은 클래스의 가상 메소드 테이블(VMT)에 대한 32비트 포인터로 저장됩니다.

가변 타입

가변 타입은 코드에서 지정하는 타입으로 타입 코드와 값(또는 값에 대한 참조)이 들어 있는 16바이트 레코드로 저장됩니다. *시스템* 유닛 및 *Variants* 유닛은 가변 타입 상수와 타입을 정의합니다.

TVarData 타입은 *가변* 변수의 내부 구조를 나타내고 Windows에서 COM과 Win32 API가 사용하는 *Variant* 타입과 같습니다. *TVarData* 타입은 변수의 내부 구조에 액세스하기 위하여 *가변* 변수의 타입 변환에서 사용될 수 있습니다.

TVarData 레코드의 *VType* 필드에는 *varTypeMask* 상수에 의해 정의된 비트인 하위 12비트에 있는 가변 타입 코드가 들어 있습니다. 게다가 *varArray* 비트로 해당 배열이 가변 타입임을 나타낼 수 있고 *varByRef* 비트로 가변 타입이 값이 아닌 참조임을 나타낼 수 있습니다.

TVarData 레코드의 *Reserved1*, *Reserved2* 및 *Reserved3* 필드는 사용되지 않습니다.

TVarData 레코드의 나머지 8바이트의 내용은 *VType* 필드에 따라 다릅니다. *varArray*나 *varByRef* 비트 모두 설정되지 않았을 경우 가변에는 특정 타입의 값이 들어 있습니다.

varArray 비트가 설정되어 있으면 가변에는 배열을 정의하는 *TVarArray* 구조에 대한 포인터가 들어 있습니다. 각 배열 요소의 타입은 *VType* 필드의 *varTypeMask* 비트에 의해 결정됩니다.

varByRef 비트가 설정되어 있으면 가변에는 *VType* 필드의 *varTypeMask*와 *varArray* 비트에 의해 지정된 타입 값에 대한 참조가 들어 있습니다.

varString 타입 코드는 private입니다. *varString* 값이 들어 있는 가변은 Delphi가 아닌 함수로 전달하면 안됩니다. Windows에서 Delphi의 Automation 지원은 *varString* 가변 타입을 외부 함수에 대한 매개변수로 전달하기 전에 자동으로 *varOleStr* 가변 타입으로 변환합니다.

Linux에서는 *VT_decimal*이 지원되지 않습니다.

12

프로그램 제어

이 장에서는 매개변수 및 함수 결과를 저장하고 전송하는 방법에 대해 설명합니다. 마지막 단원에서는 종료(exit) 프로시저에 대해 설명합니다.

매개변수 및 함수 결과

매개변수 처리와 함수 결과는 호출 규칙, 매개변수 의미론 및 전달되는 값의 타입과 크기 같은 몇 가지 요소들에 의해 결정됩니다.

매개변수 전달

매개변수는 루틴의 호출 규칙에 따라 CPU 레지스터나 스택을 통해 프로시저 및 함수로 전송됩니다. 호출 규칙에 관한 자세한 내용은 6-5 페이지의 "호출 규칙(Calling conventions)"을 참조하십시오.

변수(**var**) 매개변수는 항상 실제 저장소 위치를 가리키는 32비트 포인터로서 참조에 의해 전달됩니다.

값과 상수(**const**) 매개변수는 매개변수의 타입 타입과 크기에 따라 값에 의해(by value) 전달되거나 참조에 의해(by reference) 전달됩니다.

- 순서 매개변수는 해당 변수의 타입과 동일한 형식을 사용하여 8비트, 16비트, 32비트 또는 64비트 값으로 전달됩니다.
- 실수 매개변수는 항상 스택에 전달됩니다. *Single* 매개변수는 4바이트를 사용하고, *Double*, *Comp* 또는 *Currency* 매개변수는 8바이트를 사용합니다. *Real48*은 하위 6바이트에 저장된 *Real48* 값을 가지며 8바이트를 사용합니다. *Extended*는 하위 10바이트에 저장된 *Extended* 값을 가지며 12바이트를 사용합니다.
- 짧은 문자열 매개변수는 짧은 문자열에 대한 32비트 포인터로 전달됩니다.
- 긴 문자열이나 동적 배열 매개변수는 긴 문자열에 할당된 동적 메모리 블록에 대한 32비트 포인터로 전달됩니다. 값 **nil**은 비어 있는 긴 문자열에 대해 전달됩니다.

- 포인터, 클래스, 클래스 참조 또는 프로시저 포인터 매개변수는 32비트 포인터로 전달됩니다.
- 메소드 포인터는 두 개의 32비트 포인터로 스택에 전달됩니다. 메소드 포인터 앞에 인스턴스 포인터를 푸시하여 메소드 포인터는 최하위 주소를 사용하게 됩니다.
- **register** 및 **pascal** 규칙에서 가변 매개변수는 *가변* 값에 대한 32비트 포인터로 전달됩니다.
- 집합, 레코드 및 1, 2 또는 4바이트의 정적 배열은 8비트, 16비트 및 32비트 값으로 전달됩니다. 더 큰 집합, 레코드 및 정적 배열은 해당 값에 대한 32비트 포인터로 전달됩니다. 이 규칙의 예외는 레코드가 항상 **cdecl**, **stdcall** 및 **safecall** 규칙에서 스택에 직접 넘겨진다는 점으로, 이런 방식으로 넘겨지는 레코드의 크기는 가장 근접한 더블 워드로 올림됩니다.
- 개방형 배열 매개변수는 두 개의 32비트 값으로 전달됩니다. 첫 번째 값은 배열 데이터에 대한 포인터이고, 두 번째 값은 해당 배열의 요소 수에서 1을 뺀 값입니다.

두 개의 매개변수가 스택에 전달되면, 각 매개변수는 4바이트의 두 배(더블 워드의 전체 수)를 사용합니다. 8비트나 16비트 매개변수인 경우 이 매개변수가 단지 1바이트나 1 워드만 차지할지라도 더블 워드로 전달됩니다. 더블 워드에서 사용되지 않은 부분의 내용은 정의되지 않습니다.

pascal, **cdecl**, **stdcall** 및 **safecall** 규칙에서 모든 매개변수는 스택에 전달됩니다. **pascal** 규칙에서는 매개변수가 선언된 순서대로 왼쪽에서 오른쪽으로 푸시되므로 첫 번째 매개변수는 최상위 주소를 갖게 되고, 마지막 매개변수는 최하위 주소를 가지게 됩니다. **cdecl**, **stdcall** 및 **safecall** 규칙에서는 매개변수가 선언된 역순으로 오른쪽에서 왼쪽으로 푸시되므로 첫 번째 매개변수는 최하위 주소를 갖게 되고, 마지막 매개변수는 최상위 주소를 가지게 됩니다.

register 규칙에서는 매개변수가 3개까지 CPU 레지스터에 전달되고 나머지 매개변수는 스택에 전달됩니다. 이 매개변수는 **pascal** 규칙처럼 선언된 순서대로 전달되고, 한정된 처음 3개의 매개변수는 순서대로 EAX, EDX 및 ECX 레지스터에 전달됩니다. 실수 타입, 메소드 포인터 타입, 가변 타입, *Int64 타입* 및 구조 타입은 레지스터 매개변수로 한정하지는 않지만, 다른 모든 매개변수는 한정합니다. 3개 이상의 매개변수가 레지스터 매개변수로 한정할 경우, 처음 3개의 매개변수는 EAX, EDX 및 ECX에 전달되고, 나머지 매개변수들은 선언 순서대로 스택에 푸시됩니다. 예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E:Pointer};
```

*Test*에 대한 호출은 EAX에 *A*를 32비트 정수로 전달하고, EDX에 *B*를 *Char*에 대한 포인터로 전달하고, ECX에 *D*를 그리고 긴 문자열 메모리 블록에 대한 포인터로 전달합니다. *C*와 *E*는 두 개의 더블 워드와 32비트 포인터로 순서대로 스택에 푸시됩니다.

레지스터 저장 규칙

프로시저 및 함수는 EBX, ESI, EDI 및 EBP 레지스터를 그대로 유지해야 하지만 EAX, EDX 및 ECX 레지스터는 수정할 수 있습니다. 어셈블러로 생성자나 소멸자를 구현할 때 DL 레지스터는 반드시 유지하십시오. CLD 명령에 따라 프로시저 및 함수는 CPU의 방향 플래그가 삭제되었다는 가정 하에 호출되고, 프로시저 및 함수는 방향 플래그를 삭제한 다음에 반환되어야 합니다.

함수 결과값

다음 규칙은 함수 결과값을 반환하는데 사용됩니다.

- 순서 결과는 가능하다면 CPU 레지스터에 반환됩니다. 바이트는 AL에 반환되고, 워드는 AX에 반환되고, 더블 워드는 EAX에 반환됩니다.
- 실수 결과값은 부동 소수점 보조 프로세서의 맨 위 스택 레지스터 (ST(0))에 반환됩니다. *Currency* 타입의 함수 결과인 경우, ST(0)의 값은 10000으로 나눠지거나 곱해집니다. 예를 들어 *Currency* 값 1.234는 ST(0)에 12340으로 반환됩니다.
- 문자열, 동적 배열, 메소드 포인터, 가변 또는 *Int64* 결과인 경우, 마치 해당 함수 결과가 선언된 매개변수 뒤에 오는 추가 **var** 매개변수로서 선언된 것과 동일한 결과를 얻게 됩니다. 다시 말해서 호출자는 함수 결과를 반환할 변수를 가리키는 추가 32비트 포인터를 전달합니다.
- 포인터, 클래스, 클래스 참조 및 프로시저 포인터 결과는 EAX에 반환됩니다.
- 정적 배열, 레코드 및 집합 결과인 경우, 해당 값이 1바이트를 사용하면 AL에 반환되고, 2바이트를 사용하면 AX에 반환되고, 4바이트를 사용하면 EAX에 반환됩니다. 그 외의 결과는 선언된 매개변수 다음의 함수로 전달되는 추가적인 **var** 매개변수에 반환됩니다.

메소드 호출

메소드는 보통 프로시저 및 함수와 같은 호출 규칙을 사용합니다. 단 모든 메소드는 메소드가 호출된 인스턴스나 클래스에 대한 참조인 추가 암시적 매개변수 *Self*를 갖는다는 사실이 다릅니다. *Self* 매개변수는 32비트 포인터로 전달됩니다.

- **register** 규칙에서 *Self*는 다른 모든 매개변수 *보다 먼저* 선언된 것처럼 동작합니다. 따라서 Self는 항상 EAX 레지스터에 전달됩니다.
- **pascal** 규칙에서 *Self*는 때때로 함수 결과로 전달되는 추가 **var** 매개변수를 비롯한 다른 모든 매개변수 *보다 나중*에 선언된 것처럼 동작합니다. 그러므로 이것은 마지막에 푸시되어 다른 모든 매개변수 보다 하위 주소를 가집니다.
- **cdecl**, **stdcall** 및 **safecall** 규칙에서 *Self*는 다른 모든 매개변수 *보다 먼저* 선언된 것처럼 동작하고, 함수 결과로 전달되는 추가 **var** 매개변수가 있을 경우에는 매개변수 *보다 나중*에 선언된 것처럼 동작합니다. 그러므로 이것은 추가 **var** 매개변수일 경우를 제외하고는 마지막에 푸시됩니다.

생성자 및 소멸자

생성자 및 소멸자는 다른 메소드와 동일한 호출 규칙을 사용합니다. 단 *Boolean* 플래그 매개변수는 생성자나 소멸자 호출의 전후관계를 나타내기 위해 추가적으로 전달됩니다.

생성자 호출의 플래그 매개변수에서 *False* 값은 생성자가 인스턴스 객체를 통해서 또는 **inherited** 키워드를 사용하여 호출되었음을 나타냅니다. 이 경우 생성자는 일반적인 메소드처럼 동작합니다. 생성자 호출의 플래그 매개변수에서 *True* 값은 생성자가 클래스 참조를 통해 호출되었음을 나타냅니다. 이 경우 생성자는 *Self*에 의해 주어지는 클래스의 인스턴스를 만들고 새로 만든 객체에 대한 참조를 EAX에 반환합니다.

소멸자 호출의 플래그 매개변수에서 *False* 값은 소멸자가 **inherited** 키워드를 사용하여 호출되었음을 나타냅니다. 이 경우 소멸자는 일반적인 메소드처럼 동작합니다. 소멸자 호출의 플래그 매개변수에서 *True* 값은 소멸자가 인스턴스 객체를 통해 호출되었음을 나타냅니다. 이 경우 소멸자는 반환하기 바로 직전에 *Self*에 의해 주어진 인스턴스를 해제합니다.

플래그 매개변수는 다른 모든 매개변수 보다 먼저 선언된 것처럼 동작합니다. **register** 규칙에서 플래그 매개변수는 DL 레지스터에 전달됩니다. **pascal** 규칙에서는 다른 모든 매개변수 보다 먼저 푸시됩니다. **cdecl**, **stdcall** 및 **safecall** 규칙에서는 *Self* 매개변수 바로 앞에서 푸시됩니다.

DL 레지스터는 생성자나 소멸자가 호출 스택의 가장 바깥쪽에 있는지 여부를 나타내기 때문에 사용자는 종료하기 전에 DL 값을 복원해야 합니다. 그래야 *BeforeDestruction* 또는 *AfterConstruction*을 제대로 호출할 수 있습니다.

종료 프로시저(Exit procedures)

*종료 프로시저*는 파일 업데이트와 닫기와 같은 특정 작업이 프로그램을 종료하기 전에 수행되었는지 확인합니다. *ExitProc* 포인터 변수를 사용하여 종료 프로시저를 "설치" 할 수 있으므로, 종료가 정상적이거나 *Halt*에 대한 호출에 의해 강요되거나 런타임 오류의 결과일지라도 프로그램 종료의 일부로 항상 호출됩니다. 종료 프로시저는 매개변수가 없습니다.

참고 모든 종료 동작에는 종료 프로시저 보다 완료 섹션을 사용하는 것이 좋습니다.(3-5 페이지의 "완료 섹션" 참조) 종료 프로시저는 실행 파일, 공유 객체(Linux) 또는 .DLL (Windows)을 대상으로만 사용할 수 있습니다. 패키지에 대해서는, 종료 동작이 반드시 완료 섹션에서 구현되어야 합니다. 모든 종료 프로시저는 완료 섹션 실행 전에 호출됩니다.

프로그램 뿐만 아니라 유닛도 종료 프로시저를 설치할 수 있습니다. 유닛은 파일을 닫거나 다른 지우기 작업을 종료 프로시저에서 실행하도록 유닛의 초기화 코드의 일부로서 종료 프로시저를 설치할 수 있습니다.

종료 프로시저가 제대로 구현되면 종료 프로시저 체인의 일부가 됩니다. 이 프로시저는 설치의 역순서로 실행되어, 한 유닛의 종료 코드는 이 종료 코드에 의존하는 다른 유닛의 종료 코드가 끝나야 실행됩니다. 체인을 그대로 유지하려면 *ExitProc*의 현재 내용을 저장한 다음에 사용자의 종료 프로시저 주소를 가리키도록 해야합니다. 또한 사용자의 종료 프로시저 첫 번째 문장은 저장한 *ExitProc*의 값을 다시 설치해야 합니다.

다음 코드는 종료 프로시저의 뼈대 코드를 보여 줍니다.

