Object Pascal

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001
www.borland.com

Borland Software Corporation は，本書に記載されているアプリケーションに対する特許を取得または申請している場合があります。本書の提供は，これらの特許に関する権利を付与することを意味するものではありません。

Borland のすべてのブランドおよび製品名は，米国 Borland Software Corporation の米国における商標または登録商標です。その他のブランドまたは製品名は，その版権所有者の商標または登録商標です。

COPYRIGHT © 1983, 2001 Borland Software Corporation. All rights reserved.

目次

第 1 章 はじめに 1-1

このマニュアルの内容 1-1
  Object Pascal 1-1
  表記規則 1-2
その他の資料 1-2
ソフトウェアの登録とテクニカルサポート 1-2

第 I 部 言語の基本説明

第 2 章 概要 2-1

プログラムの構成 2-1
  Pascal ソースファイル 2-1
  アプリケーションの構築に使用する
   その他のファイル 2-2
  コンパイラが生成するファイル 2-3
プログラムの例 2-3
  単純なコンソールアプリケーション 2-3
  より複雑な例 2-4
  ネイティブアプリケーション 2-5

第 3 章 プログラムとユニット 3-1

プログラムの構造と構文 3-1
  プログラムヘッダー 3-2
  プログラムの uses 節 3-2
  ブロック 3-2
ユニットの構造と構文 3-3
  ユニットヘッダー 3-3
  インターフェース部 3-4
  実現部 3-4
  初期化部 3-4
  終了処理部 3-5
ユニット参照と uses 節 3-5
  uses 節の構文 3-6
  複数のユニットの参照と間接的なユニット参照 3-6
  ユニットの循環参照 3-7

第 4 章 構文の要素 4-1

基本的な構文要素 4-1
  特殊シンボル 4-1
  識別子 4-2
    限定付き識別子 4-2
  予約語 4-3
  指令 4-3
  数字 4-3
  ラベル 4-4
  文字列 4-4
コメントとコンパイラ指令 4-5
式 4-5
  演算子 4-6
    算術演算子 4-6
    論理演算子 4-7
    論理演算子（ビット演算子） 4-8
    文字列演算子 4-9
    ポインタ演算子 4-9
    集合演算子 4-10
    関係演算子 4-11
    クラス演算子 4-11
    @ 演算子 4-12
    演算子の優先順位 4-12
  関数呼び出し 4-13
  集合構成子 4-13
  添字 4-14
  型キャスト 4-14
    値型キャスト 4-14
    変数型キャスト 4-15
宣言と文 4-16
  宣言 4-16
  文 4-17
  単純文 4-17
    代入文 4-17
    手続きと関数の呼び出し 4-18
    goto 文 4-18
  構造化文 4-19
    複合文 4-19
    with 文 4-20
    if 文 4-21
    case 文 4-23
    制御ループ 4-24
    repeat 文 4-24
    while 文 4-25
    for 文 4-25
ブロックとスコープ 4-27
  ブロック 4-27
  スコープ 4-27
    名前の衝突 4-28

第５章 データ型，変数，定数 5-1

型について 5-1
単純型 5-2
順序型 5-3
整数型 5-3
文字型 5-4
論理型 5-5
列挙型 5-6
部分範囲型 5-8
実数型 5-9
文字列型 5-10
短い文字列 5-11
長い文字列 5-12
WideString 5-12
拡張文字セットについて 5-12
ヌルで終わる文字列の処理 5-13
ポインタ，配列，文字列定数の使用 5-14
Pascal 文字列とヌルで終わる文字列の混在 . 5-15
構造化型 5-16
集合型 5-16
配列型 5-17
静的配列 5-17
動的配列 5-18
配列の型と代入 5-20
レコード型 5-21
レコードの可変部分 5-22
ファイル型 5-24
ポインタとポインタ型 5-25
ポインタの概要 5-25
ポインタ型 5-26
文字ポインタ 5-27
その他の標準のポインタ型 5-27
手続き型 5-27
文と式での手続き型 5-29
バリアント型 5-30
バリアント型の変換 5-31
式におけるバリアント 5-32
バリアント配列 5-32
OleVariant 5-33
型の互換性と同一性 5-33
型の同一性 5-34
型の互換性 5-34
代入の互換性 5-35
型の宣言 5-35
変数 5-36
変数の宣言 5-36
絶対アドレス 5-37
動的変数 5-38
スレッドローカル変数 5-38
宣言された定数 5-38
真の定数 5-39
定数式 5-40
リソース文字列 5-40
型付き定数 5-41
配列型定数 5-41
レコード型定数 5-42
手続き型定数 5-42
ポインタ型定数 5-42

第６章 手続きと関数 6-1

手続きと関数の宣言 6-1
手続きの宣言 6-2
関数宣言 6-3
呼び出し規約 6-4
forward 宣言とインターフェース宣言 6-5
external 宣言 6-6
オブジェクトファイルのリンク 6-6
ライブラリからの関数のインポート 6-7
手続きと関数のオーバーロード 6-8
ローカル宣言 6-10
ルーチンのネスト 6-10
パラメータ 6-11
パラメータのセマンティクス（意味論） . . . 6-11
値パラメータと変数パラメータ 6-12
定数パラメータ 6-13
out パラメータ 6-13
型なしパラメータ 6-14
文字列パラメータ 6-15
配列パラメータ 6-15
オープン配列パラメータ 6-15
型可変オープン配列パラメータ 6-16
デフォルトパラメータ 6-17
デフォルトパラメータと
オーバーロードルーチン 6-18
forward 宣言とインターフェース宣言での
デフォルトパラメータ 6-18
手続きと関数の呼び出し 6-19
オープン配列コンストラクタ 6-19

第７章 クラスとオブジェクト 7-1

クラス型 7-2
継承とスコープ 7-3
TObject と TClass 7-3
クラス型の互換性 7-3
オブジェクト型 7-4

クラスメンバーの可視性 7-4
プライベートメンバー，プロテクトメンバー，
パブリックメンバー 7-5
パブリッシュメンバー 7-5
自動化メンバー 7-6
forward 宣言と相互に依存するクラス 7-7
フィールド 7-7
メソッド 7-8
メソッドの宣言と実装 7-8
inherited .7-9
Self 7-9
メソッドの結合 7-10
静的メソッド 7-10
仮想メソッドと動的メソッド 7-11
抽象メソッド 7-12
メソッドのオーバーロード 7-12
コンストラクタ 7-13
デストラクタ 7-14
メッセージメソッド 7-15
メッセージメソッドの実装 7-16
メッセージのディスパッチ 7-17
プロパティ 7-17
プロパティのへアクセス 7-18
配列プロパティ 7-19
インデックス指定子 7-21
格納指定子 7-21
プロパティのオーバーライドと再宣言 7-22
クラス参照 7-23
クラス参照型 7-23
コンストラクタとクラス参照 7-24
クラス演算子 7-25
is 演算子 7-25
as 演算子 7-25
クラスメソッド 7-26
例外 7-26
例外処理の使用 7-26
例外型の宣言 7-27
例外の生成と処理 7-28
try..except 文 7-29
例外の再生成 7-31
例外のネスト 7-31
try..finally 文 7-32
標準の例外クラスと例外ルーチン 7-32

第 8 章 標準ルーチンと入出力 8-1

ファイルの入出力 8-1
テキストファイル 8-3
型なしファイル 8-4
テキストファイルデバイスドライバ 8-4
デバイス関連関数 8-5
Open 関数 8-5
InOut 関数 8-5
Flush 関数 8-6
Close 関数 8-6
ヌルで終わる文字列 8-6
ワイド文字列 8-7
その他の標準ルーチン 8-7

第 II 部 上級機能

第 9 章 ライブラリとパッケージ 9-1

動的にロード可能なライブラリの呼び出し 9-1
静的ロード 9-2
動的ロード 9-2
動的にロード可能なライブラリの記述 9-3
exports 節 9-5
ライブラリ初期化コード 9-6
ライブラリ内のグローバル変数 9-6
ライブラリとシステム変数 9-7
ライブラリ内の例外と実行時エラー 9-7
共有メモリマネージャ（Windows のみ）. . . . 9-8
パッケージ 9-8
パッケージの宣言とソースファイル 9-9
パッケージの命名 9-10
requires 節 9-10
contains 節 9-10
パッケージのコンパイル 9-11
生成ファイル 9-11
パッケージ固有のコンパイラ指令 9-11
パッケージ固有のコマンドライン
コンパイラスイッチ 9-12

第 10 章 オブジェクトインターフェース 10-1

インターフェース型 10-1
IInterface と継承 10-2
インターフェースの識別 10-3
インターフェースの呼び出し規約 10-3
インターフェースのプロパティ 10-4
前方参照宣言 10-4
インターフェースの実装 10-4
メソッド解決節 10-5
継承実装の変更 10-6
委任によるインターフェースの実装 10-6
インターフェース型プロパティへの委任 . 10-7

クラス型プロパティへの委任 10-7
インターフェース参照 10-8
インターフェースの代入互換性 10-9
インターフェースの型キャスト 10-10
インターフェースの問い合わせ 10-10
オートメーションオブジェクト（Windows のみ）10-10
ディスパッチインターフェース型
（Windows のみ）. 10-10
ディスパッチインターフェースのメソッド
（Windows のみ） 10-11
ディスパッチインターフェースのプロパティ
（Windows のみ） 10-11
オートメーションオブジェクトへのアクセス
（Windows のみ）. 10-11
オートメーションオブジェクトの
メソッド呼び出しの構文 10-12
デュアルインターフェース（Windows のみ）. 10-13