```
var
  ExitSave:Pointer;

procedure MyExit;
begin
  ExitProc := ExitSave;  // always restore old vector first
  ⋮
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  ⋮
end.
```

위 코드는 *ExitProc*의 내용을 *ExitSave*에 저장한 다음, *MyExit* 프로시저를 설치합니다. 종료 프로세스의 일부로서 호출되었을 때 *MyExit*의 처음 동작은 이전 종료 프로시저를 다시 설치하는 것입니다.

런타임 라이브러리에서 종료 루틴은 *ExitProc*이 **nil**이 될 때까지 종료 프로시저를 계속 호출합니다. 무한 루프를 피하려면 호출하기 전에 매번 *ExitProc*를 **nil**로 설정해야 합니다. 그러면 현재 종료 프로시저가 *ExitProc*에 주소를 지정한 경우에만 다음 종료 프로시저가 호출됩니다. 종료 프로시저에서 오류가 발생할 경우 종료 프로시저는 다시 호출되지 않습니다.

종료 프로시저는 *ExitCode* 정수 변수 및 *ErrorAddr* 포인터 변수를 검사하여 종료 원인을 알아냅니다. 정상적인 종료일 경우, *ExitCode*는 0이고 *ErrorAddr*는 **nil**이 됩니다. *Halt*를 호출하여 종료했을 경우, *ExitCode*는 *Halt*에 전달된 값을 가지게 되고 *ErrorAddr*는 **nil**이 됩니다. 런타임 오류로 인한 종료일 경우, *ExitCode*는 오류 코드를 갖게 되고 *ErrorAddr*는 잘못된 문장의 주소를 갖게 됩니다.

런타임 라이브러리에 의해 설치된 마지막 종료 프로시저는 *Input* 및 *Output* 파일을 닫습니다. *ErrorAddr*이 **nil**이 아닐 경우, 런타임 오류 메시지를 출력합니다. 사용자의 런타임 오류 메시지를 출력하려면 *ErrorAddr*을 검사하여 **nil**이 아닐 경우 반환하기 전에 *ErrorAddr*에 **nil**을 설정합니다. 이렇게 하면 다른 종료 프로시저가 해당 오류를 다시 보고하지 않습니다.

일단 런타임 라이브러리가 종료 프로시저를 모두 호출했다면 반환 코드로 *ExitCode*에 저장된 값을 전달하여 해당 운영 체제에 반환합니다.

13

인라인 어셈블러 코드

기본 제공 어셈블러(built-in assembler)를 사용하여 오브젝트 파스칼 프로그램 내에서 Intel 어셈블러 코드를 작성할 수 있습니다. 기본 제공 어셈블러는 모든 8086/8087과 80386/80387 연산 코드 및 몇몇 터보 어셈블러의 표현식 연산자를 포함하는 터보 어셈블러 및 Microsoft의 매크로 어셈블러가 지원하는 구문의 커다란 서브셋을 구현합니다. 더욱이 기본 제공 어셈블러를 사용하여 어셈블러 문에 오브젝트 파스칼 식별자를 사용할 수 있습니다.

바이트, 워드 및 더블 워드를 정의하는 DB, DW 및 DD를 제외하고, EQU, PROC, STRUC, SEGMENT 및 MACRO와 같은 터보 어셈블러 지시어는 기본 제공 어셈블러에서 지원되지 않습니다. 하지만 터보 어셈블러 지시어를 통해 구현된 연산은 상응하는 오브젝트 파스칼 구문과 대략 일치합니다. 예를 들어 대부분의 EQU 지시어는 상수, 변수 및 타입 선언에 해당하고, PROC 지시어는 프로시저 및 함수 선언에 해당하고, STRUC 지시어는 레코드 타입에 해당합니다.

기본 제공 어셈블러에 대한 대안으로 외부 프로시저 및 함수를 포함하는 객체 파일을 연결할 수 있습니다. 자세한 내용은 6-7페이지의 "객체 파일과 연결"을 참조하십시오.

참고 외부 어셈블러 코드를 애플리케이션에 사용하려면 오브젝트 파스칼로 다시 작성하거나 인라인 어셈블러를 사용하여 최소한의 노력으로 다시 구현해야 합니다.

asm 문

기본 제공 어셈블러는 다음과 같은 형태의 **asm** 문을 통해 액세스합니다.

```
asm statementList end
```

여기서 *statementList*는 세미콜론, 줄 끝(end-of-line) 문자 또는 오브젝트 파스칼 주석문으로 구분된 일련의 어셈블러 문입니다.

asm 문의 주석은 반드시 오브젝트 파스칼 스타일이어야 합니다. 세미콜론은 줄의 나머지가 주석임을 나타내지 않습니다.

inline 예약어 및 **assembler** 지시어는 역 호환성을 위해 유지됩니다. 컴파일러에는 영향을 미치지 않습니다.

레지스터 사용

일반적으로 **asm** 문에서의 레지스터 사용 규칙은 **external** 프로시저나 함수의 규칙과 동일합니다. **asm** 문은 EDI, ESI, ESP, EBP 및 EBX 레지스터를 바꾸지 않고 유지해야 하지만 EAX, ECX 및 EDX 레지스터는 자유로이 수정할 수 있습니다. **asm** 문의 항목에서 BP는 현재 스택 프레임을 가리키고, SP는 스택의 맨 위를 가리키고, SS는 스택 세그먼트의 세그먼트 주소를 포함하고, DS는 데이터 세그먼트의 세그먼트 주소를 포함합니다. EDI, ESI, ESP, EBP 및 EBX를 제외한 **asm** 문은 문 항목에서 레지스터 내용에 관한 어떤 것도 가정할 수 없습니다.

어셈블러 문 구문

어셈블러 문의 구문은 다음과 같습니다.

Label: *Prefix Opcode* $Operand_1$, $Operand_2$

여기서 *Label*은 레이블이고, *Prefix*는 어셈블러 접두사 연산 코드(opcode)이고, *Opcode*는 어셈블러 명령 연산 코드 또는 지시어이고, *Operand*는 어셈블러 표현식입니다. *Label*과 *Prefix*는 옵션입니다. 일부 연산 코드는 연산자를 하나만 사용하고 일부 연산 코드는 연산자를 사용하지 않습니다.

주석은 어셈블러 문 사이에 사용되고 문 안에서는 사용되지 않습니다. 예를 들면, 다음과 같습니다.

```
MOV AX,1 {Initial value}     { OK }
MOV CX,100 {Count}           { OK }

MOV {Initial value} AX,1;    { Error! }
MOV CX, {Count} 100          { Error! }
```

레이블

오브젝트 파스칼에서 문장 앞에 레이블과 콜론을 찍어 기본 제공 어셈블러 문에서 레이블을 사용합니다. 레이블의 길이는 제한이 없지만 처음 32문자까지만 사용됩니다. 오브젝트 파스칼에서 레이블은 반드시 **asm** 문을 포함하는 블록의 **label** 선언 부분에 선언되어야 합니다. 이 규칙에 다음과 같은 예외가 하나 있습니다: *지역 레이블*.

지역 레이블은 **@** 기호로 시작하는 레이블입니다. 지역 레이블의 @ 다음에는 하나 이상의 문자, 숫자, 밑줄 또는 @ 기호가 올 수 있습니다. 지역 레이블 사용은 **asm** 문으로 제한되고, 지역 레이블의 유효 범위(scope)는 **asm** 예약어부터 이 예약어를 포함하는 **asm** 문의 끝까지입니다. 지역 레이블은 선언하지 않아도 됩니다.

명령 연산 코드

기본 제공 어셈블러는 일반 애플리케이션 사용에 대하여 Intel이 문서화한 모든 연산 코드를 지원합니다. 운영 체제 권한 명령은 지원되지 않을 수도 있습니다. 특별히 다음 명령 패밀리는 지원됩니다.

- Pentium 패밀리
- Pentium Pro

- MMX
- SIMD
- SSE
- AMD 3DNow!

각 명령에 대한 자세한 설명은 마이크로프로세서 설명서를 참조하십시오.

RET 명령 크기 조정

RET 명령 연산 코드는 항상 near 반환을 생성합니다.

자동 점프 크기 조정

다른 지시가 없을 경우 기본 제공 어셈블러는 자동으로 가장 짧고, 그래서 가장 효율적인 점프 명령 타입을 선택하여 점프 명령을 최적화합니다. 자동 점프 크기 조정은 무조건 점프 명령(JMP) 및 점프 대상이 프로시저나 함수가 아닌 레이블이면 모든 조건부 점프 명령에 적용됩니다.

무조건 점프 명령(JMP)인 경우, 기본 제공 어셈블러는 대상 레이블까지의 거리가 −128에서 127바이트이면 1바이트 연산 코드 뒤에 1바이트 변위가 있는 short 점프를 생성합니다. 그렇지 않을 경우 1바이트 연산 코드 뒤에 2바이트 변위가 있는 near 점프를 생성합니다.

조건부 점프 명령인 경우, 대상 레이블까지의 거리가 −128에서 127바이트이면 1바이트 연산 코드 뒤에 1바이트 변위가 있는 short 점프가 생성됩니다. 그렇지 않을 경우, 기본 제공 어셈블러는 대상 레이블까지 총 5바이트의 near 점프를 하는 역 조건을 가진 short 점프를 생성합니다. 어셈블러 문의 예는 다음과 같습니다.

```
JC      Stop
```

여기서 *Stop*은 short 점프의 범위에 있지 않고, 다음과 같이 이에 해당하는 기계 코드 시퀀스로 변환됩니다.

```
JNC     Skip
JMP     Stop
Skip:
```

프로시저 및 함수의 엔트리 포인트로의 점프는 항상 near입니다.

어셈블러 지시어

기본 제공 어셈블러는 다음 세 가지의 어셈블러 지시어를 지원합니다: DB(바이트 정의), DW(워드 정의) 및 DD(더블 워드 정의). 각 지시어는 해당 지시어 뒤에 오는 쉼표로 구분된 피연산자에 해당하는 데이터를 생성합니다.

DB 지시어는 일련의 바이트를 생성합니다. 각 피연산자는 −128과 255 사이의 값을 가진 상수 표현식이거나 길이에 상관 없는 문자열일 수 있습니다. 상수 표현식은 1바이트 코드를 생성하고, 문자열은 각 문자의 ASCII 코드에 해당하는 값을 가진 일련의 바이트를 생성합니다.

DW 지시어는 일련의 워드를 생성합니다. 각 피연산자는 −32,768과 65,535 사이의 값을 가진 상수 표현식이거나 주소 표현식일 수 있습니다. 주소 표현식인 경우 기본 제공 어셈블러는 해당 주소의 오프셋 부분을 포함하는 1워드 near 포인터를 생성합니다.

DD 지시어는 일련의 더블 워드를 생성합니다. 각 피연산자는 −2,147,483,648과 4,294,967,295 사이의 값을 가진 상수 표현식이거나 주소 표현식일 수 있습니다. 주소 표현식인 경우, 기본 제공 어셈블러는 뒤에 해당 주소의 세그먼트 부분을 포함하는 1워드가 있고 해당 주소의 오프셋 부분을 포함하는 1워드 far 포인터를 생성합니다.

DQ 지시어는 Int64 값에 대한 쿼드 워드(quadword)를 정의합니다.

DB, DW 및 DD 지시어가 생성한 데이터는 다른 기본 제공 어셈블러 문이 생성한 코드와 마찬가지로 항상 코드 세그먼트에 저장됩니다. 데이터 세그먼트에서 초기화되지 않은 데이터나 초기화된 데이터를 생성하려면 오브젝트 파스칼 **var** 또는 **const** 선언문을 사용해야 합니다.

DB, DW 및 DD 지시어의 예는 다음과 같습니다.

```
asm
  DB      0FFH                        { One byte }
  DB      0,99                        { Two bytes }
  DB      'A'                         { Ord('A') }
  DB      'Hello world...',0DH,0AH    { String followed by CR/LF }
  DB      12,"string"                 { Object Pascal style string }
  DW      0FFFFH                      { One word }
  DW      0,9999                      { Two words }
  DW      'A'                         { Same as DB 'A',0 }
  DW      'BA'                        { Same as DB 'A','B' }
  DW      MyVar                       { Offset of MyVar }
  DW      MyProc                      { Offset of MyProc }
  DD      0FFFFFFFFH                  { One double-word }
  DD      0,999999999                 { Two double-words }
  DD      'A'                         { Same as DB 'A',0,0,0 }
  DD      'DCBA'                      { Same as DB 'A','B','C','D' }
  DD      MyVar                       { Pointer to MyVar }
  DD      MyProc                      { Pointer to MyProc }
end;
```

터보 어셈블러에서 식별자가 DB, DW나 DD 지시어 앞에 있으면 지시어의 위치에 바이트 크기, 워드 크기, 더블 워드 크기의 변수를 선언하는 것입니다. 예를 들어 터보 어셈블러를 사용하면 다음과 같이 할 수 있습니다.

```
ByteVar     DB      ?
WordVar     DW      ?
IntVar      DD      ?
  ⋮
            MOV     AL,ByteVar
            MOV     BX,WordVar
            MOV     ECX,IntVar
```

기본 제공 어셈블러는 이러한 변수 선언문을 지원하지 않습니다. 인라인 어셈블러 문에 정의할 수 있는 유일한 기호 종류는 레이블입니다. 모든 변수는 반드시 오브젝트 파스칼 구문을 사용하여 선언되어야 합니다.

```
var
  ByteVar:Byte;
  WordVar:Word;
  IntVar:Integer;
  ⋮
asm
  MOV     AL,ByteVar
  MOV     BX,WordVar
  MOV     ECX,IntVar
end;
```

피연산자

기본 제공 어셈블러 피연산자는 상수, 레지스터, 기호 및 연산자로 구성된 표현식입니다. 피연산자 내에서 다음 예약어는 이미 정의된 의미를 갖습니다.

표 13.1 기본 제공 어셈블러 예약어

AH	BX	DI	EBX	ESP	OFFSET	SP
AL	BYTE	DL	ECX	FS	OR	SS
AND	CH	DS	EDI	GS	PTR	ST
AX	CL	DWORD	EDX	HIGH	QWORD	TBYTE
BH	CS	DX	EIP	LOW	SHL	TYPE
BL	CX	EAX	ES	MOD	SHR	WORD
BP	DH	EBP	ESI	NOT	SI	XOR

예약어는 항상 사용자 정의 식별자 보다 우선 순위를 가집니다. 예를 들면, 다음과 같습니다.