第 11 章 メモリ管理 11-1

メモリマネージャ（Windows のみ） 11-1
変数. 11-2
内部データ形式 11-2
整数型. 11-3
文字型. 11-3
論理型. 11-3
列挙型. 11-3
実数型. 11-3
Real48 型 11-4
Single 型 .11-4
Double 型 11-4
Extended 型 11-5
Comp 型 .11-5
通貨型 11-5
ポインタ型 .11-5
短い文字列型 11-5
長い文字列型 11-5
ワイド文字列型 11-6
集合型. 11-7
静的配列型 .11-7
動的配列型 .11-7
レコード型 .11-8
ファイル型. 11-9
手続き型. 11-10
クラス型. 11-10
クラス参照型. 11-11
バリアント型. 11-11

第 12 章 プログラムの制御 12-1

パラメータと関数の結果 12-1
パラメータの受け渡し. 12-1
レジスタ保存規約 12-3
関数の結果. 12-3
メソッド呼び出し. 12-3
コンストラクタとデストラクタ 12-4
終了手続き 12-4

第 13 章 インラインアセンブラコード 13-1

asm 文 13-1
レジスタの使用. 13-2
アセンブラ文の構文 13-2
ラベル 13-2
命令コード 13-2
RET 命令のサイズ 13-3
自動ジャンプサイズ 13-3
アセンブラ疑似命令. 13-3
オペランド 13-6
式 13-7
Object Pascal とアセンブラの式の相違点. . . 13-7
式の要素 13-8
定数 13-8
レジスタ 13-9
シンボル 13-10
式クラス 13-12
式の型. 13-13
式演算子. 13-14
アセンブラで記述した手続きと関数 13-15

付録 A Object Pascal の文法 A-1

索 引 I-1

表目次

4.1 予約語 4-3
4.2 指令 4-3
4.3 二項算術演算子 4-6
4.4 単項算術演算子 4-7
4.5 論理演算子 4-7
4.6 論理演算子（ビット演算子） 4-8
4.7 文字列演算子 4-9
4.8 文字ポインタ演算子 4-9
4.9 集合演算子 4-10
4.10 関係演算子 4-11
4.11 演算子の優先順位 4-12
5.1 Object Pascal の 32 ビット処理系の汎用整数型 5-3
5.2 基本整数型 5-4
5.3 基本実数型 5-9
5.4 汎用実数型 5-9
5.5 文字列型 5-10
5.6 System と SysUtils で宣言されている ポインタ型（抜粋） 5-27
5.7 バリアントの型変換規則 5-31
5.8 整数定数の型 5-39
6.1 呼び出し規約 6-5
8.1 入出力手続きと関数 8-1
8.2 ヌルで終わる文字列の関数 8-6
8.3 その他の標準ルーチン 8-7
9.1 コンパイル済みのパッケージファイル . . 9-11
9.2 パッケージ固有のコンパイラ指令 9-11
9.3 パッケージ固有のコマンドライン コンパイラスイッチ 9-12
11.1 長い文字列メモリブロックのレイアウト . 11-6
11.2 ワイド文字列動的メモリのレイアウト （Windows ） 11-6
11.3 ワイド文字列メモリブロックのレイアウト （Linux） 11-6
11.4 動的配列のメモリレイアウト 11-7
11.5 型アラインメントマスク 11-8
11.6 仮想メソッドテーブルのレイアウト . . . 11-11
13.1 組み込みアセンブラの予約語 13-6
13.2 文字列の例とその値 13-9
13.3 CPU レジスタ 13-9
13.4 組み込みアセンブラが認識するシンボル . 13-10
13.5 定義済みの型シンボル 13-14
13.6 組み込みアセンブラの式演算子の優先順位 13-14
13.7 組み込みアセンブラの式演算子 13-14

第 1 章
はじめに

このマニュアルは，Borland 開発ツールで使う Object Pascal 言語についての説明書です。

このマニュアルの内容

第 1 章〜第 7 章は，一般的なプログラミングで使われる言語要素について説明します。第 8 章では，ファイル入出力，および文字列操作について説明します。

その後は，ダイナミックリンクライブラリとパッケージに関する言語の拡張仕様と制約事項（第 9 章），オブジェクトインターフェースと（第 10 章）について説明します。最後の 3 つの章では，上級ユーザー向けの技術情報として，メモリ管理（第 11 章），プログラム制御（第 12 章），Object Pascal 言語で使うアセンブリ言語ルーチン（第 13 章）について説明します。

Object Pascal

この『Object Pascal 言語ガイド』は，Linux および Windows オペレーティングシステムで使用する Object Pascal 言語について説明しています。プラットフォームに起因する言語の違いについては，必要な場所でそのつど説明します。

Delphi/Kylix アプリケーション開発者の多くは，統合開発環境（IDE）の中で Object Pascal コードを書き，コンパイルを行います。ユニット間の依存関係の管理をはじめ，プロジェクトとソースファイルに関する細かな設定は，IDE がそのほとんどを処理してくれます。また，Borland の開発ツールを使用すると，厳密には Object Pascal 言語の仕様にはない制約が課せられることもあります。たとえば，ファイル名とプログラム名に一定の制約がありますが，IDE を使わずにコードを書いて，コマンドプロンプトでコンパイルした場合，この制約は回避できます。

このマニュアルは，IDE で作業し，ビジュアルコンポーネントライブラリ（VCL）または Borland クロスプラットフォーム用コンポーネントライブラリ（CLX：Component Library for Cross Platform）を使ってアプリケーションを構築する開発スタイルを前提としています。ただし，Object Pascal にはない Borland 固有の規則がある場合は，その旨を示しています。

表記規則

識別子は斜体で表記します。識別子とは，定数，変数，型，フィールド，プロパティ，手続き，関数，プログラム，ユニット，ライブラリ，パッケージなどの名前のことをいいます。Object Pascal の演算子，予約語，指令は，**Boldface**（太字）で表記します。サンプルコードの内容，およびファイルやコマンドプロンプトにそのまま入力する文字列は，`Monospace`（等幅文字）で表記します。

プログラム例では，以下のように，予約語と指令を **Boldface**（太字）で表記します。

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

IDE のコードエディタでも，［構文強調表示］オプションがオンの場合，予約語と指令は太字で表示されます。

プログラム例によっては，上記のように，省略記号（... または ⋮）を含んでいることがあります。この省略記号は，実際のファイルでは追加のコードがあることを意味します。省略記号はそのまま入力しないでください。

構文を説明する箇所では，プレースホルダを斜体で表記します。実際のコードを入力するときは，構文的に適切な要素で置き換えて入力してください。たとえば，関数宣言のヘッダーは，次のように表記されます。

`function` *functionName*(*argumentList*): *returnType*;

構文の説明では，省略記号（...）や添字が使われることもあります。

`function` *functionName*(*arg*$_1$, ..., *arg*$_n$): *ReturnType*;

その他の資料

製品のオンラインヘルプには，IDE とユーザーインターフェースに関する情報，および VCL または CLX に関する最新情報が記載されています。データベースアプリケーションの開発方法など，プログラミングに関する情報は『開発者ガイド』に詳しく記載されています。製品に付属するマニュアルの全体像については，『クイックスタート』を参照してください。

ソフトウェアの登録とテクニカルサポート

ボーランドでは，個人開発者，コンサルタント，企業ユーザーの方々のニーズにお応えするサポートプランを幅広く提供しています。本製品に関するサポートを受けるには，登録カードをご返送いただき，お客様のニーズにあったプランをお選びください。テクニカルサポートも含め，ボーランドが提供するサービスの詳細については，パッケージに含まれている小冊子『ご使用の前に』とともに，ボーランドの Web サイト（http://www.borland.co.jp/）をご覧ください。

第 I 部

言語の基本説明

第 I 部の各章は，ほとんどのプログラミングタスクで必要となる言語の基本的は要素について説明しています。各章は次の通りです。

- 第 2 章「概要」
- 第 3 章「プログラムとユニット」
- 第 4 章「構文の要素」
- 第 5 章「データ型，変数，定数」
- 第 6 章「手続きと関数」
- 第 7 章「クラスとオブジェクト」
- 第 8 章「標準ルーチンと入出力」

第 2 章

概要

Object Pascal は，型チェックの厳密なコンパイラ型の高水準言語であり，構造化されたオブジェクト指向設計をサポートします。コードが読みやすい，高速なコンパイル，複数のユニットファイルの使用によるモジュラープログラミングが可能であるなどの利点があります。

Object Pascal には，Borland のコンポーネントフレームワークと RAD 環境をサポートする特殊な機能があります。このマニュアルの説明と例では，ほとんどの場合，Object Pascal を使って，Delphi や Kylix などの Borland 開発ツールで アプリケーションを開発することを前提としています。

プログラムの構成

プログラムは通常，ユニットと呼ばれるソースコードモジュールに分かれています。プログラムは，プログラム名を指定するヘッダーで始まります。ヘッダーに続いてオプションの **uses** 節があり，宣言と文のブロックが続きます。**uses** 節には，プログラムにリンクされているユニットが列挙されます。ユニットは複数のプログラムから使用することができ，多くの場合は各ユニットにも独自の **uses** 節があります。

uses 節は，モジュール間の依存関係に関する情報をコンパイラに提供します。この情報はモジュール自体に含まれているため，Object Pascal のプログラムではメイクファイル，ヘッダーファイル，プリプロセッサのための **include** 指令などを必要としません。IDE にプロジェクトがロードされるとプロジェクトマネージャによってメイクファイルが生成されますが，これらのメイクファイルはプロジェクトグループに複数のプロジェクトが含まれる場合にのみ保存されます。

プログラムの構造と相互依存についての詳細は，第 3 章「プログラムとユニット」を参照してください。

Pascal ソースファイル

コンパイラでは，次の 3 種類のファイルに Pascal ソースコードがあることを前提としています。

- ユニットソースファイル（拡張子 .pas）

- プロジェクトファイル（拡張子 .dpr）
- パッケージソースファイル（拡張子 .dpk）

ユニットソースファイルには，アプリケーションのコードの大部分が含まれます。各アプリケーションには，1 つのプロジェクトファイルといくつかのユニットファイルがあります。プロジェクトファイルは従来の Pascal のメインプログラムファイルに相当し，ユニットファイルをまとめてアプリケーションを構成する働きがあります。アプリケーションのプロジェクトファイルは，Borland 開発ツールによって自動的に保守されます。

コマンドラインからプログラムをコンパイルする場合は，すべてのソースコードをユニットファイル（.pas ファイル）に記述することもできます。しかし，IDE を使ってアプリケーションを構築する場合は，プロジェクトファイル（.dpr ファイル）が必要です。

パッケージソースファイルはプロジェクトファイルに似ていますが，パッケージと呼ばれる特殊な動的にリンク可能なライブラリの作成に使用されます。パッケージについての詳細は，第 9 章「ライブラリとパッケージ」を参照してください。

アプリケーションの構築に使用するその他のファイル

ソースコードモジュールのほかに，Borland 製品ではいくつかの非 Pascal ファイルを使ってアプリケーションを構築します。これらのファイルは IDE によって自動的に保守され，以下のものがあります。

- 拡張子 .dfm（Delphi）または .xfm（Kylix）のフォームファイル
- 拡張子 .res のリソースファイル
- 拡張子 .dof（Delphi）または .kof（Kylix）のプロジェクトオプションファイル

フォームファイルは，テキストファイルか，あるいはビットマップや文字列などが含まれるコンパイル済みのリソースファイルです。各フォームファイルはそれぞれ 1 つのフォームを表し，通常はアプリケーションのウィンドウまたはダイアログボックスに相当します。IDE を使うと，フォームファイルをテキストとして表示および編集することができ，フォームファイルをテキストまたはバイナリ形式で保存できます。デフォルトではフォームファイルはテキストとして保存されますが，通常はこのテキストを直接編集することはなく，Borland のビジュアル設計ツールを使うのが普通です。プロジェクトには少なくとも 1 つのフォームがあり，フォームには関連付けられたユニットファイル（.pas ファイル）があります。デフォルトでは，このユニットファイルの名前はフォームファイルと同じです。

フォームファイルのほかに，プロジェクトではリソースファイル（.res ファイル）にアプリケーションのアイコンのビットマップを格納します。デフォルトでは，このファイルの名前はプロジェクトファイル（.dpr ファイル）と同じです。アプリケーションのアイコンを変更するには，［プロジェクトオプション］ダイアログボックスを使用してください。

プロジェクトオプションファイル（.DOF ファイル）には，コンパイラとリンカの設定，検索ディレクトリ，バージョン情報などが格納されます。各プロジェクトにはプロジェクトオプションファイルが関連付けられており，プロジェクトオプションファイルの名前はプロジェクトファイル（.DPR

ファイル）と同じです。通常，このファイルのオプションは［プロジェクトオプション］ダイアログボックスから設定します。

IDE の各種ツールは，異なる形式でデータをファイルに格納します。デスクトップ設定（.dsk または .desk）ファイルには，ウィンドウの配置やその他の設定に関する情報が格納されます。デスクトップ設定は，プロジェクトごとに，あるいは環境全体に作成できます。これらのファイルは，コンパイルには直接影響しません。

コンパイラが生成するファイル

アプリケーションまたは標準のダイナミックリンクライブラリを 1 回目に構築すると，プロジェクトで使用されている新しいユニットのそれぞれに対して，コンパイル済みユニットファイル .dcu（Windows）または .dcu/.dpu（Linux）がコンパイラによって作成されます。次に，プロジェクトに含まれるすべての .dcu（Windows）または .dcu/.dpu（Linux）ファイルがリンクされ，単一の実行形式ファイルまたは共有ライブラリが作成されます。パッケージを 1 回目に構築すると，パッケージに含まれる各ユニットに対して .dcu（Windows）または .dpu（Linux）ファイルがコンパイラによって作成されます。次に，.dcp ファイルとパッケージファイルが作成されます。ライブラリとパッケージについての詳細は，第 9 章「ライブラリとパッケージ」を参照してください。**-GD** スイッチを使うと，マップファイルと .drc ファイルがリンカによって生成されます。.drc ファイルには文字列リソースが格納されており，これをコンパイルするとリソースファイルを生成できます。

プロジェクトを再構築する場合，個別のユニットが再コンパイルされるのは，前回のコンパイル以後にユニットのソースファイル（.pas ファイル）を変更したか，対応する .dcu（Windows）または .dcu/.dpu（Linux）ファイルが見つからないか，またはコンパイラに対して再コンパイルを明示的に指示したときのみです。コンパイラがコンパイル済みユニットファイルを使用できるようになっていれば，ユニットのソースファイルは存在しなくてもかまいません。

プログラムの例

以下の例では，Object Pascal によるプログラミングの基本的な特徴について説明します。ここに示す例は単純な Object Pascal アプリケーションで，IDE からはコンパイルできませんが，コマンドラインからはコンパイルできます。

単純なコンソールアプリケーション

次のプログラムは単純なコンソールアプリケーションです。コマンドラインからコンパイルして実行することができます。

```
program Greeting;

{$APPTYPE CONSOLE}

var MyMessage: string;

begin
```

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

1 行目では，Greeting というプログラムを宣言しています。`{$APPTYPE CONSOLE}` 指令は，このプログラムがコマンドラインから実行されるコンソールアプリケーションであることをコンパイラに指示するものです。次の行では，MyMessage という変数を宣言しています。この変数には文字列が格納されます（Object Pascal には真正な文字列データ型があります）。次に，文字列「Hello world!」を MyMessage 変数に割り当て，Writeln 手続きを使って MyMessage の内容を標準出力に送っています。Writeln は，コンパイラによってすべてのアプリケーションに自動的に取り込まれる System ユニットで暗黙に定義されています。

Greeting.pas または Greeting.dpr というファイルにこのプログラムを入力し，コマンドラインで次のように入力するとコンパイルできます。

```
Delphi の場合： DCC32 Greeting
Kylix の場合： dcc Greeting
```

生成された実行形式ファイルを実行すると，「Hello world!」というメッセージが出力されます。

このプログラム例は，非常に単純である点のほかに，いくつかの点で一般的な Borland の開発ツールのアプリケーションと異なります。第一に，このプログラムはコンソールアプリケーションです。通常，Borland の開発ツールでは，グラフィックインターフェースを持つアプリケーションを作成します。このため，Borland の開発ツールアプリケーションで Writeln を呼び出すことはほとんどありません。さらに，このプログラム例は Writeln を除くプログラム全体が単一のファイルに記述されています。標準的なアプリケーションでは，プログラムヘッダー（プログラム例の 1 行目）は別のプロジェクトファイルに記述されます。ユニットファイルで定義されたメソッドがいくつか呼び出されることを除き，プロジェクトファイルには実際のアプリケーションロジックは含まれません。

より複雑な例

次の例では，プロジェクトファイルとユニットファイルという 2 つのファイルに分かれているプログラムを示します。プロジェクトファイルは次のとおりです。このファイルは Greeting.dpr という名前で保存できます。

```
program Greeting;

{$APPTYPE CONSOLE}

uses Unit1;

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

1 行目では，Greeting というプログラムを宣言しています。このプログラムもやはりコンソールアプリケーションです。`uses Unit1;` 節では，Unit1 というユニットを Greeting に含めることをコンパイラに指示しています。最後に，PrintMessage 手続きが呼び出され，「Hello World!」という文字列が渡されます。PrintMessage 手続きは，Unit1 で定義されています。次に，Unit1 のソースコードを示します。これは Unit1.pas というファイルに保存できます。

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

procedure PrintMessage(msg: string);
begin
  Writeln(msg);
end;

end.
```

Unit1 では，単一の文字列を引数として受け取り，この文字列を標準出力に送る PrintMessage という手続きが定義されています。Pascal では，値を返さないルーチンを手続きと呼びます。値を返すルーチンは関数と呼びます。PrintMessage が Unit1 で 2 回宣言されていることに注意してください。1 回目の宣言は予約語 **interface** の下にあります。これは，Greeting など，Unit1 を使用するほかのモジュールから PrintMessage を使用できるようにするものです。2 回目の宣言は予約語 **implementation** の下にあります。これは，PrintMessage を実際に定義するものです。

コマンドラインで次のように入力すると，Greeting をコンパイルできます。

```
Delphi の場合： DCC32 Greeting
Kylix の場合： dcc Greeting
```

コマンドライン引数として Unit1 を指定する必要はありません。コンパイラが Greeting.dpr を処理するときに，Greeting プログラムが依存するユニットファイルが自動的に検索されます。生成された実行形式ファイルは，前のプログラム例と同じように "Hello world!" というメッセージを出力します。

ネイティブアプリケーション

次の例は，IDE で VCL または CLX コンポーネントを使用して構築したアプリケーションです。このプログラムでは自動的に生成されたフォームファイルとリソースファイルを使用するため，ソースコードのみでコンパイルすることはできません。しかし，Object Pascal の重要な機能はいくつか示されています。このプログラムでは，複数のユニットを使用するほかに，クラスとオブジェクトを使用しています。クラスとオブジェクトについては第 7 章「クラスとオブジェクト」で後述します。