```
var
  Ch:Char;
  ⋮
asm
  MOV     CH, 1
end;
```

위의 예제에서 *Ch* 변수가 아닌 CH 레지스터에 1을 로드합니다. 예약어와 동일한 이름을 가진 사용자 정의 기호에 액세스하려면 다음과 같이 오버라이드 연산자 **&**를 사용해야 합니다.

```
MOV     &Ch, 1
```

기본 제공 어셈블러 예약어와 동일한 이름을 가진 사용자 정의 식별자는 사용하지 않는 것이 가장 좋습니다.

표현식

기본 제공 어셈블러는 모든 표현식을 32비트 정수 값으로 계산합니다. 기본 제공 어셈블러는 부동 소수점과 문자열 상수를 제외한 문자열 값을 지원하지 않습니다.

표현식은 *표현식 요소*와 *연산자*로 구성되고, 각 표현식은 연결된 *표현식 클래스*와 *표현식 타입*을 가집니다.

오브젝트 파스칼과 어셈블러 표현식의 차이

오브젝트 파스칼 표현식과 기본 제공 어셈블러 표현식의 가장 중요한 차이점은 어셈블러 표현식은 컴파일 시 계산될 수 있는 상수 값으로 변환해야 하는 점입니다. 예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
const
  X = 10;
  Y = 20;
var
  Z:Integer;
```

다음은 올바른 문장입니다.

```
asm
  MOV     Z,X+Y
end;
```

*X*와 *Y*가 모두 상수이기 때문에, 간단하게 상수 30을 표현식 X + Y라고 쓸 수 있습니다. 결과 명령은 단지 값 30을 변수 *Z*로 이동하면 됩니다. 그러나 *X*와 *Y*가 다음과 같은 변수일 경우,

```
var
  X, Y:Integer;
```

기본 제공 어셈블러는 컴파일 시 X + Y의 값을 계산할 수 없습니다. 이 경우 *X*와 *Y*의 합을 *Z*로 이동하려면 다음을 사용합니다.

```
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     Z,EAX
end;
```

오브젝트 파스칼 표현식에서 변수 참조는 해당 변수의 *내용*을 나타냅니다. 하지만 어셈블러 표현식에서 변수 참조는 해당 변수의 *주소*를 나타냅니다. 오브젝트 파스칼에서 표현식 X + 4(여기서 *X*는 변수)는 X 더하기 4의 내용을 의미하는 반면, 기본 제공 어셈블러에서 이 표현식은 *X*의 주소보다 4바이트 상위 주소에 있는 워드의 내용을 의미합니다. 그러므로 다음과 같은 코드를 쓸 수는 있지만

```
asm
  MOV     EAX,X
end;
```

이 코드가 *X* 더하기 4의 값을 AX에 로드하지는 않습니다. 대신 *X*보다 4바이트 상위에 저장된 워드의 값을 로드합니다. *X*의 내용에 4를 더하는 올바른 방법은 다음과 같습니다.

```
asm
  MOV     EAX,X
  ADD     EAX,Y
end;
```

표현식 요소

표현식의 요소는 *상수*, *레지스터* 및 *기호*입니다.

상수

기본 제공 어셈블러는 *숫자 상수*와 *문자열 상수*라는 두 가지 상수 타입을 지원합니다.

숫자 상수

숫자 상수는 반드시 정수여야 하며, 숫자 상수 값은 반드시 −2,147,483,648 과 4,294,967,295 사이여야 합니다.

기본적으로 숫자 상수는 십진수를 사용하지만 기본 제공 어셈블러는 이진수, 8진수 및 16진수도 지원합니다. 숫자 뒤에 *B*를 쓰면 이진수를 나타낼 수 있고, 숫자 뒤에 *O*를 쓰면 8진수, 숫자 뒤에 *H*를 쓰거나 숫자 앞에 **$**를 쓰면 16진수를 나타낼 수 있습니다.

숫자 상수는 1부터 9사이의 숫자 중 하나 또는 **$** 문자로 시작해야 합니다. *H* 접미사를 사용하여 16진수를 표시할 때 첫 번째 유효 숫자가 A~F 중 하나로 시작한다면, 해당 숫자 앞에 별도의 0이 필요합니다. 예를 들어, `0BAD4H`와 `$BAD4`는 16진수 상수이지만 `BAD4H`는 문자로 시작되기 때문에 식별자입니다.

문자열 상수

문자열 상수는 반드시 작은 따옴표나 큰 따옴표로 묶어야 합니다. 같은 모양의 연속된 인용 부호 두 개는 한 문자로 간주합니다. 다음은 문자열 상수의 예제입니다.

```
'Z'
'Delphi'
‘Linux’
"That's all folks"
'"That''s all folks," he said.'
'100'
'"'
"'"
```

길이에 상관 없는 문자열 상수는 DB 지시어에서 사용할 수 있는데, 이러한 문자열 상수는 문자열에서 해당 문자의 ASCII 값을 포함하는 일련의 바이트를 할당합니다. 다른 모든 경우에 문자열 상수는 4자 이하여야 하며, 표현식에 사용할 수 있는 숫자 값을 나타냅니다. 문자열 상수의 숫자 값은 다음과 같이 계산됩니다.

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

여기서 *Ch1*는 가장 오른쪽(마지막) 문자이고 *Ch4*는 가장 왼쪽(첫 번째) 문자입니다. 문자열이 4개의 문자보다 짧을 경우 가장 왼쪽 문자를 0으로 가정합니다. 다음 표는 문자열 상수와 문자열 상수와 숫자 값을 보여 줍니다.

표 13.2 문자열 예제 및 문자열 값

문자열	값
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a'	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

레지스터

다음 예약 기호는 CPU 레지스터를 나타냅니다.

표 13.3 CPU 레지스터

32비트 범용	EAX EBX ECX EDX	32비트 포인터 또는 인덱스	ESP EBP ESI EDI
16비트 범용	AX BX CX DX	16비트 포인터 또는 인덱스	SP BP SI DI
8비트 하위 레지스터	AL BL CL DL	16비트 세그먼트 레지스터	CS DS SS ES
		32비트 세그먼트 레지스터	FS GS
8비트 상위 레지스터	AH BH CH DH	보조 프로세서 레지스터 스택	ST

피연산자가 레지스터 이름만으로 구성된 경우 이것을 *레지스터 피연산자*라고 합니다. 모든 레지스터는 레지스터 피연산자로 사용될 수 있으며, 일부 레지스터는 다른 컨텍스트에서 사용할 수 있습니다.

기본 레지스터(BX 및 BP)와 인덱스 레지스터(SI 및 DI)는 인덱싱을 나타내기 위하여 대괄호를 쓸 수 있습니다. 유효한 기본/인덱스 레지스터 조합은 [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI] 및 [BP+DI] 입니다. [EAX+ECX], [ESP] 및 [ESP+EAX+5]와 같이 모든 32비트 레지스터를 사용하여 인덱싱할 수도 있습니다.

세그먼트 레지스터(ES, CS, SS, DS, FS 및 GS)는 지원되지만 세그먼트는 32비트 애플리케이션에서 정상적으로 사용할 수 없습니다.

ST 기호는 8087 부동 소수점 레지스터 스택의 맨 위 레지스터를 나타냅니다. 8개의 부동 소수점 레지스터는 각각 ST(*X*)를 사용하여 참조할 수 있는데, 여기서 *X*는 레지스터 스택 맨 위와의 거리를 나타내는 0과 7사이의 상수입니다.

기호

기본 제공 어셈블러를 사용하여 상수, 타입, 변수, 프로시저 및 함수를 비롯한 어셈블러 표현식의 모든 오브젝트 파스칼 식별자에 액세스할 수 있습니다. 더욱이 기본 제공 어셈블러는 함수 몸체 내에서 *Result* 변수에 해당하는 특수 기호인 *@Result*를 구현할 수 있습니다. 예를 들어 다음 함수는

```
function Sum(X, Y:Integer):Integer;
begin
  Result := X + Y;
end;
```

어셈블러에서 다음과 같이 쓸 수 있습니다.

```
function Sum(X, Y:Integer):Integer; stdcall;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

다음 기호는 **asm** 문에 사용할 수 없습니다.

- 표준 프로시저 및 함수(예: *WriteLn*과 *Chr*).
- *Mem*, *MemW*, *MemL*, *Port* 및 *PortW* 특수 배열.
- 문자열, 부동 소수점 및 집합 상수.
- 현재 블록에 선언되지 않은 레이블.
- 함수 외부에 있는 *@Result* 기호.

다음 표는 **asm** 문에서 사용할 수 있는 기호 종류를 요약한 것입니다.

표 13.4 기본 제공 어셈블러가 인식하는 기호

기호	값	분류	타입
Label	레이블의 주소	메모리 참조	SHORT
Constant	상수 값	즉시(immediate) 값	0
Type	0	메모리 참조	타입 크기
Field	필드의 오프셋	메모리	타입 크기
Variable	변수의 주소	메모리 참조	타입 크기
Procedure	프로시저의 주소	메모리 참조	NEAR
Function	함수의 주소	메모리 참조	NEAR
Unit	0	즉시(immediate)값	0
@Code	코드 세그먼트 주소	메모리 참조	0FFF0H
@Data	데이터 세그먼트 주소	메모리 참조	0FFF0H
@Result	결과 변수 오프셋	메모리 참조	타입 크기

최적화를 사용할 수 없도록 하면 프로시저 및 함수에 선언된 변수인 지역 변수는 항상 스택에 할당되고 EBP에 상대적으로 액세스됩니다. 그리고 지역 변수 기호의 값은 EBP의 부호있는 오프셋입니다. 어셈블러는 자동으로 지역 변수에 대한 참조에 [EBP]를 추가합니다. 예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
var Count:Integer;
```

함수나 프로시저 안에서 다음 명령,

```
MOV     EAX,Count
```

MOV EAX,[EBP−4]로 어셈블합니다.

기본 제공 어셈블러는 **var** 매개변수를 32비트 포인터로 취급하며, **var** 매개변수의 크기는 항상 4입니다. **var** 매개변수에 액세스하기 위한 구문은 값 매개변수에 액세스하기 위한 구문과는 다릅니다. **var** 매개변수의 내용에 액세스하려면, 우선 32비트 포인터를 로드한 다음 포인터가 가리키는 위치에 액세스해야 합니다. 예를 들면, 다음과 같습니다.

```
function Sum(var X, Y:Integer):Integer; stdcall;
begin
  asm
    MOV     EAX,X
    MOV     EAX,[EAX]
    MOV     EDX,Y
    ADD     EAX,[EDX]
    MOV     @Result,AX
  end;
end;
```

식별자는 **asm** 문 내에서 한정될 수 있습니다. 예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
type
  TPoint = record
    X, Y:Integer;
  end;
  TRect = record
    A, B:TPoint;
  end;
var
  P:TPoint;
  R: TRect;
```

필드에 액세스하기 위하여 **asm** 문에 다음과 같은 구문을 사용할 수 있습니다.

```
MOV     EAX,P.X
MOV     EDX,P.Y
MOV     ECX,R.A.X
MOV     EBX,R.B.Y
```

타입 식별자는 변수를 생성하는데 사용됩니다. 다음 명령은 각각 [EDX]의 내용을 EAX로 로드하는 동일한 기계 코드를 생성합니다.

```
MOV     EAX,(TRect PTR [EDX]).B.X
MOV     EAX,TRect(EDX]).B.X
MOV     EAX,TRect[EDX].B.X
MOV     EAX,[EDX].TRect.B.X
```

표현식 분류

기본 제공 어셈블러는 표현식을 *레지스터*, *메모리 참조* 및 *즉시 값*, 세 가지로 분류합니다.

레지스터 이름만으로 구성된 표현식이 *레지스터* 표현식입니다. 레지스터 표현식의 예는 AX, CL, DI 및 ES입니다. 피연산자로 사용되는 레지스터 표현식은 어셈블러에게 CPU 레지스터에 작동하는 명령을 생성하도록 지시합니다.

메모리 위치를 나타내는 표현식이 메모리 참조입니다. 오브젝트 파스칼의 레이블, 변수, 타입이 지정된 상수, 프로시저 및 함수가 이 범주에 속합니다.

레지스터가 아니고 메모리 위치와 연관되지 않은 표현식이 즉시(immediate) 값입니다. 이 그룹에는 오브젝트 파스칼의 타입이 지정되지 않은 상수와 타입 식별자가 포함됩니다.

즉시 값과 메모리 참조는 피연산자로서 사용되면 서로 다른 코드를 생성합니다.예를 들면, 다음과 같습니다.

```
const
  Start = 10;
var
  Count:Integer;
  ⋮
asm
  MOV     EAX,Start            { MOV EAX,xxxx }
  MOV     EBX,Count            { MOV EBX,[xxxx] }
  MOV     ECX,[Start]          { MOV ECX,[xxxx] }
  MOV     EDX,OFFSET Count     { MOV EDX,xxxx }
end;
```

*Start*가 즉시 값이기 때문에 첫 번째 MOV는 즉시 이동 명령으로 어셈블됩니다. 그러나 *Count*가 메모리 참조이므로 두 번째 MOV는 메모리 이동 명령으로 번역됩니다. 세 번째 MOV에서 대괄호는 *Start*를 메모리 참조(이 경우 데이터 세그먼트에서 오프셋 10에 있는 워드)로 변환합니다. 네 번째 MOV에서 OFFSET 연산자는 *Count*를 즉시 값(데이터 세그먼트에서 *Count*의 오프셋)으로 변환합니다.

대괄호와 OFFSET 연산자는 상호 보완적입니다. 다음 **asm** 문은 이전 **asm** 문의 처음 두 줄과 일치하는 기계 코드를 만듭니다.