このプログラムには，プロジェクトファイルと 2 つの新しいユニットファイルがあります。最初にプロジェクトファイルを示します。

```
program Greeting;  { コメントは中カッコで囲む }

uses
  Forms,  { Linux ではユニット名を QForms に変更する }
  Unit1 in 'Unit1.pas'  { Form1 のためのユニット },
  Unit2 in 'Unit2.pas'  { Form2 のためのユニット },

{$R *.res}  { この指令はプロジェクトのリソースファイルをリンクする }

begin
  { Application の呼び出し }
```

```
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

このプログラムの名前も Greeting です。このプログラムでは 3 つのユニットを使用します。Forms は VCL および CLX の一部です。Unit1 はアプリケーションのメインフォーム（Form1）に関連付けられています。Unit2 は別のフォーム（Form2）に関連付けられています。

このプログラムでは，Application という名前のオブジェクトを繰り返し呼び出します。Application オブジェクトは，Forms ユニットで定義されている TApplication クラスのインスタンスです。すべてのプロジェクトには自動生成された Application オブジェクトがあります。Application オブジェクトへの呼び出しのうちの 2 つでは，TApplication クラスの CreateForm メソッドを呼び出しています。CreateForm の 1 回目の呼び出しでは，Unit1 で定義されている TForm1 クラスのインスタンスである Form1 が作成されます。CreateForm の 2 回目の呼び出しでは，Unit2 で定義されている TForm2 クラスのインスタンスである Form2 が作成されます。

Unit1 は次のとおりです。

```
unit Unit1;

interface

uses  { これらのユニットは VCL（ビジュアルコンポーネントライブラリ）に含まれる }
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
{
Linux では uses 節は次のようになる .
uses  { これらのユニットは CLX に含まれる }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}

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

var
  Form1: TForm1;

implementation

uses Unit2;  { Form2 はこのユニットで定義されている }

{$R *.dfm}  { この指令は Unit1 のフォームファイルをリンクする }

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

end.
```

Unit1 では，TForm から派生した TForm1 クラスと，このクラスのインスタンスである Form1 を作成します。TForm1 には，TButton のインスタンスである Button1 というボタンと，実行時にユーザーが

Button1 を選択したときに必ず呼び出される TForm1.Button1Click という手続きがあります。TForm1.Button1Click では，Form1 を隠し，Form2.ShowModal を呼び出して Form2 を表示します。Form2 は Unit2 で定義されています。

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
{
Linux では uses 節は次のようになる .
uses  { これらのユニットは CLX に含まれる }
  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 では，TForm2 というクラスと，このクラスのインスタンスである Form2 を作成します。TForm2 には，TButton のインスタンスである CancelButton というボタンと，TLabel のインスタンスである Label1 というラベルがあります。ソースコードではわかりませんが，Label1 には「Hello world!」というキャプションが表示されます。このキャプションは，Form2 のフォームファイルである Unit2.dfm で定義されています。

Unit2 では 1 つの手続きが定義されています。TForm2.CancelButtonClick は，実行時にユーザーが CancelButton を選択したときに必ず呼び出され，Form2 を閉じます。この手続きは，プログラム実行中に発生するイベントに反応するので，イベントハンドラと呼ばれます。Unit1 の TForm1.Button1Click もイベントハンドラです。イベントハンドラは，Form1 と Form2 のフォームファイル（Windows では .dfm，Linux では .xfm ファイル）で特定のイベントに割り当てられています。

Greeting プログラムを起動すると，Form1 が表示され，Form2 は表示されません。デフォルトでは，プロジェクトファイルで最初に作成されるフォームのみが実行時に表示されます。このフォームはプロジェクトのメインフォームと呼ばれます。Form1 のボタンをユーザーが選択すると，Form1 が隠されて Form2 が表示され，メッセージ「Hello world!」が表示されます。ユーザーが CancelButton をクリックするか，タイトルバーの閉じるボタンを選択すると，Form2 が閉じられて Form1 が再び表示されます。

第3章

プログラムとユニット

プログラムは，ユニットと呼ばれるソースコードモジュールを使って作成されます。各ユニットは独自のファイルに格納され，個別にコンパイルされます。アプリケーションはコンパイル済みユニットをリンクして作成されます。ユニットを使用してできることは以下のとおりです。

- 大規模なプログラムをモジュールに分割し，個別に編集できるようにする
- ライブラリを作成し，複数のプログラムで共有できるようにする
- ソースコードを公開せずにほかの開発者にライブラリを配布する

従来の Pascal プログラムでは，メインプログラムを含むすべてのソースコードが .pas ファイルに記述されていました。Borland のツールでは，メインプログラムはプロジェクトファイル（.dpr ファイル）に記述され，残りのソースコードのほとんどはユニットファイル（. ぱ ｓ ファイル）に記述されます。アプリケーション（プロジェクト）は，単一のプロジェクトファイルと 1 つ以上のユニットファイルで構成されます。厳密に言えば，プロジェクトでユニットを明示的に使用する必要はありません。ただし，System ユニットはすべてのプログラムによって自動的に使用されます。コンパイラがプロジェクトを構築するには，各ユニットのソースファイルまたはコンパイル済みのユニットファイルが必要です。

プログラムの構造と構文

プログラムに含まれるものは以下のとおりです。

- プログラムヘッダー
- **uses** 節（オプション）
- 宣言と文で構成されるブロック

プログラムヘッダーはプログラムの名前を指定します。**uses** 節はプログラムで使用されるユニットを示します。ブロックには，宣言と，プログラム実行時に実行される文が含まれます。IDE では，これらの 3 つの要素が単一のプロジェクトファイル（.dpr ファイル）にあることを前提としています。

次の例は Editor というプログラムのプロジェクトファイルです。

```
1      program Editor;
2
3      uses
4        Forms,
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.
```

1 行目はプログラムヘッダーです。**uses** 節は 3 ～ 6 行目にあります。8 行目はプロジェクトのリソースファイルをプログラムにリンクするコンパイラ指令です。10 ～ 14 行目はプログラム実行時に実行される文のブロックです。最後に，すべてのソースファイルと同じように，プロジェクトファイルはピリオドで終わります。

これはかなり典型的なプロジェクトファイルです。通常，プログラムのロジックのほとんどはユニットファイルにあるので，プロジェクトファイルは短いファイルです。プロジェクトファイルの生成と保守は自動的に行われます。ほとんどの場合，手動で編集する必要はありません。

プログラムヘッダー

プログラムヘッダーはプログラムの名前を指定します。予約語 **program** の後に有効な識別子が続き，その後のセミコロンで終わります。識別子とプロジェクトファイルの名前が一致していなければなりません。前の例では，プログラム名が Editor であるので，プロジェクトファイルの名前は Editor.dpr でなければなりません。

標準 Pascal では，プログラムヘッダーでプログラム名の後にパラメータを指定できます。

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

Borland の Object Pascal コンパイラはこれらのパラメータを無視します。

プログラムの uses 節

uses 節は，プログラムの一部として使用されるユニットを示します。指定されたユニットに独自の **uses** 節がある場合もあります。**uses** 節についての詳細は，3-5 ページの「ユニット参照と uses 節」を参照してください。

ブロック

ブロックには，プログラム実行時に実行される単一の単純文または構造化文が含まれます。ほとんどのプログラムの場合，ブロックは予約語 **begin** と **end** で囲まれた複合文であり，この複合文はプロジェクトの Application オブジェクトへのメソッド呼び出し文で構成されます。Application 変数はす

べてのプロジェクトに存在し，TApplication，TWebApplication，または TServiceApplication のインスタンスが格納されています。定数，型，変数，手続き，および関数の宣言をブロックに含めることもできます。これらの宣言はブロックの文よりも前に記述しなければなりません。

ユニットの構造と構文

ユニットは，型（クラスを含む），定数，変数，およびルーチン（関数と手続き）で構成されます。ユニットはそれぞれ独自のユニットファイル（.pas ファイル）で定義されます。

ユニットファイルはユニットヘッダーで始まり，インターフェース部，実現部，初期化部，および終了処理部が続きます。初期化部と終了処理部はオプションです。ユニットファイルの概略は次のとおりです。

```
unit Unit1;

interface

uses { ここにユニットを指定する }

  { インターフェース部を記述する }

implementation

uses { ここにユニットを指定する }

  { 実現部を記述する }

initialization
  { 初期化部を記述する }

finalization
  { 終了処理部を記述する }

end.
```

ユニットは **end** とピリオドで終わらなければなりません。

ユニットヘッダー

ユニットヘッダーはユニットの名前を指定します。予約語 **unit** の後に有効な識別子が続き，その後にセミコロンが続きます。Borland のツールで開発されたアプリケーションの場合は，識別子とユニットファイルの名前が一致する必要があります。たとえば，次のようなユニットヘッダーがあるとします。