```
asm
  MOV     EAX,OFFSET [Start]
  MOV     EBX,[OFFSET Count]
end;
```

메모리 참조와 즉시 값은 *재배치* 값과 *절대* 값으로 나눌 수 있습니다. 재배치란 링커가 기호에 절대 주소를 지정하는 프로세스입니다. 재배치 표현식은 연결 시 재배치를 필요로 하는 값을 나타내고, 반면 절대 표현식은 이러한 재배치를 필요로하지 않는 값을 나타냅니다. 전통적으로 레이블, 변수, 프로시저 또는 함수를 참조하는 표현식은 재배치 표현식입니다. 왜냐하면 컴파일 시 이러한 기호들의 최종 주소가 알려지지 않기 때문입니다. 유일하게 상수만 연산하는 표현식은 절대 표현식입니다.

기본 제공 어셈블러를 사용하면 절대 값에서 필요한 연산을 모두 수행할 수 있지만 재배치 값에서의 연산은 상수의 더하기와 빼기로 제한됩니다.

표현식 타입

모든 기본 제공 어셈블러 표현식은 단순히 타입, 좀 더 정확하게 말하자면 크기를 가집니다. 왜냐하면 어셈블러는 표현식의 타입을 메모리 위치의 크기로 간주하기 때문입니다. 예를 들어 *Integer* 변수의 타입은 4인데, 이 표현식이 4바이트를 차지하기 때문입니다. 기본 제공 어셈블러는 가능할 때마다 타입을 확인하는데, 다음과 같은 명령에서

```
var
  QuitFlag:Boolean;
  OutBufPtr:Word;
  ⋮
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

어셈블러는 *QuitFlag*의 크기가 1(1바이트)이고, *OutBufPtr*의 크기는 2(1워드)임을 확인합니다. 다음 명령에서는

```
MOV     DL,OutBufPtr
```

DL이 1바이트 크기의 레지스터이고 *OutBufPtr*은 1워드이기 때문에 오류가 발생합니다. 메모리 참조의 타입은 타입 변환을 통해 변경할 수 있으므로 위의 명령은 다음과 같이 사용하는 것이 옳습니다.

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

이러한 MOV 명령은 모두 *OutBufPtr* 변수의 첫 번째(최하위) 바이트를 참조합니다.

경우에 따라 메모리 참조 타입이 지정되어 있지 않습니다. 다음과 같이 대괄호로 묶인 즉시 값은 타입이 지정되지 않은 예입니다.

```
MOV     AL,[100H]
MOV     BX,[100H]
```

표현식 [100H]는 타입이 지정되지 않았으므로 이것은 단지 데이터 세그먼트에서 주소 100H의 내용만을 나타내고, 해당 타입은 첫 번째 피연산자(AL의 바이트, BX의 워드)에서 결정할 수 있기 때문에 기본 제공 어셈블러는 이러한 명령을 모두 사용할 수 있습니다. 다른 피연산자의 타입을 정할 수 없는 경우 기본 제공 어셈블러는 다음과 같은 명시적 타입 변환이 필요합니다.

```
INC     BYTE PTR [100H]
IMUL    WORD PTR [100H]
```

다음 표는 현재까지 선언한 오브젝트 파스칼 타입 이외에 기본 제공 어셈블러가 제공하는 이미 정의된 타입 기호를 요약한 것입니다.

표 13.5 이미 정의된 타입 기호

기호	타입
BYTE	1
WORD	2
DWORD	4

표 13.5 이미 정의된 타입 기호 (계속)

기호	타입
QWORD	8
TBYTE	10

표현식 연산자

기본 제공 어셈블러는 다양한 연산자를 제공합니다. 우선 순위는 오브젝트 파스칼과는 다릅니다. 예를 들어 asm 문에서 AND는 더하기 및 빼기 연산자보다 낮은 우선 순위를 가집니다. 다음 표는 기본 제공 어셈블러의 표현식 연산자를 우선 순위가 높은 순서대로 나열한 것입니다.

표 13.6 기본 제공 어셈블러 표현식 연산자의 우선 순위

연산자	주의	우선 순위
&		가장 높음
(), [], ., HIGH, LOW		
+, –	단항 + 및 –	
:		
OFFSET, SEG, TYPE, PTR, *, /, MOD, SHL, SHR, +, –	이항 + 및 –	
NOT, AND, OR, XOR		가장 낮음

다음 표는 기본 제공 어셈블러의 표현식 연산자를 정의한 것입니다.

표 13.7 기본 제공 어셈블러 표현식 연산자의 정의

연산자	설명
&	**식별자 오버라이드**. & 바로 뒤에 오는 식별자는 철자가 기본 제공 어셈블러 예약어 기호와 동일하더라도 사용자 정의 기호로 취급합니다.
(...)	**하위 표현식**. 괄호 내의 표현식을 모두 계산한 다음, 결과값을 하나의 표현식 요소로 취급합니다. 괄호 내에서 표현식 앞에 다른 표현식이 올 수 있습니다. 이 경우 결과는 첫 번째 표현식의 타입을 가진 두 표현식 값의 합입니다.
[...]	**메모리 참조**. 대괄호 내의 표현식을 모두 계산한 다음, 결과값을 하나의 표현식 요소로 취급합니다. 대괄호 내의 표현식은 + 연산자로 BX, BP, SI 또는 DI 레지스터와 결합하여 CPU 레지스터 인덱싱을 나타낼 수 있습니다. 대괄호 내에서 표현식 앞에 다른 표현식이 올 수 있습니다. 이 경우 결과는 첫 번째 표현식의 타입을 가진 두 표현식 값의 합입니다. 이 결과는 항상 메모리 참조입니다.
.	**구조 멤버 선택자**. 이 결과는 마침표 뒤에 있는 표현식의 타입을 가진, 마침표 앞에 있는 표현식과 마침표 뒤에 있는 표현식의 합입니다. 마침표 앞의 표현식이 지정한 유효 범위에 속하는 기호는 마침표 뒤의 표현식에서 액세스할 수 있습니다.
HIGH	연산자 뒤에 있는 워드 크기 표현식의 중요도가 높은 8비트를 반환합니다. 표현식은 절대 즉시 값(absolute immediate value)이어야 합니다.
LOW	연산자 뒤에 있는 워드 크기 표현식의 중요도가 낮은 8비트를 반환합니다. 표현식은 절대 즉시 값이어야 합니다.
+	**단항 +**. + 뒤에 오는 표현식을 변경하지 않고 반환합니다. 표현식은 절대 즉시 값(absolute immediate value)이어야 합니다.

표 13.7 기본 제공 어셈블러 표현식 연산자의 정의 (계속)

연산자	설명
–	**단항 –**. – 뒤에 오는 표현식의 부정 값을 반환합니다. 표현식은 절대 즉시 값이어야 합니다.
+	**더하기**. 이 표현식은 즉시 값이거나 메모리 참조일 수 있지만 표현식 중 하나만 재배치 값일 수 있습니다. 표현식 중 하나가 재배치 값일 경우 결과 또한 재배치 값이 됩니다. 표현식 중 어느 하나가 메모리 참조일 경우 결과 또한 메모리 참조가 됩니다.
–	**빼기**. 첫 번째 표현식은 세 분류 모두 될 수 있지만 두 번째 표현식은 반드시 절대 즉시 값이어야 합니다. 결과는 첫 번째 표현식과 동일한 분류가 됩니다.
:	**세그먼트 오버라이드**. 콜론 뒤의 표현식이 콜론 앞의 세그먼트 레지스터 이름(CS, DS, SS, FS, GS, 또는 ES)에 의해 주어진 세그먼트에 속한다고 어셈블러에게 지시합니다. 결과는 콜론 뒤의 표현식의 값을 가진 메모리 참조입니다. 세그먼트 오버라이드가 명령 피연산자에서 사용될 경우, 명령은 지시한 세그먼트가 선택되었음을 확인하기 위하여 적절한 세그먼트 오버라이드 접두사를 붙입니다.
OFFSET	연산자 뒤에 오는 표현식의 오프셋 부분(더블 워드)을 반환합니다. 결과는 즉시 값입니다.
TYPE	연산자 뒤에 오는 표현식의 타입(바이트 단위 크기)을 반환합니다. 즉시 값의 타입은 0입니다.
PTR	**타입 변환 연산자**. 결과는 연산자와 연산자 앞의 표현식의 타입 뒤에 오는 표현식의 값을 가진 메모리 참조입니다.
*	**곱하기**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
/	**정수 나누기**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
MOD	**정수 나누기의 나머지**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
SHL	**왼쪽으로 논리 시프트**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
SHR	**오른쪽으로 논리 시프트**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
NOT	**비트 단위 부정.** 표현식은 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
AND	**비트 단위 논리곱**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
OR	**비트 단위 논리합**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.
XOR	**비트 단위 배타적 논리합**. 두 표현식은 모두 절대 즉시 값이어야 하고 결과는 절대 즉시 값입니다.

어셈블러 프로시저 및 함수

begin...end 문 없이 인라인 어셈블러 코드를 사용하여 완전한 프로시저 및 함수를 작성할 수 있습니다. 예를 들면, 다음과 같습니다.

```
function LongMul(X, Y:Integer}:Longint;
asm
  MOV     EAX,X
  IMUL    Y
end;
```

컴파일러는 이 루틴에서 다음과 같은 몇 가지 최적화를 수행합니다.

- 값 매개변수를 지역 변수로 복사하는 어떠한 코드도 생성하지 않습니다. 이것은 모든 문자열 타입 값 매개변수와 크기가 1, 2 또는 4바이트인 다른 값 매개변수에 영향을 줍니다. 루틴 내에서 이러한 매개변수는 마치 **var** 매개변수인 것처럼 취급해야 합니다.
- 함수가 문자열, 가변 또는 인터페이스 참조를 반환하지 않을 경우 *@Result* 기호에 대한 참조는 오류이므로 컴파일러는 함수 결과 변수를 지정하지 않습니다. 문자열, 가변 및 인터페이스인 경우 호출자는 항상 *@Result* 포인터를 지정합니다.
- 컴파일러는 단지 중첩 루틴, 지역 매개변수를 갖는 루틴, 스택에 매개변수를 갖는 루틴에 대한 스택 프레임을 생성합니다.
- 해당 루틴에 대해 자동으로 생성된 입력 코드 및 종료 코드는 다음과 같습니다.

```
PUSH    EBP             ;Present if Locals <> 0 or Params <> 0
MOV     EBP,ESP         ;Present if Locals <> 0 or Params <> 0
SUB     ESP,Locals      ;Present if Locals <> 0
 ⋮
MOV     ESP,EBP         ;Present if Locals <> 0
POP     EBP             ;Present if Locals <> 0 or Params <> 0
RET     Params          ;Always present
```

  Locals 가 가변 , 긴 문자열 또는 인터페이스를 포함할 경우 , 0 으로 초기화되지만 완료되지는 않습니다 .

- *Locals*는 지역 변수의 크기이고 *Params*는 매개변수의 크기입니다. *Locals*와*Params* 모두 0이면 입력 코드는 없고 종료 코드는 RET 명령으로 간단히 구성됩니다.

어셈블러 함수는 다음과 같은 결과를 반환합니다.

- 순서 값은 AL(8비트 값), AX(16비트 값) 또는 EAX(32비트 값)에 반환됩니다.
- 실수 값은 보조 프로세서의 레지스터 스택의 ST(0)에 반환됩니다. (*Currency* 값은 10000으로 나눠지거나 곱해집니다.)
- 긴 문자열을 포함하는 포인터는 EAX에 반환됩니다.
- 짧은 문자열과 가변은 *@Result*가 가리키는 임시 위치에 반환됩니다.

부 록

A

오브젝트 파스칼 문법

```
Goal -> (Program | Package  | Library  | Unit)

Program -> [PROGRAM Ident ['(' IdentList ')'] ';']
           ProgramBlock '.'

Unit -> UNIT Ident ';'
        InterfaceSection
        ImplementationSection
        InitSection '.'

Package -> PACKAGE Ident ';'
           [RequiresClause]
           [ContainsClause]
           END '.'

Library -> LIBRARY Ident ';'
           ProgramBlock '.'

ProgramBlock -> [UsesClause]
                Block

UsesClause -> USES IdentList ';'

InterfaceSection -> INTERFACE
                    [UsesClause]
                    [InterfaceDecl]...

InterfaceDecl -> ConstSection
              -> TypeSection
              -> VarSection
              -> ExportedHeading

ExportedHeading -> ProcedureHeading ';' [Directive]
                -> FunctionHeading ';' [Directive]

ImplementationSection -> IMPLEMENTATION
                         [UsesClause]
                         [DeclSection]...

Block -> [DeclSection]
```

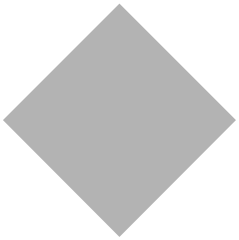
```
        CompoundStmt

DeclSection -> LabelDeclSection
            -> ConstSection
            -> TypeSection
            -> VarSection
            -> ProcedureDeclSection

LabelDeclSection -> LABEL LabelId

ConstSection -> CONST (ConstantDecl ';')...

ConstantDecl -> Ident '=' ConstExpr
             -> Ident ':' TypeId '=' TypedConstant

TypeSection -> TYPE (TypeDecl ';')...

TypeDecl -> Ident '=' Type
         -> Ident '=' RestrictedType

TypedConstant -> (ConstExpr | ArrayConstant | RecordConstant)

ArrayConstant -> '(' TypedConstant/','... ')'

RecordConstant -> '(' RecordFieldConstant/';'... ')'

RecordFieldConstant -> Ident ':' TypedConstant

Type -> TypeId
     -> SimpleType
     -> StrucType
     -> PointerType
     -> StringType
     -> ProcedureType
     -> VariantType
     -> ClassRefType

RestrictedType -> ObjectType
               -> ClassType
               -> InterfaceType

ClassRefType -> CLASS OF TypeId

SimpleType -> (OrdinalType | RealType)

RealType -> REAL48
         -> REAL
         -> SINGLE
         -> DOUBLE
         -> EXTENDED
         -> CURRENCY
         -> COMP

OrdinalType -> (SubrangeType | EnumeratedType | OrdIdent)

OrdIdent -> SHORTINT
         -> SMALLINT
         -> INTEGER
         -> BYTE
         -> LONGINT
         -> INT64
         -> WORD
         -> BOOLEAN
```

```
         -> CHAR
         -> WIDECHAR
         -> LONGWORD
         -> PCHAR

VariantType -> VARIANT
            -> OLEVARIANT

SubrangeType -> ConstExpr '..' ConstExpr

EnumeratedType -> '(' EnumeratedTypeElement/','... ')'

EnumeratedTypeElement -> Ident [ '=' ConstExpr ]

StringType -> STRING
           -> ANSISTRING
           -> WIDESTRING
           -> STRING '[' ConstExpr ']'

StrucType -> [PACKED] (ArrayType | SetType | FileType | RecType)

ArrayType -> ARRAY ['[' OrdinalType/','... ']'] OF Type

RecType -> RECORD [FieldList] END

FieldList ->  FieldDecl/';'... [VariantSection] [';']

FieldDecl -> IdentList ':' Type

VariantSection -> CASE [Ident ':'] TypeId OF RecVariant/';'...

RecVariant -> ConstExpr/','...  ':' '(' [FieldList] ')'

SetType -> SET OF OrdinalType

FileType -> FILE OF TypeId

PointerType -> '^' TypeId

ProcedureType -> (ProcedureHeading | FunctionHeading) [OF OBJECT]

VarSection -> VAR (VarDecl ';')...

VarDecl -> IdentList ':' Type [(ABSOLUTE (Ident | ConstExpr)) | '=' ConstExpr]

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['+' | '-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator ['(' ExprList ')']
       -> '@' Designator
       -> Number
       -> String
       -> NIL
       -> '(' Expression ')'
       -> NOT Factor
       -> SetConstructor
       -> TypeId '(' Expression ')'

RelOp -> '>'
      -> '<'
      -> '<='
      -> '>='
      -> '<>'
```

```
      -> IN
      -> IS
      -> AS

AddOp -> '+'
      -> '-'
      -> OR
      -> XOR

MulOp -> '*'
      -> '/'
      -> DIV
      -> MOD
      -> AND
      -> SHL
      -> SHR

Designator -> QualId ['.' Ident | '[' ExprList ']' | '^']...

SetConstructor -> '[' [SetElement/','...] ']'

SetElement -> Expression ['..' Expression]

ExprList -> Expression/','...

Statement -> [LabelId ':'] [SimpleStatement | StructStmt]

StmtList -> Statement/';'...

SimpleStatement -> Designator ['(' ExprList ')']
                -> Designator ':=' Expression
                -> INHERITED
                -> GOTO LabelId

StructStmt -> CompoundStmt
           -> ConditionalStmt
           -> LoopStmt
           -> WithStmt

CompoundStmt -> BEGIN StmtList END

ConditionalStmt -> IfStmt
                -> CaseStmt

IfStmt -> IF Expression THEN Statement [ELSE Statement]

CaseStmt -> CASE Expression OF CaseSelector/';'... [ELSE StmtList] [';'] END

CaseSelector -> CaseLabel/','... ':' Statement

CaseLabel -> ConstExpr ['..' ConstExpr]

LoopStmt -> RepeatStmt
         -> WhileStmt
         -> ForStmt

RepeatStmt -> REPEAT Statement UNTIL Expression

WhileStmt -> WHILE Expression DO Statement

ForStmt -> FOR QualId ':=' Expression (TO | DOWNTO) Expression DO Statement

WithStmt -> WITH IdentList DO Statement

ProcedureDeclSection -> ProcedureDecl
```

```
                    -> FunctionDecl

ProcedureDecl -> ProcedureHeading ';' [Directive]
                 Block ';'

FunctionDecl -> FunctionHeading ';' [Directive]
                Block ';'

FunctionHeading -> FUNCTION Ident [FormalParameters] ':' (SimpleType | STRING)

ProcedureHeading -> PROCEDURE Ident [FormalParameters]

FormalParameters -> '(' FormalParm/';'... ')'

FormalParm -> [VAR | CONST | OUT] Parameter

Parameter -> IdentList  [':' ([ARRAY OF] SimpleType | STRING | FILE)]
          -> Ident ':' SimpleType '=' ConstExpr

Directive -> CDECL
          -> REGISTER
          -> DYNAMIC
          -> VIRTUAL
          -> EXPORT
          -> EXTERNAL
          -> FAR
          -> FORWARD
          -> MESSAGE
          -> OVERRIDE
          -> OVERLOAD
          -> PASCAL
          -> REINTRODUCE
          -> SAFECALL
          -> STDCALL

ObjectType -> OBJECT [ObjHeritage] [ObjFieldList] [MethodList] END

ObjHeritage -> '(' QualId ')'

MethodList -> (MethodHeading [';' VIRTUAL])/';'...

MethodHeading -> ProcedureHeading
              -> FunctionHeading
              -> ConstructorHeading
              -> DestructorHeading

ConstructorHeading -> CONSTRUCTOR Ident [FormalParameters]

DestructorHeading -> DESTRUCTOR Ident [FormalParameters]

ObjFieldList -> (IdentList ':' Type)/';'...

InitSection -> INITIALIZATION StmtList [FINALIZATION StmtList] END
            -> BEGIN StmtList END
            -> END

ClassType -> CLASS [ClassHeritage]
             [ClassFieldList]
             [ClassMethodList]
             [ClassPropertyList]
             END

ClassHeritage -> '(' IdentList ')'
```

```
ClassVisibility -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]
ClassFieldList -> (ClassVisibility ObjFieldList)/';'...
ClassMethodList -> (ClassVisibility MethodList)/';'...
ClassPropertyList -> (ClassVisibility PropertyList ';')...
PropertyList -> PROPERTY  Ident [PropertyInterface]  PropertySpecifiers
PropertyInterface -> [PropertyParameterList] ':' Ident
PropertyParameterList -> '[' (IdentList ':' TypeId)/';'... ']'
PropertySpecifiers -> [INDEX ConstExpr]
                      [READ Ident]
                      [WRITE Ident]
                      [STORED (Ident | Constant)]
                      [(DEFAULT ConstExpr) | NODEFAULT]
                      [IMPLEMENTS TypeId]
InterfaceType -> INTERFACE [InterfaceHeritage]
                 [ClassMethodList]
                 [ClassPropertyList]
                 END
InterfaceHeritage -> '(' IdentList ')'
RequiresClause -> REQUIRES IdentList... ';'
ContainsClause -> CONTAINS IdentList... ';'
IdentList -> Ident/','...
QualId -> [UnitId '.'] Ident
TypeId -> [UnitId '.'] <type-identifier>
Ident -> <identifier>
ConstExpr -> <constant-expression>
UnitId -> <unit-identifier>
LabelId -> <label-identifier>
Number -> <number>
String -> <string>
```



색인

기호

- 4-4, 4-6, 4-9, 4-10
" 13-7
4-4
$ 4-4, 4-5
(*, *) 4-5
(,) 4-2, 4-12, 4-14, 5-6, 5-42, 6-2, 6-3, 6-10, 7-2, 10-1
* 4-2, 4-6, 4-10
+ 4-4, 4-6, 4-9, 4-10
, 3-5, 3-6, 4-23, 5-6, 5-21, 5-23, 6-10, 7-17, 9-5, 9-9, 10-6, 13-2
. 3-2, 4-2, 4-13, 5-26, 9-9, 10-5
.dcp 파일 9-11
/ 4-2, 4-6
// 4-5
: 4-2, 4-17, 4-23, 5-21, 5-22, 5-37, 5-42, 6-3, 6-10, 7-17, 7-19, 7-28, 13-2
:= 4-17, 4-26
  명명된 매개 변수 10-12
; 3-2, 3-5, 3-6, 4-16, 4-17, 4-20, 4-25, 5-21, 5-22, 5-37, 6-2, 6-3, 6-10, 7-2, 7-6, 9-5, 9-9, 10-1, 10-4, 10-10
  else 절 앞 4-22
< 4-2, 4-9, 4-10
<= 4-9, 4-10
<> 4-9, 4-10
= 4-2, 4-9, 4-10, 4-17, 5-36, 5-38, 5-39, 5-42, 6-10, 6-16, 10-5
> 4-2, 4-9, 4-10
>= 4-9, 4-10
@ 4-6, 4-12, 5-26, 5-30, 5-43, 7-17
@@ 5-30
@Result 13-9, 13-15
[,] 4-13, 4-14, 5-11, 5-12, 5-17, 5-31, 6-14, 6-19, 7-17, 7-19, 7-20, 10-1, 10-10
^ 4-6, 4-9, 5-19, 5-27
  가변 5-33
  포인터 개요 5-26
_ 4-2
{, } 4-5, 10-1, 10-10

가

가변 5-30 ~ 5-34, 11-12
  가변 타입 5-27, 5-30 ~ 5-34
  가변 타입 배열 5-33
  가변 타입 배열 및 문자열 5-33
  레코드 5-23
  메모리 관리 11-2
  변환 5-30, 5-31 ~ 5-33
  부분 계산 4-8
  연산자 4-6, 5-33
  인터페이스 10-9, 10-10
  초기화 5-31, 5-38
  파일 5-24
  타입 변환 5-31
  해제 5-39
  Automation 11-12
  OleVariant 5-34
  완전한 계산 4-8
가변 개방형 배열 매개변수 6-15, 6-18
가변 매개변수 6-18
가변 부분(레코드) 5-22 ~ 5-24
가변 타입
  메모리 관리 11-11
가변 타입 배열 5-30, 5-33
가시성(클래스 멤버) 7-4
  인터페이스 10-1
가상 메소드 7-9, 7-10
  생성자 7-14
  Automation 7-6
  오버로드 7-12
가상 메소드 테이블 11-10
간접 유닛 참조 3-6 ~ 3-7
값 매개변수 6-10, 6-11, 6-18, 12-1
  개방형 배열 생성자 6-19
값 타입 변환 4-14
값에 의한 전달 (매개 변수) 10-12
값에 의한 전달 (매개변수) 6-11, 12-1
개방형 배열 매개변수 6-14, 6-18
  동적 배열 6-14
개방형 배열 생성자 6-16, 6-18
객체 4-20, 7-1
  'of object' 5-28
  메모리 11-10
  비교 4-11
  파일 5-24
객체 타입 7-4
객체 파일
  루틴 호출 6-7
곱하기 4-6
공백 4-1
공백 문자 4-1
공백(blank) 4-1
공유 객체 파일 2-3, 9-1
  동적으로 로드할 수 있는 9-2
  예외 9-7
  함수 가져오기 6-7
공집합 5-17
관계 연산자 4-10
교집합 4-10
구문
  설명 1-2
구분자 4-1, 4-5
구조 5-21
구조문 4-19
구조 타입 5-16
  가변 5-30
  레코드 5-23
  파일 5-24
구현 섹션 3-3, 3-4, 3-7
  메소드 7-8
  유효 범위 4-28
  uses 절 3-7
굵게 1-2
기본 매개변수 6-10, 6-16 ~ 6-18
  오버로드 6-9, 6-17
  Automation 객체 10-12
  forward 선언문과 인터페이스 선언문 6-18
기본 속성 7-20
  인터페이스 10-1
기본 속성(COM 객체) 5-33
기본 제공 어셈블러 13-1 ~ 13-15
기본 제공된 타입 5-1
기본 타입 5-1, 5-8, 5-17, 5-18, 5-19
기수형 5-3
기호 4-1, 4-2
  어셈블러 13-9
기호 쌍 4-2
긴 문자열 4-9, 5-10, 5-12

동적으로 로드할 수 있는 라이브러리 9-8
레코드 5-23
메모리 관리 11-2, 11-6
파일 5-24

나

나누기 4-6
내부 데이터 형식 11-3 ~ 11-12
내부 블록 4-28
논리 연산자 4-8
논리곱 4-7
논리합 4-7
비트 단위 4-8

다

다중 스레드 애플리케이션 5-39
동적으로 로드할 수 있는 라이브러리 9-7
다중 유닛 참조 3-6 ~ 3-7
다차원 배열 5-18, 5-20, 5-42
다형성 7-9, 7-11, 7-14
단항 연산자 4-6
대/소문자 구분 4-1, 4-2, 6-8
유닛 이름과 파일 4-2
더하기 4-6
포인터 4-9
데스크탑 설정 파일 2-3
데이터 정렬 5-16, 11-8
데이터 형식, 내부 11-3 ~ 11-12
디렉토리 경로
uses 절 3-5
디자인 타임 패키지 9-8
동등 연산자 4-10
동적 메소드 7-9, 7-10
동적 배열 5-19, 11-7
가변 5-30
개방형 배열 매개변수 6-14
다차원 5-20
동적으로 로드할 수 있는 라이브러리 9-8
레코드 5-23
메모리 관리 11-2
비교 5-19
자르기 5-20
파일 5-24
할당 5-19
해제 5-39
동적 변수 5-38
동적으로 로드할 수 있는 라이브러리 9-8
포인터 상수 5-44
동적으로 로드할 수 있는 라이브러리 6-7, 9-1 ~ 9-12
긴 문자열 9-8
동적 배열 9-8
동적 변수 9-8
변수 9-1
작성 9-3
전역 변수 9-6
정적으로 로딩 9-1
예외 9-7

라

라이브러리 검색 경로 3-6
라이브러리에서 루틴 가져오기 9-1
라인 피드 4-5
랩어라운드(순서형) 5-4, 5-5
런타임 패키지 9-8
레이블 4-1, 4-4, 4-18
어셈블러 13-2
레지스터 12-2, 12-3
어셈블러 13-2, 13-8, 13-11, 13-15
집합 저장 11-7
레코드 4-20, 5-21 ~ 5-24
가변 5-30
가변 부분 5-22 ~ 5-24
레코드형 5-21
메모리 11-8
상수 5-43
속성 7-5
유효 범위 4-28, 5-22
로케일 5-13
루틴 6-1 ~ 6-19
내보내기 9-5
표준 8-1 ~ 8-10
루틴 호출 9-1

마

매개 변수
위치 10-12
이름 10-12
Automation 메소드 호출 10-12
매개변수 5-29, 6-2, 6-3, 6-10 ~ 6-18
가변 개방형 배열 6-15
가변적인 수 6-6
값 6-11, 12-1
개방형 배열 6-14
기본값 6-16 ~ 6-18, 10-12
레지스터 12-2
매개변수 목록 6-10
배열 6-10, 6-14
배열 속성 인덱스 7-19
변수(var) 6-11, 12-1
상수 6-12, 12-1
속성 7-17
실제 매개변수 6-18
오버로드 6-6, 6-8, 6-9
전달 12-1
짧은 문자열 6-14
출력(out) 6-12
타입이 지정되지 않은 매개변수 6-13, 6-18
타입이 지정된 매개변수 6-10
파일 6-10
프로그램 제어 12-1
형식 매개변수 6-18
호출 규칙 6-5
register 6-5
멀티바이트 문자 집합 5-13
문자열 처리 루틴 8-7
멤버, 클래스 7-1
가시성 7-4
인터페이스 10-1
몸체(루틴) 6-1
리소스 문자열 5-41
리소스 파일 2-2, 2-3, 3-2
메모리 4-1, 5-2, 5-25, 5-26, 5-31, 5-38, 7-14
공유 메모리 관리자 9-8
관리 11-1 ~ 11-12
동적으로 로드할 수 있는 라이브러리 9-6
오버레이(레코드) 5-23
힙 5-38
메모리 참조 (어셈블러) 13-11
메시지 디스패칭 7-16
메시지 핸들러 7-15
inherited 7-16
오버라이드 7-16
메인 폼 2-6
메소스 7-1, 7-2, 7-8 ~ 7-17
가상 7-6, 7-9, 7-10
구현 7-8
동적 7-9, 7-10
바인딩 7-9
이중 인터페이스 6-5
정적 7-9, 7-10
추상 7-12
클래스 메소드 7-1, 7-25
포인터 4-12, 5-28