```
unit MainForm;
```

このようなユニットヘッダーのソースファイルは MAINFORM.pas という名前になり，コンパイル済みユニットが格納されるファイルの名前は MAINFORM.dcu になります。

ユニット名はプロジェクト内で重複があってはなりません。ユニットファイルが異なるディレクトリにある場合でも，名前が同じ 2 つのユニットを 1 つのプログラムで使用することはできません。

インターフェース部

ユニットのインターフェース部は予約語 **interface** で始まり，実現部の直前で終わります。インターフェース部は，クライアントが使用できる定数，型，変数，手続き，および関数を宣言します。クライアントとは，ユニットを宣言して使用するほかのユニットまたはプログラムのことです。クライアントは，クライアントで宣言した場合と同じようにこれらの要素にアクセスできます。このため，これらの要素はパブリックな要素と呼ばれます。

手続きや関数のインターフェース宣言は，そのルーチンのヘッダーのみを含みます。手続きまたは関数のブロックは実現部に入ります。したがって，インターフェース部の手続きと関数の宣言は，**forward** 指令が使われていないにもかかわらず，**forward** 宣言と同じように機能します。

クラスのインターフェース宣言は，クラスのすべてのメンバーの宣言を含んでいなければなりません。

インターフェース部が独自に **uses** 節を持つ場合もあります。この場合，**uses** 節は予約語 **interface** の直後になければなりません。**uses** 節についての詳細は，3-5 ページの「ユニット参照と uses 節」を参照してください。

実現部

ユニットの実現部は予約語 **implementation** で始まり，初期化部の直前で終わるか，初期化部がない場合はユニットの末尾で終わります。実現部は，インターフェース部で宣言された手続きと関数を定義します。実現部の内部では，これらの手続きと関数の定義と呼び出しをどのような順序でも行うことができます。パブリックな手続きと関数を実現部で定義するときは，ヘッダーのパラメータリストを省略できます。ただし，省略しない場合は，インターフェース部の宣言とパラメータリストが正確に一致する必要があります。

実現部では，パブリックな手続きと関数を定義するほかに，ユニットのプライベートな定数，型（クラスを含む），変数，手続き，および関数を定義することもできます。プライベートな要素にクライアントがアクセスすることはできません。

実現部が独自の **uses** 節を含む場合もあります。この場合，**uses** 節は予約語 **implementation** の直後になければなりません。**uses** 節についての詳細は，3-5 ページの「ユニット参照と uses 節」を参照してください。

初期化部

初期化部はオプションです。初期化部は予約語 **initialization** で始まり，終了処理部の直前で終わるか，終了処理部がない場合はユニットの末尾で終わります。初期化部には，プログラムの起動時に実行される文が含まれます。これらの文は記述されている順序で実行されます。たとえば，初期化の必要なデータ構造を定義している場合は，初期化部で初期化を行うことができます。

クライアントが使用する各ユニットの初期化部は，クライアントの **uses** 節でユニットが指定されているのと同じ順序で実行されます。

終了処理部

終了処理部はオプションであり，初期化部を持つユニットにのみ存在することができます。終了処理部は予約語 **finalization** で始まり，ユニットの末尾で終わります。終了処理部の文は，メインプログラムが終了したときに実行されます。初期化部で割り当てたリソースを解放するときは，終了処理部を使用します。

終了処理部は初期化部と逆の順序で実行されます。たとえば，アプリケーションでユニット A，B，および C をこの順序で初期化した場合は，C，B，A の順に終了処理が実行されます。

いったんユニットの初期化コードが実行を開始すると，それに対応する終了処理部がアプリケーションのシャットダウン時に必ず実行されます。したがって，終了処理部は初期化が不完全だったデータを処理できなければなりません。その理由は，実行時エラーが発生した場合には，初期化コードが完全に実行されない可能性があるからです。

ユニット参照と uses 節

uses 節は，その節があるプログラム，ライブラリ，またはユニットが使用するユニットを指定します。ライブラリについての詳細は，第 9 章「ライブラリとパッケージ」を参照してください。**uses** 節は以下の場所に記述することができます。

- プログラムまたはライブラリのプロジェクトファイル
- ユニットのインターフェース部
- ユニットの実現部

ほとんどのプロジェクトファイルと，ほとんどのユニットのインターフェース部には **uses** 節があります。ユニットの実現部に独自の **uses** 節がある場合もあります。

System ユニットはすべてのアプリケーションによって自動的に使用され，**uses** 節で明示的に指定することはできません。System には，ファイル入出力，文字列処理，浮動小数点演算，動的メモリ割り当てなどのルーチンが実装されています。SysUtils など，その他の標準ライブラリユニットは **uses** 節で指定する必要があります。ほとんどの場合は，プロジェクトでソースファイルの生成と保守が行われるときに，必要なすべてのユニットが **uses** 節に自動的に追加されます。

ユニット宣言および **uses** 節では，Kylix の場合は特に，ユニット名と対応するファイル名は大文字と小文字が一致していなければなりません。識別子の限定など，それ以外のコンテキストではユニット名の大文字小文字は区別されません。ユニット参照に関する問題を回避するには，次のようにユニットソースファイルを明示的に参照するようにします。

```
uses MyUnit in "myunit.pas";
```

プロジェクトファイルにこのような明示的な参照がある場合，他のソースファイルでは，次のように大文字小文字が一致していなくても，単純な **uses** 節でユニットを参照することができます。

```
uses Myunit;
```

uses 節の配置と内容についての詳細は，3-6 ページの「複数のユニットの参照と間接的なユニット参照」と 3-7 ページの「ユニットの循環参照」を参照してください。

uses 節の構文

uses 節は予約語 **uses** で始まり，1 つ以上のユニット名がカンマ区切りで指定された後，セミコロンで終わります。例を以下に示します。

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

プログラムまたはライブラリの **uses** 節では，ユニット名に続いて予約語 **in** とソースファイルの名前を指定できます。ソースファイル名は引用符で囲みます。ディレクトリパスは指定することもしないこともでき，絶対パスでも相対パスでも指定できます。例を以下に示します。

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;
uses
  QForms,
  Main,
  Extra in '../extra/extra.pas';
```

「**in** ...」をユニット名の後に指定するのは，ユニットのソースファイルを指定する必要がある場合です。IDE はユニット名とユニットがあるファイルの名前が一致することを前提としているため，通常はこのように指定する必要はありません。**in** を使用する必要があるのは，ソースファイルの場所が明確でない場合のみです。たとえば以下のような場合です。

- 使用するソースファイルがプロジェクトファイルと異なるディレクトリにあり，そのディレクトリがコンパイラの検索パスにも一般ライブラリ検索パスにもない場合
- コンパイラの検索パスに含まれる複数のディレクトリに同じ名前のユニットが存在する場合
- コンソールアプリケーションをコマンドラインからコンパイルする場合で，ユニット名とソースファイル名が一致しないとき

コンパイラは，**in** ... 構文によって，ユニットがプロジェクトの一部であるかどうかを判別します。プロジェクト（.dpr）ファイルの **in** の付いた **uses** 節に指定されているユニットとファイル名はプロジェクトの一部とみなされ，他のユニットはプロジェクトで使用されているけれども属してはいないことになります。この区別はコンパイルには影響しませんが，プロジェクトマネージャやプロジェクトブラウザなどの IDE ツールには関わりがあります。

ユニットの **uses** 節では，コンパイラに対して **in** を使ってソースファイルの場所を指定することはできません。コンパイラの検索パス，一般ライブラリ検索パス，または使用する側のユニットと同じディレクトリにすべてのユニットがなければなりません。さらに，ユニット名とソースファイル名が一致していなければなりません。

複数のユニットの参照と間接的なユニット参照

uses 節でユニットが指定されている順序は，ユニット初期化の順序を決定し（3-4 ページの「初期化部」を参照），コンパイラが識別子を特定する方法に影響を与えます。同じ名前の変数，定数，型，手続き，または関数が 2 つのユニットで宣言されている場合，コンパイラは **uses** 節で後に指定されているユニットのものを使用します。もう一方のユニットの識別子にプログラムからアクセスするには，UnitName.Identifier のように限定子を追加しなければなりません。

uses 節で指定する必要があるユニットは，その節があるプログラムまたはユニットで使用するユニットだけです。つまり，ユニット B で宣言されている定数，型，変数，手続き，または関数をユニット A が参照する場合，ユニット A はユニット B を明示的に使用する必要があります。ユニット B がユニット C の識別子を参照している場合は，ユニット A はユニット C に間接的に依存します。この場合，ユニット A の **uses** 節でユニット C を指定する必要はありませんが，コンパイラがユニット A を処理するには，ユニット B とユニット C の両方をコンパイラが見つけることができなければなりません。

次の例は，このような間接的依存を示します。

```
program Prog;
uses Unit2;
const a = b;
⋮
unit Unit2;
interface
uses Unit1;
const b = c;
⋮
unit Unit1;
interface
const c = 1;
⋮
```

この例では，Prog は Unit2 に直接依存しており，Unit2 は Unit1 に直接依存しています。したがって，Prog は Unit1 に間接的に依存しています。Unit1 は Prog の uses 節に指定されていないため，Unit1 で宣言されている識別子に Prog からアクセスすることはできません。

クライアントモジュールをコンパイルするには，クライアントが直接または間接的に依存するすべてのユニットをコンパイラが見つけることができなければなりません。ただし，ユニットのソースコードが変更されていない場合は，ソースファイル（.pas ファイル）ではなく .dcu（Windows）または .dcu/.dpu（Linux）ファイルのみが必要になります。

ユニットのインターフェース部を変更した場合は，そのユニットに依存するほかのユニットを再コンパイルしなければなりません。しかし，実現部など，インターフェース部以外の部分のみが変更された場合は，依存するユニットを再コンパイルする必要はありません。コンパイラはこれらの依存関係を自動的に追跡し，必要な場合にのみユニットを再コンパイルします。

ユニットの循環参照

ユニットが直接または間接に相互を参照する場合，これらのユニットは相互に依存していると言います。各ユニットのインターフェース部の **uses** 節の間に循環的な参照が存在しない限り，相互依存は許容されます。つまり，いずれかのユニットのインターフェース部から始まって，ほかのユニットのインターフェース部の参照をたどっていった結果，元のユニットに戻ってしまってはなりません。相互依存のパターンを有効にするには，循環参照の各経路が少なくとも 1 つの実現部の **uses** 節を経由しなければなりません。

このため，相互に依存するユニットが 2 つあるもっとも単純な場合には，これらのユニットのインターフェース部にある **uses** 節で相互に相手を参照することはできません。したがって，次の例ではコンパイル時にエラーが発生します。

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

ただし，どちらかの参照を実現部に移動すれば，これらのユニットは相互に参照できます。

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

循環参照が発生する可能性を少なくするには，できる限り実現部の **uses** 節でユニットを参照したほうがよいでしょう。インターフェース部の **uses** 節でユニットを参照する必要があるのは，そのユニットの識別子をインターフェース部で使用する場合のみです。

第4章

構文の要素

Object Pascal は，A ～ Z と a ～ z の英字，0 ～ 9 の数字，およびその他の標準文字を含む ASCII 文字セットを使用します。大文字と小文字は区別されません。スペース（ASCII 32）と，行の終わりを表す復帰文字（ASCII 13）などの制御文字（ASCII 0 ～ 31）は空白と呼ばれます。

トークンと呼ばれる基本的な構文要素が組み合わされることによって，式，宣言，および文が構成されます。文はプログラム内で実行可能なアルゴリズム上の処理を表します。式は文の一部をなす構文単位であり，ある値を表します。宣言は，関数名や変数名などの識別子を定義します。識別子は式や文で使用できます。必要に応じて，宣言が識別子のためのメモリを割り当てる場合もあります。

基本的な構文要素

もっとも単純なレベルで見ると，プログラムはセパレータで区切られた一連のトークンによって構成されます。トークンはプログラムのテキストで意味のある最小の単位です。セパレータは空白またはコメントです。厳密に言えば，2 つのトークンをセパレータで区切る必要がない場合もあります。たとえば，次のコードは正しい Object Pascal コードです。

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

しかし，慣行と可読性を考慮すると，これは次のように記述することが求められます。

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

トークンは，特殊シンボル，識別子，予約語，指令，数字，ラベル，および文字列に分類されます。セパレータは，トークンが文字列である場合に限り，トークンの一部となることができます。隣接する識別子，予約語，数字，およびラベルの間には，1 つ以上のセパレータが必要です。

特殊シンボル

特殊シンボルは，英数字以外の 1 文字，または英数字以外の 2 文字の組み合わせで，固定された意味を持つものです。以下の各文字は特殊シンボルです。

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

以下の文字の組み合わせも特殊シンボルです。

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

左大カッコ（**[**）は左カッコとピリオドの組み（**(.**）に相当し，右大カッコ（**]**）はピリオドと右カッコの組み（**.)**）に相当します。左カッコとアスタリスクの組み（**(***）と右カッコとアスタリスクの組み（***)**）は，左右の中カッコ（**{ }**）に相当します。

!，**"**（二重引用符），**%**，**?**，****，**_**（下線記号），**|**（縦棒），および **~**（チルダ）は特殊文字でないことに注意してください。

識別子

識別子は，定数，変数，フィールド，型，プロパティ，手続き，関数，プログラム，ユニット，ライブラリ，およびパッケージを表します。識別子の長さは任意ですが，意味のあるのは最初の 255 文字だけです。識別子は英字または下線記号（_）で始めなければならず，途中に空白を入れることはできません。2 文字目以降には英字，数字，および下線記号を使うことができます。予約語は識別子として使用できません。

Object Pascal は大文字と小文字を区別しないので，CalculateValue という識別子は以下のいずれの方法でも記述しても同じです。

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Linux では，**uses** 節のユニット名を表す識別子だけは，大文字と小文字が区別されます。ユニット名はファイル名に対応しているので，大文字と小文字の区別が統一されていないとコンパイルに影響を与える場合があります。

限定付き識別子

複数の箇所で宣言されている識別子を使う場合は，識別子を限定することが必要なときがあります。限定付き識別子の構文は次のとおりです。

識別子$_1$. 識別子$_2$

識別子$_1$ が識別子$_2$ を限定します。たとえば，CurrentValue という変数が 2 つのユニットでそれぞれ宣言されている場合，Unit2 の CurrentValue にアクセスすることを指定するには，次のように記述します。

```
Unit2.CurrentValue
```

限定子は重ねて指定できます。次に例を示します。

```
Form1.Button1.Click
```

この例は，Form1 の Button1 の Click メソッドを呼び出します。

識別子を限定しない場合，識別子の解釈は 4-27 ページの「ブロックとスコープ」で後述するスコープの規則に従って決定されます。

予約語

以下の予約語は，識別子として再定義または使用することはできません。

表 4.1　予約語

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**

表 4.1 の予約語のほかに，**private**，**protected**，**public**，**published**，および **automated** もオブジェクト型宣言の中では予約語と同じように働きます。ただし，それ以外の場合では指令として扱われます。**at** と **on** にも特別な意味があります。

指令

指令は，Object Pascal で特別な意味を持つ語ですが，予約語とは異なり，ユーザー定義の識別子が記述できない場所でのみ使われます。このため，指令とまったく同じスペルの識別子を定義することはできますが，このような識別子の定義はお勧めできません。

表 4.2　指令

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

数字

整数定数と実数定数を 10 進表記で表すには，カンマやスペースを含まない連続した数字として表記し，符号を表すにはプレフィクスとして + 演算子または - 演算子を付けます。値はデフォルトで正に

なるので，たとえば 67258 は +67258 と同等です。また，定数の値は定義済みの実数型最大値または整数型最大値の範囲に含まれなければなりません。

小数点または指数部のある数は実数を表し，それ以外の数は整数を表します。実数に文字 E または e が含まれる場合は，「〜掛ける 10 の…乗」と解釈します。たとえば，7E-2 は 7×10^{-2} を表し，12.25e+6 と 12.25e6 はどちらも 12.25×10^{6} を表します。

$8F のようなドル記号のプレフィクスは 16 進数を表します。単項演算子 − が付いていない 16 進数値は正の値として扱われます。代入を行うときに，16 進値が，受け取り側の型の値の範囲を超える場合は，エラーが生成されます。ただし，受け取り側が Integer（32 ビット整数）の場合は警告が生成されます。その場合，Integer の正の範囲を超える値は，2 の補数の整数表現形式の負の数値として扱われます。

実数型と整数型についての詳細は，第 5 章「データ型，変数，定数」を参照してください。数値のデータ型についての詳細は，5-39 ページの「真の定数」を参照してください。

ラベル

ラベルは 4 桁以下の数字の列，つまり 0 〜 9999 の数です。先頭のゼロには意味がありません。識別子をラベルとして使うこともできます。

ラベルは **goto** 文で使います。**goto** 文とラベルについての詳細は，4-18 ページの「goto 文」を参照してください。

文字列

文字列は，文字列リテラルまたは文字列定数とも呼ばれ，引用符付き文字列，制御文字列，または引用符付き文字列と制御文字列の組み合わせで構成されます。セパレータは，引用符付き文字列の内部でのみ使用できます。

文字列は拡張 ASCII 文字セットの文字の列であり，文字数は最大で 255 文字です。単引用符で囲んで 1 行で記述します。単引用符の間に何もない引用符付き文字列はヌル文字列になります。引用符付き文字列中の連続した 2 つの単引用符は 1 つの単引用符を表します。次に例を示します。

```
'BORLAND'           { BORLAND }
'You''ll see'       { You'll see }
''''                { ' }
''                  { ヌル文字列 }
' '                 { スペース }
```

制御文字列は 1 つ以上の制御文字の列です。制御文字は # シンボルに続く 0 〜 255 の符号なし整数定数（10 進数または 16 進数）で表記し，整数定数に対応する ASCII 文字を表します。たとえば，次のような制御文字列があるとします。

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

これは次の引用符付き文字列と一致します。

```
'You'
```

引用符付き文字列と制御文字列を組み合わせると，長い文字列を表すことができます。たとえば，次のように記述すると，

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

「Line 1」と「Line 2」の間に復帰改行を挿入できます。ただし，2 つの引用符付き文字列を同じ方法で結合することはできません。連続する 2 つの単引用符は 1 つの単引用符として解釈されるためです。引用符付き文字列を結合するには，4-9 ページの「文字列演算子」で後述する + 演算子を使用するか，1 つの引用符付き文字列にまとめます。

文字列の長さはその文字列にある文字の数を表します。任意の長さの文字列はどの文字列型とも互換性があり，PChar 型と互換性があります。**{$X+}** で拡張構文が有効になっている場合は長さが 1 の文字列はどの文字型とも互換性があり，長さが n（n は 1 以上）の文字列は要素数が n の文字型のゼロベースの配列および packed 配列と互換性があります。文字列型についての詳細は，第 5 章「データ型，変数，定数」を参照してください。

コメントとコンパイラ指令

コメントは，隣接するトークンを区切るセパレータとして機能する場合と，コンパイラ指令として機能する場合を除き，コンパイラによって無視されます。

コメントを記述するには，以下の 3 通りの方法があります。

```
{ 左中カッコと右中カッコの間のテキストはコメントです。}
(* 左カッコとアスタリスクの組み合わせと
   アスタリスクと右カッコの間のテキストもコメントです。*)
// スラッシュ 2 つと行の終わりの間のテキストはコメントです。
```

始まりの **{** または **(*** の直後にドル記号（**$**）のあるコメントは，コンパイラ指令です。次に例を示します。

```
{$WARNINGS OFF}
```

これは警告メッセージの生成を無効にするコンパイラ指令です。

式

式は値を返す構文要素です。次に例を示します。

```
X                  { 変数 }
@X                 { 変数のアドレス }
15                 { 整数定数 }
InterestRate       { 変数 }
Calc(X,Y)          { 関数呼び出し }
X * Y              { X と Y の積 }
Z / (1 - Z)        { Z と (1 - Z) の商 }
X = 1.5            { 論理型 }
C in Range1        { 論理型 }
not Done           { 論理型の否定 }
['a','b','c']      { 集合 }
Char(48)           { 値型キャスト }
```

もっとも単純な式は変数と定数です。変数と定数についての詳細は，第 5 章「データ型，変数，定数」を参照してください。複雑な式は，演算子，関数呼び出し，集合構成子，添字，および型キャストを使って単純な式をもとにして構成されます。

演算子

演算子は，Object Pascal 言語に組み込まれた定義済み関数のように機能します。たとえば，`(X + Y)` という式は `X` と `Y` の 2 つの変数および **+** 演算子で構成されます。`X` と `Y` はオペランド（演算の対象となる項）と呼ばれます。`X` と `Y` が整数または実数を表す場合，`(X + Y)` はその和を返します。演算子には，**@**，**not**，**^**，*****，**/**，**div**，**mod**，**and**，**shl**，**shr**，**as**，**+**，**-**，**or**，**xor**，**=**，**>**，**<**，**<>**，**<=**，**>=**，**in**，および **is** があります。