호출 규칙 12-3
호출 디스패칭 7-11
생성자 7-13, 12-4
Automation 7-6, 10-12
dispatch 인터페이스 10-11
오버라이드 7-10, 7-11, 10-6
오버로딩 7-12
소멸자 7-14, 12-4
published로 선언 7-5
메소드 오버라이드 7-10,
메소드 지시어
순서 7-8
메소드 지시어의 순서 7-8
메소드 포인터 4-12, 5-28
메소드 확인 절 10-5, 10-6 10-6
숨기기 7-11
메소드 호출 디스패칭 7-11
메타클래스 7-23
명명된 매개 변수 10-12
문 4-1, 4-17 ~ 4-27, 6-1
문자
문자열 리터럴 4-5, 5-5
와이드 5-13, 11-3
타입 5-5, 11-3
포인터 5-27
문자 집합
멀티바이트(MBCS) 5-13
싱글바이트(SBCS) 5-13
파스칼 4-1, 4-2, 4-4
확장 5-13
ANSI 5-5, 5-12, 5-13
문자 연산자 4-9
문자열 4-1, 4-4, 5-44
가변 5-31
가변 개방형 배열 매개변수 6-16
가변 타입 배열 5-33
리터럴 4-4, 5-44
매개변수 6-14
메모리 관리 11-5, 11-6
비교 4-11, 5-11
상수 4-4, 5-44, 13-7
연산자 4-9, 5-15
와이드 5-13, 11-2
와이드 문자열 8-7
인덱스 4-14
타입 5-10 ~ 5-16
Null 종료 5-13 ~ 5-16, 5-27
문자열 연결 4-9
밑줄 4-2

바

바인딩
메소드 7-9
필드 7-7
반환 값(함수)
생성자 7-13
반환 타입(함수) 6-3, 6-4
반환값(함수) 6-3, 6-4
방화벽 예외 6-5
배열 5-3, 5-18 ~ 5-21
'array of const' 6-15
가변 5-30, 5-33
개방형 배열 생성자 6-16, 6-18
다차원 5-18, 5-20
동적 5-19, 5-39, 6-14, 11-7
매개변수 6-10, 6-14
문자 4-5, 5-13, 5-14, 5-17, 5-18
문자 배열 및 문자열 상수 4-5, 5-14, 5-42
인덱스 4-14
정적 5-18, 11-7
지정문 5-21
상수 5-42
PByteArray으로 액세스 5-27
PWordArray으로 액세스 5-27
배열 속성 7-5, 7-19
기본 7-20
저장소 지정자 7-21
dispatch 인터페이스 10-11
범위 검사 5-4, 5-5, 5-9
변수 4-6, 5-37 ~ 5-39
동적 5-38
동적으로 로드할 수 있는 라이브러리 9-1
메모리 관리 11-2
스레드 5-39
전역 5-38, 10-9
절대 주소 5-38
지역 5-38, 6-9
초기화 5-38
파일 8-2
힙 할당 5-38
선언 5-37
변수 타입 변환 4-14, 4-15
변수(var) 매개변수 6-10, 6-11, 6-18, 12-1
복합문 4-19, 4-20
분할작업
애플리케이션 9-8
블록 4-27 ~ 4-28
라이브러리 9-6
외부 및 내부 4-28
유효 범위 4-27 ~ 4-29
프로그램 3-1, 3-2
프로시저 3-4, 6-1, 6-2
함수 3-4, 6-1, 6-3
try...except 7-27, 7-30
try...finally 7-31
비교
객체 4-11
관계 연산자 4-10
동적 배열 5-19
문자열 4-11, 5-11
실수 타입 4-11
압축 문자열 4-11
정수 타입 4-11
클래스 4-11
클래스 참조 타입 4-11
PChar 타입 4-11
변환
가변 5-30, 5-31 ~ 5-33
부등 연산자 4-10
부분 계산 4-7
부분범위 타입 4-7, 4-23, 5-8
부울 연산자 4-7
완전한 계산과 부분 계산 비교 4-7
부울 타입 5-5, 5-6, 11-3
부정 4-7
비트 단위 4-8
부호
타입 변환 4-14
숫자 4-4
비트 단위 연산자, not 4-8
빼기 4-6
포인터 4-9

사

산술 연산자 4-6, 5-4
상속 7-2, 7-3, 7-5
인터페이스 10-2
상수 4-6, 5-39
레코드 5-43
배열 5-42
선언 5-39 ~ 5-44
어셈블러 13-7
타입 호환성 5-40
타입이 지정된 상수 5-42
포인터 5-43
True 5-39
상수 매개변수 6-10, 6-12, 6-18, 12-1
개방형 배열 생성자 6-19

상수 표현식
  기본 매개변수 6-17
  배열 상수 5-42
  변수 초기화 5-38
  부분범위 타입 5-8, 5-9
  상수 선언 5-40, 5-42, 5-43
  열거 타입 5-7
  정의 5-41
  타입 5-40
  case 문 4-23
상호 종속 유닛 3-7
상호 종속 클래스 7-6
새 프로시저 5-19, 5-26, 5-38, 9-8
생성자 7-1, 7-9, 7-13
  클래스 참조 7-23
  호출 규칙 12-4
  예외 7-28, 7-32
서브셋 연산자 4-10
선언 4-1, 4-16, 4-27
  구현 7-8
  메소드 7-8
  속성 7-17, 7-19
  인터페이스 3-4
  정의 7-6, 7-8, 10-4
  클래스 7-2, 7-7, 7-8, 7-17, 10-5
  필드 7-7
  forward 3-4, 7-6, 10-4
선언 정의 7-6, 7-8, 10-4
선언된 타입 5-1
선언문
  변수 5-37
  상수 5-39, 5-42
  정의 6-6
  지역 6-9
  타입 5-36
  패키지 9-9
  프로시저 6-1, 6-2
  함수 6-1, 6-3
  forward 6-6
선행자 5-2
소멸자 7-1, 7-13, 7-14
  호출 규칙 12-4
소스 코드 파일 2-2
속성 7-1, 7-17 ~ 7-22
  기본 7-20
  기본값 10-1
  레코드 7-5
  매개변수 7-17
  배열 7-5, 7-19
  선언 7-17, 7-19
  쓰기 전용 7-19
  액세스 지정자 7-17
  오버라이드 7-6, 7-21
  인터페이스 10-4
  읽기 전용 7-19
속성 오버라이드 7-21
  액세스 지정자 7-22
  Automation 7-6
수퍼셋 연산자 4-10
순환 참조
  유닛 3-7 ~ 3-8
  패키지 9-10
순서 5-2
  열거 타입 5-6, 5-7
순서 감소 5-3, 5-4, 5-5
순서 증가 5-3, 5-4, 5-5
순서 타입 5-2 ~ 5-9
순환문 4-19, 4-25
숫자 4-1, 4-4
  레이블 4-4, 4-18
  타입 5-40
  어셈블러 13-7
스레드 변수 5-39
  패키지 9-9
스택 크기 11-2
스트리밍(데이터) 5-2, 7-5
시스템 유닛 3-1, 3-5, 5-27, 5-31, 5-32, 7-3, 7-28, 8-1, 8-7, 10-2, 10-3, 10-5, 10-10, 11-11
  동적으로 로드할 수 있는 라이브러리 9-6, 9-7
  메모리 관리 11-2
  범위 4-29
  수정 8-1
  uses 절 8-1
실행 파일 2-3
실수(부동 소수점) 연산자 4-6
실수 타입 5-9, 11-4
  비교 4-11
  변환 4-15
  published로 선언 7-5
식별자 4-1, 4-2, 4-3
  전역 및 지역 4-28
  유효 범위 4-27 ~ 4-29
  예외 핸들러 7-29
실제 매개변수 6-18
쓰기 전용 속성 7-19

아

압축 레코드 11-8
압축 문자열 5-18
  비교 4-11
압축 배열 4-5, 4-9, 5-18
애플리케이션 분할작업 9-8
애플리케이션 변수 2-6
액세스 지정자 7-1, 7-17
  배열 속성 7-19
  오버라이드 7-22
  오버로딩 7-13, 7-18
  호출 규칙 6-5, 7-18
  Automation 7-6
  index 지정자 7-21
어셈블리어
  기본 제공 어셈블러 13-1 ~ 13-15
  어셈블러 루틴 13-14
  오브젝트 파스칼 13-1, 13-4, 13-5, 13-6, 13-9, 13-11, 13-13
  외부 루틴 6-7
역참조 연산자 4-9, 5-19
  가변 5-33
  포인터 개요 5-26
연산 코드(어셈블러) 13-2
연산자 4-6 ~ 4-13
  어셈블러 13-13
  우선 순위 4-12, 7-25
  클래스 7-24
연산자 우선 순위 4-12, 7-25
열거 타입 5-6 ~ 5-8, 11-3
  익명 값 5-8, 7-5
  published로 선언 7-5
영숫자 4-1, 4-2
예제 프로그램 2-3 ~ 2-5
예약어 4-1, 4-2, 4-3
  목록 4-3
  어셈블러 13-5
예외 4-19, 7-13, 7-15, 7-26 ~ 7-32
  동적으로 로드할 수 있는 라이브러리 9-6, 9-7
  발생 7-27
  생성자 7-28, 7-32
  선언 7-27
  소멸 7-28, 7-29
  예외 전달 7-29, 7-31, 7-32
  재발생 7-30
  중첩 7-31
  처리 7-27, 7-28, 7-29, 7-30, 7-32
  초기화 섹션 7-28
  파일 I/O 8-3
  표준 루틴 7-32
  표준 예외 7-32
예외 클래스 7-27, 7-32
예외 핸들러 7-26, 7-28
  식별자 7-29
오른쪽으로 시프트 (비트 단위 연산자) 4-8

오버로드된 메소드 7-12
  액세스 지정자 7-13, 7-18
  published로 선언 7-5
오버로드된 프로시저 및 함수 6-6, 6-8
  기본 매개변수 6-9, 6-17
  동적으로 로드할 수 있는 라이브러리 9-5
  forward 선언문 6-9
와이드 문자 및 문자열 5-13
  메모리 관리 11-2
  표준 루틴 8-7
완료 섹션 3-3, 3-5, 12-4
완전한 계산 4-7
왼쪽으로 시프트 (비트 단위 연산자) 4-8
유닛 2-1, 3-1 ~ 3-8
  구문 3-3 ~ 3-8, 4-17
  유효 범위 4-28
유닛코드 5-5, 5-13
유닛 파일 3-1, 3-3
  대/소문자 구분 4-2
유효 범위 4-27 ~ 4-29
  레코드 5-22
  형 식별자 5-37
유효 범위(scope)
  클래스 7-3
이름
  내보낸 루틴 9-5
  식별자 4-16
  유닛 3-3, 3-6
  패키지 9-9
  프로그램 3-1, 3-2
  함수 6-3, 6-4
이름 충돌 3-6, 4-28
이미 정의된 타입 5-1
이벤트 2-7, 7-5
이벤트 핸들러 2-7, 7-5
이중 인터페이스 10-3, 10-13
  메소드 6-5
이탤릭체 1-2
이항 연산자 4-6
  외부 블록 4-28
익명 값(열거 타입) 5-8, 7-5
인덱스 4-14
  가변 타입 배열 5-33
  문자열 5-11
  문자열 가변 5-31
  배열 5-18, 5-19, 5-20
  배열 속성 7-19
  var 매개변수 5-33, 6-12
인덱스 지정자 (Windows만 해당) 9-5
인라인 어셈블러 코드 13-1 ~ 13-15
인용 문자열 4-4, 5-44
  어셈블러 13-7
인터페이스 7-2, 10-1 ~ 10-13
  구현 10-4 ~ 10-7
  레코드 5-23
  메모리 관리 11-2
  메소드 확인 절 10-5, 10-6
  속성 10-1, 10-4, 10-6
  액세스 10-8 ~ 10-10
  이중 인터페이스 10-13
  인터페이스 참조 10-8 ~ 10-10
  인터페이스 타입 10-1 ~ 10-4
  위임 10-6
  쿼리 10-10
  타입 변환 10-10
  해제 5-39
  호출 규칙 10-3
  호환성 10-9
  Automation 10-10
  dispatch 인터페이스 타입 10-10
  GUID 10-1, 10-3, 10-10
인터페이스 구현 오버라이드 10-6
인터페이스 구현 숨기기 10-6
인터페이스 선언 3-4
  기본 매개변수 6-18
인터페이스 섹션 3-3, 3-4, 3-7
  메소드 7-8
  유효 범위 4-28
  forward 선언문 6-6
  uses 절 3-7
읽기 전용 속성 7-19
일반문 4-17
일반 타입 5-1, 5-2
위임(인터페이스 구현) 10-6
위임된 인터페이스 10-7
위치 매개변수 10-12

자

자손 7-3, 7-5
장치 드라이버, 텍스트 파일 8-4
장치 함수 8-4, 8-5
저장소 지정자 7-21
  배열 속성 7-21
전역 변수 5-38
  동적으로 로드할 수 있는 라이브러리 9-6
  메모리 관리 11-2
  인터페이스 10-9
전역 식별자 4-28
절대 주소 5-38
절대 표현식 (어셈블러) 13-11
재귀 프로시저 및 함수 호출 6-4
재배치 표현식 (어셈블러) 13-11
점프 명령(어셈블러) 13-3
정렬(데이터) 5-16, 11-8
정수 연산자 4-6
정수형 4-7, 5-3, 5-4
  데이터 형식 11-3
  비교 4-11
  변환 4-15
  상수 5-40
정의 선언 6-6
정적 메소드 7-9, 7-10
정적 배열 5-18, 11-7
  가변 5-30
정적으로 로드된 라이브러리 9-1
제어 문자 4-1, 4-4
제어 문자열 4-4
제어(프로그램) 6-18, 12-1 ~ 12-5
조건문 4-19
조상 7-3
주석 4-1, 4-5
주소 연산자 4-12, 5-26, 5-30, 5-43
  속성 7-17
지시어 4-1, 4-3
  목록 4-3
  순서 7-8
  어셈블러 13-3
  컴파일러 3-2, 4-5
지역 변수 5-38, 6-9
  메모리 관리 11-2
지역 식별자 4-28
집합
  가변 5-30
  공집합 5-17
  메모리 11-7
  연산자 4-10
  집합 생성자 4-13
  집합 타입 5-17
집합형
  published로 선언 7-5
종료 프로시저 9-6, 12-4 ~ 12-5

패키지 12-4
종료(exit) 프로시저 12-4 ~ 12-5
종속 관계
유닛 3-6 ~ 3-8
줄 끝 문자 4-1, 8-3
중첩 루틴 5-29, 6-9
중첩 예외 7-31
중첩된 조건문 4-22
즉시 값(어셈블러) 13-11
짧은 문자열 5-3, 5-10, 5-12

차

차집합 4-10
참조에 의한 전달 (매개 변수) 10-12
참조에 의한 전달 (매개변수) 6-11, 6-12, 12-1
참조 카운트 11-6, 11-7
참조 카운팅 5-12, 10-9
철저한 타입 지정 5-1
초기화
가변 5-31, 5-38
객체 7-13
동적으로 로드할 수 있는 라이브러리 9-6
변수 5-38
유닛 3-4
파일 5-38
초기화 섹션 3-3, 3-4
예외 7-28
추상 메소드 7-12

카

캐리지 리턴 4-1, 4-5
컨텍스트가 많은 도움말 (오류 처리) 7-32
컴파일러 2-2, 2-3, 2-5, 3-1
명령줄 2-3 ~ 2-5
지시어 3-2, 4-5
패키지 9-11
콘솔 애플리케이션 2-3, 8-3
쿼리(인터페이스) 10-10
클라이언트 3-4
클래스 7-1 ~ 7-32
가변 5-30
메모리 11-10
메타클래스 7-23
비교 4-11
연산자 4-11, 7-24
유효 범위 4-28
클래스 메소드 7-1, 7-25
클래스 참조 7-23
클래스 타입 7-1, 7-2
클래스 타입 선언 7-2, 7-4, 7-6, 7-7, 7-8, 7-17, 10-5
파일 5-24
호환성 7-3, 10-9
클래스 멤버 숨기기 7-8, 7-11
reintroduce 7-12
클래스 참조 타입 7-23
가변 5-30
메모리 11-11
비교 4-11
생성자 7-23
클래시스 유닛 7-9, 7-23

타

타입 5-1 ~ 5-37
가변 5-30 ~ 5-34
객체 7-4
구조 5-16
기본 5-1
기본 제공 5-1
내부 데이터 형식 11-3 ~ 11-12
레코드 5-21 ~ 5-24, 11-8
문자 5-5, 11-3
문자열 5-10 ~ 5-16, 11-5, 11-6
배열 5-18 ~ 5-21, 11-7
분류 5-1
부분범위 5-8
부울 5-5, 11-3
사용자 정의 5-1
상수 5-40
선언 5-1, 5-36
순서 5-2 ~ 5-9
실수 5-9, 11-4
어셈블러 13-12
열거 5-6 ~ 5-8
열거 타입 11-3
예외 7-27
유효 범위 5-37
이미 정의된 5-1
인터페이스 10-1 ~ 10-4
일반 5-1, 5-2
절차적 11-10
정수 5-3, 11-3
집합 5-17, 11-7
클래스 7-1, 7-2, 7-4, 7-6, 7-7, 7-8, 7-17, 11-10
클래스 참조 7-23, 11-11
타입 구분 5-34
파일 5-24, 11-8
포인터 5-27
프로시저 타입 5-28 ~ 5-30
할당 호환 5-36
호환성 5-17, 5-29, 5-35, 5-36
automated 가능 7-6
Automation 가능 10-11
dispatch 인터페이스 10-10
타입 검사(객체) 7-24
타입 식별자 5-2
타입 변환 4-14 ~ 4-16, 7-8
가변 5-31
상수 선언문에서 5-40
열거 타입 5-8
인터페이스 10-10
타입이 지정되지 않은 매개변수 6-13
확인 7-25, 10-10
타입이 지정되지 않은 매개변수 6-13
타입이 지정되지 않은 파일 5-25, 8-2, 8-4
터보 어셈블러 13-1, 13-4
텍스트 파일 8-2, 8-3
텍스트 파일 장치 드라이버 8-4
텍스트 파일 형식 5-24
텍스트 형식 5-24, 8-3
토큰 4-1
특수 기호 4-1, 4-2

파

파일
가변 5-30
매개변수 6-10
메모리 11-8
생성 2-2, 2-3, 9-9, 9-11
소스 코드 2-2
초기화 5-38
타입이 지정되지 않은 파일 5-25, 8-2
타입이 지정된 8-4
타입이 지정된 파일 5-24, 8-2
텍스트 8-2, 8-3
파일 타입 8-2
파일 타입 5-24
파일 변수 8-2
파일 I/O 8-1 ~ 8-6
예외 8-3

패키지 9-8 ~ 9-12
  동적으로 로딩 9-8
  선언 9-9
  스레드 변수 9-9
  정적으로 로딩 9-8
  컴파일 9-11
  컴파일러 스위치 9-12
  컴파일러 지시어 9-11
  uses 절 9-8
패키지 파일 2-2, 2-3, 9-8, 9-9, 9-11
포인터 5-25 ~ 5-27
  가변 5-30
  가변 개방형 배열 매개변수 6-16
  개요 5-25
  긴 문자열 5-16
  레코드 5-23
  메모리 11-5
  메소드 포인터 5-28
  문자 5-27
  산술 4-9
  상수 5-43
  연산자 4-9
  파일 5-24
  포인터 타입 4-12, 5-26, 5-27, 11-5
  표준형 5-27
  프로시저 타입 4-12, 5-28 ~ 5-30
  함수 4-12, 5-28
  nil 5-27, 11-5
  Null 종료 문자열 5-14, 5-16
  var 매개변수 6-12
포인터 타입 5-25, 5-26, 5-27, 11-5
폼 2-2
폼 파일 2-2, 2-6, 3-1, 7-5, 7-21
표기법 1-2
표준 루틴 8-1 ~ 8-10
  와이드 문자 문자열 8-7
  Null 종료 문자열 8-6, 8-7
표현식 4-1, 4-5
  어셈블러 13-6 ~ 13-14
프로그램 2-1 ~ 2-5, 3-1 ~ 3-8
  구문 3-1 ~ 3-3
  예제 2-3 ~ 2-5
프로그램 제어 6-18, 12-1 ~ 12-5
프로시저 3-4, 6-1 ~ 6-19
  선언 6-2, 6-6
  어셈블러 13-14
  오버로드 6-6, 6-8
  외부에서 호출 6-6
  중첩 5-29, 6-9
  포인터 4-12, 5-28
  프로시저 호출 4-18, 6-1, 6-2, 6-18 ~ 6-19
프로시저 및 함수 재귀 호출 6-1
프로시저 타입 4-15, 5-28 ~ 5-30
  동적으로 로드할 수 있는 라이브러리 호출 9-2
  루틴 호출 5-30
  메모리 11-10
  지정문 5-30
호환성 5-29
프로시저 포인터 4-12, 5-28
프로젝트 2-6, 3-6
프로젝트 옵션 파일 2-2
프로젝트 파일 2-2, 3-1, 3-2, 3-6
프로토타입 6-1
피연산자 4-6
필드 5-21 ~ 5-24, 7-1, 7-7
  published로 선언 7-5

하

한정된 식별자 4-2, 4-29, 5-22
  포인터 5-26
  타입 변환 4-15
  Self 사용 7-9
할당 호환성 10-9
할당 호환 5-36
할당 호환성 7-3
할당되지 않음(가변) 5-33
할당문 4-17
  타입 변환 4-15
함수 3-4, 6-1 ~ 6-19
  레지스터에 값 반환 12-3, 13-15
  반환 형식 6-3, 6-4
  반환값 6-3, 6-4
  선언 6-3, 6-6
  어셈블러 13-14
  오버로드 6-6, 6-8
  외부에서 호출 6-6
  중첩 5-29, 6-9
  포인터 4-12, 5-28
  함수 호출 4-13, 4-18, 6-1, 6-18 ~ 6-19
합집합 4-10
형식 매개변수 6-18
헤더
  루틴 6-1
  유닛 3-3
  프로그램 2-1, 3-1, 3-2
호출 규칙 5-29, 6-5, 12-1
  공유 라이브러리 9-4
  메소드 12-3
  액세스 지정자 6-5, 7-18
  인터페이스 10-3, 10-7
  varargs 지시어 6-6
확인된 타입 변환
  객체 7-25
  인터페이스 10-10
확장 구문 4-5, 4-18, 5-14, 6-1, 6-4
후행자 5-2, 4-4, 9-9, 10-1, 10-10
힌트 지시어 4-17
힙 메모리 5-38, 11-2

숫자

16비트 애플리케이션 (역 호환성) 6-5
16진수 4-4

A

$A 지시어 11-8
absolute(지시어) 5-38
Add 메소드(TCollection) 7-9
Addr 함수 5-26
_AddRef 메소드 10-2, 10-5, 10-9
alignment (data) *내부 데이터 형식 참조*
AllocMemCount 변수 11-2
AllocMemSize 변수 11-2
ampersand 기호 참조 4-7, 4-8
ANSI 문자 5-5, 5-12, 5-13
AnsiChar 타입 5-5, 5-11, 5-13, 5-27, 11-3
AnsiString type *긴 문자열 참조*
AnsiString 타입 5-10, 5-12, 5-13, 5-15, 5-27
  가변 타입 배열 5-33
  메모리 관리 11-6
Append 프로시저 8-2, 8-3, 8-5, 8-6
Application 변수 3-2
as 4-11, 7-24, 7-25, 10-10
ASCII 4-1, 4-4, 5-13
asm 문 13-1, 13-14
assembler(지시어) 6-6

assembler(지시어) 13-1
Assert 프로시저 7-26
assertions 7-26
Assign 프로시저
  사용자 지정 8-4
Assigned 함수 5-30, 10-9
AssignFile 프로시저 8-2, 8-3, 8-6
asterisk 기호 *참조*
at(예약어) 7-27
at-sign @연산자
automated 가능 형식 7-6
automated 클래스 멤버 7-4, 7-6
Automation 7-6, 10-10 ~ 10-13
  가변 및 11-12
  메소드 호출 10-12
  이중 인터페이스 10-13
  참조COM
Automation 가능 형식 10-11

B

$B 지시어 4-8
begin(예약어) 3-2, 4-20, 6-2, 6-3
BlockRead 프로시저 8-4
BlockWrite 프로시저 8-4
BORLANDMM.DLL 9-8
braces 기호 *참조*
brackets 기호 *참조*
Break 프로시저 4-25
  예외 핸들러 7-29
Break 프로시저
  try...finally 블록 7-32
BSTR 타입 (COM) 5-13
Byte 타입 5-4, 11-3
  어셈블러 13-12
ByteBool 타입 5-6, 11-3

C

C++ 6-6, 10-1, 11-10
caret 기호 *참조*
case 문 4-23
case(예약어) 4-23, 5-22
-cc 컴파일러 스위치 8-3
cdecl(호출 규칙) 6-5, 12-2
  생성자 및 소멸자 12-4
  Self 12-3
  varargs 6-6
Char 타입 5-5, 5-13, 5-27, 11-3
Chr 함수 5-5
circumflex 기호 *참조*
ClassParent 메소드 7-24
ClassType 메소드 7-24
Close 함수 8-4, 8-6
CloseFile 프로시저 8-6
CloseFile 함수 8-5
CLX 1-2
CmdLine 변수 9-7
colon 기호 *참조*
COM 10-4
  가변 5-30, 5-33
  가변 타입 11-11
  인터페이스 10-2, 10-10 ~ 10-13
  참조 Automation
  out 매개변수 6-12
COM 오류 처리 6-5
comma 기호 *참조*
ComObj 유닛 7-6, 10-11
Comp 타입 5-9, 5-10, 11-5
compile-time binding 정적 메소드 *참조*components, of classes 숫자 *참조*
const(예약어) 5-39, 5-42, 6-10, 6-12, 6-15, 12-1
constants
  numeric 숫자 *참조*
contains 절 9-9, 9-10
Continue 프로시저 4-25
  예외 핸들러 7-29
  try...finally 블록 7-32
conversion
  타입 변환 *참조*
Copy 함수 5-20
copy-on-write 의미론 5-12
CORBA
  가변 5-30
  인터페이스 10-3
  out 매개변수 6-12
CPU 레지스터 *참조*
Create 메소드 7-13
Currency 타입 5-9, 5-10, 5-27, 11-5

D

data types 타입 *참조*
.dcp 파일 2-3
.dcu 파일 2-3, 3-7, 9-10, 9-11
Dec 프로시저 5-3, 5-4
default 지정자 7-6, 7-17, 7-21
default(지시어) 7-20, 10-11
DefaultHandler 메소드 7-16, 7-17
DefWindowProc 함수 7-16
$DENYPACKAGEUNIT 지시어 9-11
deprecated 지시어 4-17
$DESIGNONLY 지시어 9-11
.desk 파일 2-3
Destroy 메소드 7-13, 7-15, 7-29
.dfm 파일 2-2, 2-7, 7-5
directives
  예약어 *참조*
Dispatch 메소드 7-16
dispatch 인터페이스 타입 10-10
dispid(지시어) 7-6, 10-2, 10-11
dispinterface 10-10
dispinterface(예약어) 10-2
Dispose 프로시저 5-19, 5-38, 7-4, 9-8, 11-1, 11-2
div 4-6
dlclose 9-2
DLL 9-1 ~ 9-8
  긴 문자열 9-8
  다음에서 루틴 호출 6-7
  다중 스레드 9-7
  동적 배열 9-8
  동적 변수 9-8
  동적으로 로딩 9-2
  변수 9-1
  예외 9-7
  작성 9-3
  전역 변수 9-6
  정적으로 로딩 9-1
.DLL 파일 6-7, 9-1
DLL_PROCESS_DETACH 9-7
DLL_THREAD_ATTACH 9-7
DLL_THREAD_DETACH 9-7
dlopen 9-2
dlsym 9-2
do(예약어) 4-20, 4-25, 4-26, 7-28
.dof 파일 2-2
dollar 기호 *참조*
Double 타입 5-9, 11-5
downto(예약어) 4-26
.dpk 파일 2-2, 9-11
.dpr 파일 2-2, 3-1, 3-6
.dpu 파일 2-3, 3-7, 9-10, 9-11
.drc 파일 2-3
.dsk 파일 2-3

DWORD 타입
(어셈블러) 13-12
dynamic-link libraries DLL *참조*

E

E (숫자로) 4-4
EAssertionFailed 7-26
else(예약어) 4-22, 4-24, 7-28
end(예약어) 3-2, 4-20, 4-23, 5-21, 5-22, 6-2, 6-3, 7-2, 7-28, 7-31, 9-9, 10-1, 10-10, 13-1
Eof 함수 8-5
Eoln 함수 8-5
error handling 예외 *참조*
ErrorAddr 변수 12-5
EStackOverflow 예외 11-2
EVariantError 예외 5-32
except(예약어) 7-28
ExceptAddr 함수 7-32
Exception 클래스 7-32
ExceptionInformation 변수 9-7
ExceptObject 함수 7-32
Exit 프로시저 6-1
  예외 핸들러 7-29
  try...finally 블록 7-32
ExitCode 변수 9-6, 12-5
ExitProc 변수 9-6, 12-4
export(지시어) 6-5
exports 절 4-27, 9-5
  오버로드된 루틴 9-5
Extended형 4-7, 5-9, 5-10, 5-27, 11-5
external(지시어) 6-6
external(지시어) 9-1, 9-2