@，**not**，および **^** の各演算子は単項演算子で，1 つのオペランドを取ります。その他の演算子はすべて二項演算子で，2 つのオペランドを取ります。ただし，**+** と **-** は例外で，単項演算子と二項演算子のどちらとしても機能します。単項演算子は，`-B` のように常にそのオペランドの前に置きます。ただし，**^** は例外で，`P^` のようにオペランドの後に置きます。二項演算子は，`A = 7` のようにオペランドの間に置きます。

一部の演算子は，渡されたデータの型によって動作が異なります。たとえば **not** は，整数オペランドに対してビットごとの否定演算を行い，論理型オペランドに対しては論理否定演算を行います。後の表では，これらの演算子は複数の分類に記載されています。

^，**is**，および **in** を除くいずれの演算子でも，バリアント型のオペランドを取ることができます。詳細については 5-30 ページの「バリアント型」を参照してください。

以下の各節では，Object Pascal のデータ型について読者に一定の知識があることを前提としています。データ型についての詳細は，第 5 章「データ型，変数，定数」を参照してください。

複雑な式での演算子の優先順位についての詳細は，4-12 ページの「演算子の優先順位」を参照してください。

算術演算子

算術演算子は，実数または整数のオペランドを取り，**+**，**-**，*****，**/**，**div**，および **mod** があります。

表 4.3　二項算術演算子

演算子	演算	オペランド	結果	例
+	加算	整数型，実数型	整数型，実数型	`X + Y`
-	減算	整数型，実数型	整数型，実数型	`Result - 1`
*****	乗算	整数型，実数型	整数型，実数型	`P * InterestRate`
/	実数除算	整数型，実数型	実数型	`X / 2`
div	整数除算	整数型	整数型	`Total` **div** `UnitSize`
mod	剰余	整数型	整数型	`Y` **mod** `6`

表 4.4　単項算術演算子

演算子	演算	オペランド	結果	例
+	符号恒等	整数型，実数型	整数型，実数型	+7
-	符号否定	整数型，実数型	整数型，実数型	-X

算術演算子には，次の規則が適用されます。

- x/y の値は，x と y の型に関係なく，常に Extended 型になる。これ以外の算術演算子については，少なくとも 1 つのオペランドが実数型の場合，結果は Extended 型になる。いずれのオペランドも実数型でない場合，少なくとも 1 つのオペランドが Int64 型であるときは，結果は Int64 型になる。それ以外の場合は，結果は Integer 型になる。オペランドの型が整数型の部分範囲である場合，オペランドは整数型として扱われる
- x **div** y の値は，x/y の値を 0 の方向に向かってもっとも近い整数に丸めた値
- **mod** 演算子はオペランドによる除算の剰余を返す。つまり，x **mod** y = x - (x **div** y) * y になる
- x/y，x **div** y，または x **mod** y という形の式で y がゼロである場合，実行時エラーが発生する

論理演算子

論理演算子 **not**，**and**，**or**，および **xor** は，任意の論理型のオペランドを取り，Boolean 型の値を返します。

表 4.5　論理演算子

演算子	演算	オペランド	結果	例
not	否定	論理型	論理型	**not** (C **in** MySet)
and	論理積	論理型	論理型	Done **and** (Total > 0)
or	論理和	論理型	論理型	A **or** B
xor	排他的論理和	論理型	論理型	A **xor** B

これらの演算は標準的なブール論理に従います。たとえば，x **and** y という形の式は，x と y のどちらも True である場合にのみ True になります。

完全な論理評価とショートサーキット論理評価

コンパイラは，**and** 演算子と **or** 演算子について，完全評価とショートサーキット評価（部分的評価）という 2 つの評価モードをサポートしています。完全評価では，式全体の結果がすでに判明している場合でも，すべてのオペランドを評価します。ショートサーキット評価では，左から右へ厳密に評価が実行され，式全体の結果が判明した時点で評価が中止されます。たとえば，`A and B` という式がショートサーキットモードで評価される場合，A が False であれば，A を評価した時点で式全体が False であることが判明するので，コンパイラは B を評価しません。

ショートサーキット評価は実行時間が最短になることが保証されており，ほとんどの場合はコードサイズも最小になるので，通常はショートサーキット評価を使うことをお勧めします。どちらかのオペランドが関数であり，その副作用によってプログラムの実行が変化するような場合には完全評価が便利です。

また，ショートサーキット評価を使うと，完全評価では実行時に不正となる可能性がある演算を含む構文も記述することができます。たとえば，次のコードは文字列 S の最初のカンマまでを繰り返し処理します。

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

S にカンマが含まれていない場合は，最後の繰り返しで I は S の長さよりも大きい値になります。**while** 条件のテストが次に行われるとき，完全評価では S[I] の読み出しが試みられるので，実行時エラーが発生することになります。ここでショートサーキット評価を使用すれば，**while** 条件の前半が False になると，後半の (S[I] <> ',') は評価されません。

評価モードを制御するには，**$B** コンパイラ指令を使います。デフォルトの状態は，ショートサーキット評価を有効にする **{$B-}** です。完全評価をローカルに有効にするには，**{$B+}** 指令をコードに追加します。プロジェクト全体で完全評価を有効にするには，［プロジェクトオプション］ダイアログの［コンパイラ］タブで［完全論理式評価］を選択します。

メモ　いずれかのオペランドにバリアントが含まれている場合，コンパイラは（**{$B-}** 状態であっても）必ず完全評価を行います。

論理演算子（ビット演算子）

以下の論理演算子は，整数オペランドに対するビットごとの操作を行います。たとえば，X に格納されている値の 2 進表現が 001101 であり，Y に格納されている値の 2 進表現が 100001 であるとします。

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

この式は 101101 という値を Z に代入します。

表 4.6　論理演算子（ビット演算子）

演算子	演算	オペランド	結果	例
not	ビットごとの否定	整数型	整数型	**not** X
and	ビットごとの論理積	整数型	整数型	X **and** Y
or	ビットごとの論理和	整数型	整数型	X **or** Y
xor	ビットごとの排他的論理和	整数型	整数型	X **xor** Y
shl	ビットごとの左シフト	整数型	整数型	X **shl** 2
shr	ビットごとの右シフト	整数型	整数型	Y **shr** I

ビット演算子には，次の規則が適用されます。

- **not** 演算の結果の型はオペランドの型と同じになる
- **and**，**or**，または **xor** 演算のオペランドが 2 つとも整数型である場合，結果の型は，2 つのオペランドが取りうるすべての値を範囲に含む定義済みの整数型のうち，もっとも小さい範囲を持つ型になる
- x **shl** y と x **shr** y の各演算は，x の値を左または右に y ビットだけシフトする。これは x に対する 2y の乗算または除算に相当し，結果は x と同じ型になる。たとえば，N に格納されている値が

01101（10 進数の 13）である場合，`N shl 1` は 11010（10 進数の 26）を返す。y の値は，x の型のサイズの剰余として解釈されることに注意してください。したがって，たとえば x が Integer の場合，`x shl 40` は `x shl 8` として解釈されます。Integer は 32 ビットであり，`40 mod 32 = 8` となるためです。

文字列演算子

関係演算子 =，<>，<，>，<=，および >= は，すべて文字列オペランドを取ることができます。4-11 ページの「関係演算子」を参照してください。+ 演算子は 2 つの文字列を結合します。

表 4.7　文字列演算子

演算子	演算	オペランド	結果	例
+	結合	文字列型，パック文字列型，文字型	文字列	`S + '.'`

文字列の結合には，次の規則が適用されます。

- + のオペランドは，文字列型，パック文字列型（Char 型の packed 配列），または文字型のいずれでも可能である。ただし，どちらかのオペランドが WideChar 型である場合，もう一方のオペランドは長い文字列型でなければならない
- + 演算の結果はすべての文字列型と互換性がある。ただし，両方のオペランドが短い文字列型または文字型である場合，結果の長さが 255 文字を超えると，256 文字目以降は切り捨てられる

ポインタ演算子

関係演算子 <，>，<=，および >= は，すべて PChar 型のオペランドを取ることができます。4-11 ページの「関係演算子」を参照してください。以下の演算子もポインタをオペランドに取ることができます。ポインタについての詳細は，5-25 ページの「ポインタとポインタ型」を参照してください。

表 4.8　文字ポインタ演算子

演算子	演算	オペランド	結果	例
+	ポインタ加算	文字ポインタ型，整数型	文字ポインタ型	`P + I`
-	ポインタ減算	文字ポインタ型，整数型	文字ポインタ型，整数型	`P - Q`
^	ポインタ逆参照	ポインタ型	ポインタの基本型	`P^`
=	等しい	ポインタ型	論理型	`P = Q`
<>	等しくない	ポインタ型	論理型	`P <> Q`

^ 演算子はポインタを逆参照します。つまり，ポインタが指すメモリアドレスに格納されている値を返します。オペランドのポインタには任意のポインタ型を使用できますが，汎用の Pointer 型は逆参照する前に型キャストを行わなければなりません。

P = Q になるのは，P と Q が同じアドレスを指している場合のみです。それ以外の場合は，P <> Q が True になります。

+ 演算子と - 演算子を使うと，文字ポインタのオフセットを増減できます。また，- を使うと 2 つの文字ポインタのオフセットの差を計算することもできます。以下の規則が適用されます。

- I が整数で P が文字ポインタである場合，P + I は P が指すアドレスに I を加算し，P より I 文字後のアドレスを指すポインタを返す。I + P という式は P + I と同じである。P - I は P が指すアドレスから I を減算し，P より I 文字前の文字を指すポインタを返す
- P と Q がどちらも文字ポインタである場合，P - Q は P が指すアドレス（高い方のアドレス）と Q が指すアドレス（低い方のアドレス）の差を計算し，P と Q の間の文字数に相当する整数を返す。P + Q は定義されていない