F

False 5-6, 11-3
far(지시어) 6-5
fields
  레코드, 클래스 *참조*
file(예약어) 5-24
FilePos 함수 8-2
FileSize 함수 8-2
Finalize 프로시저 5-19
finally(예약어) 7-31
floating-point types 실수 타입 *참조*
Flush 함수 8-4, 8-5
for 문 4-19, 4-25, 4-26
forward 선언
  루틴 3-4
  인터페이스 10-4
  클래스 7-6
Forward 선언문
  루틴 6-6
forward 선언문
  기본 매개변수 6-18
  오버로드 6-9
Free 메소드 7-15
FreeLibrary 함수 9-2
FreeMem 프로시저 5-38, 9-8, 11-1, 11-2

G

$G 지시어 9-11
-$G- 컴파일러 스위치 9-12
GetHeapStatus 함수 11-2
GetMem 프로시저 5-26, 5-38, 9-8, 11-1, 11-2
GetMemoryManager 프로시저 11-2
GetProcAddress 함수 9-2
getter read 지시자 *참조*
globally unique identifiers GUID *참조*
goto 문 4-18
grammar (formal) A-1 ~ A-6
greater-than 기호 *참조*
GUID 10-1, 10-3, 10-10
  생성 10-3

H

$H 지시어 5-11, 6-14
Halt 프로시저 12-4, 12-5
Hello world! 2-3
HelpContext 속성 7-32
hiding class members
  오버로드된 메소드 *참조*
High 함수 5-3, 5-4, 5-12, 5-18, 5-20, 6-15
HInstance 변수 9-7
$HINTS 지시어 4-17

I

$I 지시어 8-3
IDE
  Delphi *참조*
IDispatch 10-9, 10-10
  이중 인터페이스 10-13
if...then 문 4-22
  중첩 4-22
IInterface 10-2
implements(지시어) 7-22, 10-6
$IMPLICITBUILD 지시어 9-11
$IMPORTEDDATA 지시어 9-11
in(예약어) 4-10, 5-17, 5-33, 9-9
Inc 프로시저 5-3, 5-4
index 지정자 7-6, 7-17, 7-20
index(지시어) 6-7
inherited(예약어) 7-9, 7-13
  메시지 핸들러 7-16
  호출 규칙 12-4
InheritsFrom 메소드 7-24
Initialize 프로시저 5-38
inline(예약어) 13-1
InOut 함수 8-4, 8-5
Input 변수 8-3
input 파일I/O *참조*
input
(프로그램 매개변수) 3-2
Int64형 4-7, 5-3, 5-4, 5-10, 11-3
  가변 5-30
  표준 함수 및 프로시저 5-4
integrated development environment IDE *참조*
IntToHex 함수 5-4
IntToStr 함수 5-4
Invoke 메소드 10-10
IOResult 함수 8-3, 8-4
is 4-11, 5-33, 7-24
IsLibrary 변수 9-7
IUnknown 10-2, 10-5, 10-9, 10-13

J

$J 지시어 5-42
Java 10-1

K

.kof 파일 2-2

L

-$LE- 컴파일러 스위치 9-12
Length 함수 5-11, 5-18, 5-20
less-than 기호 *참조*
libraries DLL 또는 동적으로 로드된 라이브러리 *참조*
library (예약어) 9-3
library 지시어 4-17

-$LN- 컴파일러 스위치 9-12
LoadLibrary 함수 9-2
local 지시어 (Linux에서만) 9-4
LongBool 타입 5-6, 11-3
Longint 타입 5-4, 11-3
Longword 타입 5-4, 11-3
Low 함수 5-3, 5-4, 5-12, 5-18, 5-20, 6-15
-$LU- 컴파일러 스위치 9-12

M

$M 지시어 7-4, 7-6
$MAXSTACKSIZE 지시어 11-2
members 집합 *참조*
Message 속성 7-32
message(지시어) 7-15
  인터페이스 10-7
Messages 유닛 7-15
$MINSTACKSIZE 지시어 11-2
minus 기호 *참조*
mod 4-6
modules 유닛 *참조*

N

name(지시어) 6-7, 6-8
name(지시어) 9-5
names
  식별자 *참조*
near(지시어) 6-5
New 프로시저 7-4, 11-1, 11-2
nil 5-27, 5-30, 5-39, 11-5
nodefault 지정자 7-6, 7-17, 7-21
not 4-6, 4-7
Null (가변) 5-31, 5-32
Null 문자 5-13, 11-6, 11-7, 11-9
Null 문자열 4-4
Null 종료 문자열 5-13 ~ 5-16, 5-27, 11-6, 11-7
  표준 루틴 8-6, 8-7
  Pascal 문자열과 혼합 5-15
Null(가변) 5-33

O

Object Inspector 7-5
Object Inspector (Delphi) 7-5
object interfaces 인터페이스, COM, CORBA *참조*
objects
  클래스 *참조*
of object (메소드 포인터) 5-28
of(예약어) 4-23, 5-17, 5-19, 5-24, 5-28, 6-14, 6-15, 7-23
Ole Automation 5-34
OleVariant 5-34
OleVariant 타입 5-27, 5-34
on(예약어) 7-28
Open 함수 8-4, 8-5
OpenString 6-14
or 4-7, 4-8
Ord 함수 5-3, 5-4
out(출력) 매개변수 6-10, 6-12, 6-18
out(예약어) 6-10, 6-12
OutlineError 7-32
Output 변수 8-3
output 파일I/O *참조*
output (프로그램 매개변수) 3-2

P

$P 지시어 6-14
packed(예약어) 5-16, 11-8
PAnsiChar 타입 5-13, 5-27
PAnsiString 타입 5-27
parameters
  오버로드된 프로시저 및 함수
parentheses 기호 *참조*
.pas 파일 2-3, 3-1, 3-3, 3-7
pascal(호출 규칙) 6-5, 12-2
  생성자 및 소멸자 12-4
  Self 12-3
PByteArray 타입 5-27
PChar 타입 4-5, 4-9, 5-13, 5-14, 5-15, 5-27, 5-44
  비교 4-11
PCurrency 타입 5-27
PDouble 타입 5-27
period 기호 *참조*
PExtended 타입 5-27
PGUID 10-3
PInteger 타입 5-27
platform 지시어 4-17
plus 기호 *참조*
POleVariant 타입 5-27
pound 기호 *참조*
Pred 함수 5-3, 5-4
private 클래스 멤버 7-4, 7-5
program(예약어) 3-2
Project Manager 2-1
protected 클래스 멤버 7-4, 7-5
PShortString 타입 5-27
PSingle 타입 5-27
PString 타입 5-27
PTextBuf 타입 5-27
Ptr 함수 5-26
public 식별자 (인터페이스 섹션) 3-4
public 클래스 멤버 7-4, 7-5
published 클래스 멤버 7-4, 7-5
  $M 지시어 7-6
  제한 7-5
PVariant 타입 5-27
PVarRec 타입 5-27
PWideChar 타입 5-13, 5-14, 5-27
PWideString형 5-27
PWordArray형 5-27

Q

QueryInterface 메소드 10-2, 10-5, 10-10
quotation marks 기호 *참조*
QWORD 타입 (어셈블러) 13-13

R

raise(예약어) 4-19, 7-27, 7-29, 7-30
read 지정자 7-6, 7-17
  객체 인터페이스 10-1, 10-4, 10-7
  배열 속성 7-19
  오버로딩 7-13, 7-18
  index 지정자 7-20
Read 프로시저 8-2, 8-3, 8-4, 8-5, 8-6
Readln 프로시저 8-5, 8-6
readonly(지시어) 10-2, 10-11
Real 타입 5-10
Real48 타입 5-9, 5-10, 7-5, 11-4
$REALCOMPATIBILITY 지시어 5-10
ReallocMem 프로시저 5-38, 11-1
register 6-5

register(호출 규칙) 6-5, 7-6, 7-13, 7-14, 12-2
  동적으로 로드할 수 있는 라이브러리 9-4
  생성자 및 소멸자 12-4
  인터페이스 10-3, 10-7
  Self 12-3
reintroduce(지시어) 7-12
_Release 메소드 10-2, 10-5, 10-9
Rename 프로시저 8-6
repeat 문 4-19, 4-25
requires 절 9-8, 9-9, 9-10
.RES 파일 2-2, 3-2
reserved words
  지시어 *참조*
Reset 프로시저 8-2, 8-3, 8-4, 8-5, 8-6
resident(지시어) 9-5
resourcestring (예약어) 5-41
Result 변수 6-3, 6-4
RET 명령 13-3
Rewrite 프로시저 8-2, 8-3, 8-4, 8-5, 8-6
Round 함수 5-4
routines
  함수, 프로시저 *참조*
RTTI 7-5, 7-12, 7-21
$RUNONLY 지시어 9-11
runtime binding 동저 메소드, 가상 메소드 *참조* runtime type informatiom RTTI *참조*

S

$S 지시어 11-2
safecall(호출 규칙) 6-5, 12-2
  생성자 및 소멸자 12-4
  이중 인터페이스 10-13
  인터페이스 10-3
  Self 12-3
Seek 프로시저 8-2
SeekEof 함수 8-5
SeekEoln 함수 8-5
Self 7-9
  클래스 메소드 7-25
  호출 규칙 12-3
semicolon 기호 *참조*
SetLength 프로시저 5-11, 5-16, 5-19, 5-20, 6-15
SetMemoryManager 프로시저 11-2
SetString 프로시저 5-16
setter 지정자 Write*참조*
ShareMem 유닛 9-8
shl 4-8
Shortint 타입 5-4, 11-3
ShortString 타입 5-10, 5-12, 5-27, 11-5
  가변 타입 배열 5-33
  매개변수 6-14
ShowException 프로시저 7-32
shr 4-8
Single 타입 5-9, 11-4
SizeOf 함수 5-2, 5-5, 6-15
slash 기호 *참조*
Smallint 타입 5-4, 11-3
.so 파일 9-1
stdcall(호출 규칙) 6-5, 12-2
  공유 라이브러리 9-4
  인터페이스 10-3
  생성자 및 소멸자 12-4
  Self 12-3
stored 지정자 7-6, 7-17
Str 프로시저 8-6
StrAlloc 함수 5-38
StrDispose 프로시저 5-38
string
  문자집합 *참조*
  handling 표준루틴, Null종료 문자열 *참조*
string(예약어) 5-11
StringToWideChar 함수 8-7
StrToInt64 함수 5-4
StrToInt64Def 함수 5-4
StrUpper 함수 5-15
Succ 함수 5-3, 5-4
symbols
  I-1의 특수기호 및 기호 *참조*
syntax
  formal A-1 ~ A-6
System 유닛 6-16
SysUtils 유닛 3-5, 5-27, 6-10, 6-16, 7-26, 7-27, 7-28, 7-32
  동적으로 로드할 수 있는 라이브러리 9-8
  uses 절 8-1

T

$T 지시어 4-12
tag(레코드) 5-23
TAggregatedObject 10-7
TBYTE 타입 (어셈블러) 13-13
TByteArray 타입 5-27
TClass 7-3, 7-23, 7-24
TCollection 7-23
  Add 메소드 7-9
TCollectionItem 7-23
TDateTime 5-32
TGUID 10-3
then (예약어) 4-22
threadvar 5-39
TInterfacedObject 10-2, 10-5
to(예약어) 4-26
TObject 7-3, 7-16, 7-24
TPersistent 7-6
True 5-6, 11-3
True 상수 5-39
Trunc 함수 5-4
try...except 문 4-19, 7-27, 7-28
try...finally 문 4-19, 7-31
TTextBuf 타입 5-27
TTextRec 타입 5-27
TVarData 5-31, 11-11
TVarRec 5-27
TVarRec 타입 6-16
TWordArray 5-27
Type Library 편집기 10-3

U

UCS-2 5-13
UCS-4 5-13
Unassigned(가변) 5-31, 5-32
UniqueString 프로시저 5-16
until(예약어) 4-25
UpCase 함수 5-11
uses 절 2-1, 3-1, 3-2, 3-4, 3-5 ~ 3-8
  구문 3-5, 3-6
  시스템 유닛 8-1
  인터페이스 섹션 3-7
  ShareMem 9-8
  SysUtils 유닛 8-1

V

Val 프로시저 8-6
var(예약어) 5-37, 6-10, 6-11, 12-1
varargs(지시어) 6-6
VarArrayCreate 함수 5-33
VarArrayDimCount 함수 5-33
VarArrayHighBound 함수 5-33
VarArrayLock 함수 5-33, 10-12

VarArrayLowBound 함수 5-33
VarArrayOf 함수 5-33
VarArrayRedim 함수 5-33
VarArrayRef 함수 5-33
VarArrayUnlock 프로시저 5-33, 10-12
VarAsType 함수 5-31
VarCast 프로시저 5-31
varOleString 상수 5-33
varString 상수 5-33
VarType 함수 5-31
varTypeMask 상수 5-31
VCL 1-2
virgule 기호 *참조*
VirtualAlloc 함수 11-1
VirtualFree 함수 11-1
Visual Component Library VCL *참조*
VMT 11-10

W

$WARNINGS 지시어 4-17
$WEAKPACKAGEUNIT 지시어 9-11
while 문 4-19, 4-25
WideChar 타입 4-9, 5-5, 5-11, 5-13, 5-27, 11-3
WideCharLenToString 함수 8-7
WideCharToString 함수 8-7
WideString 타입 5-10, 5-13, 5-27
  메모리 관리 11-6
Windows 7-16
  가변 타입 11-11
  메모리 관리 11-2
  메시지 7-15
Windows 유닛 9-2
with 문 4-19, 4-20, 5-22
Word 타입 5-4, 11-3
  어셈블러 13-12
WordBool 타입 5-6, 11-3
write 지정자 7-6, 7-17
  객체 인터페이스 10-1, 10-4
  배열 속성 7-19
  오버로딩 7-13, 7-18
  index 지정자 7-20
Write 프로시저 8-2, 8-3, 8-4, 8-5, 8-6
write 프로시저 5-3
Writeln 프로시저 2-4, 8-5, 8-6
writeonly(지시어) 10-2, 10-11

X

$X 지시어 5-14, 6-1, 6-4
$X 지시어 4-5, 4-18
.xfm 파일 2-2, 2-7, 7-5
xor 4-7, 4-8

Z

$Z 지시어 11-3
-$Z- 컴파일러 스위치 9-12