集合演算子

以下の演算子は集合をオペランドに取ることができます。

表 4.9　集合演算子

演算子	演算	オペランド	結果	例
+	和	集合型	集合型	`Set1 + Set2`
-	差	集合型	集合型	`S - T`
*	積	集合型	集合型	`S * T`
<=	左辺が右辺の部分集合	集合型	論理型	`Q <= MySet`
>=	右辺が左辺の部分集合	集合型	論理型	`S1 >= S2`
=	等しい	集合型	論理型	`S2 = MySet`
<>	等しくない	集合型	論理型	`MySet <> S1`
in	帰属関係	順序型，集合型	論理型	A **in** Set1

+，-，および * には以下の規則が適用されます。

- 順序値 O が X または Y に含まれる場合にのみ，O は X + Y に含まれる。O が X に含まれ，Y に含まれていない場合にのみ，O は X - Y に含まれる。O が X と Y の両方に含まれている場合にのみ，O は X * Y に含まれる
- +，-，または * 演算の結果は **set of** A..B 型になる。A は結果の集合でもっとも小さい順序値，B はもっとも大きい順序値である

<=，>=，=，<>，および **in** には以下の規則が適用されます。

- X <= Y は，X のすべての要素が Y の要素である場合にのみ True になる。Z >= W は W <= Z と同じである。U = V が True になるのは，U と V の各要素が同じである場合のみである。それ以外の場合は，U <> V が True になる
- 順序値 O が集合 S の要素である場合にのみ，O **in** S は True になる

関係演算子

関係演算子は 2 つのオペランドの比較に使用します。演算子 =，<>，<=，および >= は集合にも適用されます（4-10 ページの「集合演算子」を参照）。また = と <> はポインタにも適用されます（4-9 ページの「ポインタ演算子」を参照）。

表 4.10　関係演算子

演算子	演算	オペランド	結果	例
=	等しい	単純，クラス，クラス参照，インターフェース，文字列，パック文字列の各型	論理型	`I = Max`
<>	等しくない	単純，クラス，クラス参照，インターフェース，文字列，パック文字列の各型	論理型	`X <> Y`
<	より小さい	単純，文字列，パック文字列，PChar の各型	論理型	`X < Y`
>	より大きい	単純，文字列，パック文字列，PChar の各型	論理型	`Len > 0`
<=	以下	単純，文字列，パック文字列，PChar の各型	論理型	`Cnt <= I`
>=	以上	単純，文字列，パック文字列，PChar の各型	論理型	`I >= 1`

ほとんどの単純型については，比較に関して複雑な規則はありません。たとえば，I = J が True になるのは，I と J が同じ値である場合のみです。それ以外の場合は，I <> J が True になります。関係演算子には，次の規則が適用されます。

- オペランドは互換性のある型でなければならない。例外として，実数と整数は比較できる
- 文字列は拡張 ASCII 文字セットの文字順に従って比較される。文字型は長さが 1 の文字列として扱われる
- 2 つのパック文字列を比較するには，要素の数が一致していなければならない。n 個の要素を持つパック文字列を文字列と比較する場合，パック文字列は長さが n の文字列として扱われる
- 演算子 <，>，<=，および >= は，2 つの PChar が同じ文字配列内を指している場合にのみ，PChar 型のオペランドに適用できる
- 演算子 = と <> では，クラス型とクラス参照型のオペランドを取ることができる。クラス型のオペランドの場合，= と <> はポインタの場合と同じ規則に従って評価される。つまり，C と D が同じインスタンスオブジェクトを指している場合にのみ C = D が True になり，それ以外の場合は C <> D が True になる。クラス参照型のオペランドの場合は，C と D が同じクラスを表している場合にのみ C = D が True になり，それ以外の場合は C <> D が True になる。クラスについての詳細は，第 7 章「クラスとオブジェクト」を参照

クラス演算子

as と **is** の各演算子は，クラスとインスタンスオブジェクトをオペランドに取ります。**as** はインターフェースをオペランドに取ることもできます。詳細については，第 7 章「クラスとオブジェクト」と第 10 章「オブジェクトインターフェース」を参照してください。

関係演算子 = と <> はクラスをオペランドに取ることもできます。4-11 ページの「関係演算子」を参照してください。

@ 演算子

@ 演算子は，変数，関数，手続き，またはメソッドのアドレスを返します。つまり，@ はオペランドを指すポインタを生成します。ポインタについての詳細は，5-25 ページの「ポインタとポインタ型」を参照してください。@ には，次の規則が適用されます。

- X が変数である場合，**@**X は X のアドレスを返す。ただし，X が手続き型変数である場合は特殊な規則が適用される。5-29 ページの「文と式での手続き型」を参照。デフォルトの **{$T-}** コンパイラ指令が有効である場合，**@**X の型は Pointer である。**{$T+}** 状態では，**@**X の型は ^T（T は X の型）になる
- F がルーチン（関数または手続き）である場合，**@**F は F のエントリポイントを返す。**@**F の型は常に Pointer である
- クラスで定義されているメソッドに @ を適用する場合は，メソッド識別子をクラス名で限定しなければならない。次に例を示します。

```
@TMyClass.DoSomething
```

  これは TMyClass の DoSomething メソッドを指す。クラスとメソッドについての詳細は，第 7 章「クラスとオブジェクト」を参照

演算子の優先順位

複雑な式では，優先順位の規則に従って演算の実行順序が決まります。

表 4.11　演算子の優先順位

演算子	**優先順位**
@，**not**	1 位（最高）
*，/，**div**，**mod**，**and**，**shl**，**shr**，**as**	2 位
+，-，**or**，**xor**	3 位
=，<>，<，>，<=，>=，**in**，**is**	4 位（最低）

優先順位の高い演算子は優先順位の低い演算子よりも先に評価され，優先順位の等しい演算子は左から評価されます。したがって次の式は，

```
X + Y * Z
```

Y と Z を掛けて，その結果を X に加えます。優先順位の高い * が，+ より先に実行されます。しかし，

```
X - Y + Z
```

この式では，X から Y を引いて，その結果に Z が加えられます。- と + は優先順位が等しいため，左側の演算が先に実行されます。

カッコを使用すると，優先順位の規則をオーバーライドできます。カッコ内の式は先に評価され，次に 1 つのオペランドとして扱われます。次に例を示します。

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

この式では，X と Y の和と Z が乗算されます。

一見すると不要と思われる状況でもカッコが必要な場合があります。次に例を示します。

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

この式が次のように解釈されることを意図して書かれたものだとしても，

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

カッコがなければ，コンパイラは演算子の優先順位の規則に従って次のように解釈します。

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

このため，Z が論理型でない限り，コンパイルエラーが発生します。

厳密にはカッコが不要な場合でも，カッコを使用することでコードの記述や解釈が容易になります。たとえば，最初の例は次のように記述することもできます。

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

コンパイラにとってこれらのカッコは不要ですが，カッコがあることにより，プログラマもコードの読み手も演算子の優先順位に留意する必要がなくなります。

関数呼び出し

関数は値を返すので，関数呼び出しは式の一種です。たとえば，2 つの整数引数を取って整数を返す Calc という関数を定義した場合，`Calc(24, 47)` という関数呼び出しは整数型の式です。I と J が整数型の変数である場合は，`I + Calc(J, 8)` も整数型の式になります。関数呼び出しの例には以下のものがあります。

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

関数についての詳細は，第 6 章「手続きと関数」を参照してください。

集合構成子

集合構成子は集合型の値を示します。次に例を示します。

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

これは，5，6，7，および 8 を要素とする集合を表します。次の集合構成子でも，

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

同じ集合を表すことができます。

集合構成子の構文は次のとおりです。

[項目$_1$，...，項目$_n$]

各項目は，集合の基本型を型とする順序値を表す式か，そのような 2 つの順序値を表す式の間に 2 つのピリオド（**..**）を記述したものです。x**..**y という形式の項目は，x から y までのすべての順序値を表します。ただし，x が y よりも大きい場合，x**..**y は何も表さず，**[**x**..**y**]** は空集合を表します。集合構成子 [ ] は空集合を表します。**[**x**]** は x を唯一の要素とする集合を表します。

集合構成子の例を示します。

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

集合についての詳細は，5-16 ページの「集合型」を参照してください。

添字

文字列，配列，配列プロパティ，および文字列または配列を指すポインタには，添字を付けてアクセスすることができます。たとえば，FileName が文字列変数である場合，`FileName[3]` という式は FileName が表す文字列の 3 つ目の文字を返します。`FileName[I + 1]` は I が指す文字の直後の文字を返します。文字列についての詳細は，5-10 ページの「文字列型」を参照してください。配列と配列プロパティについての詳細は，5-17 ページの「配列型」と 7-19 ページの「配列プロパティ」を参照してください。

型キャスト

ある式を別の型の式として扱うと便利な場合があります。型キャストを行うと，式の型を一時的に変更することによってこれを実現できます。たとえば，`Integer('A')` は文字 A を整数に型キャストします。

型キャストの構文は次のとおりです。

型識別子 (式)

式が変数である場合，結果は変数型キャストと呼ばれます。それ以外の場合，結果は値型キャストと呼ばれます。この 2 種類の型キャストは同じ構文を使用しますが，適用される規則は異なります。

値型キャスト

値型キャストでは，型識別子とキャストされる式が両方とも順序型か，両方ともポインタ型でなければなりません。値型キャストの例には以下のものがあります。

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

結果の値はカッコ内の式を変換することで得られます。指定した型のサイズが式のサイズと異なっている場合，切り捨てまたは拡張が行われることがあります。式の符号は常に保持されます。

次の文は，

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

`Integer('A')` の値である 65 を変数 I に代入します。

値型キャストの後に限定子を付けることはできません。また，代入文の左側では値型キャストを使用できません。

変数型キャスト

変数はすべて任意の型にキャストできます。ただし，変数と型のサイズが同じであり，整数と実数が混在しないことが条件となります。数値型を変換するには，Int や Trunc などの標準関数を使用してください。以下に変数型キャストの例を示します。

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

変数型キャストは代入文のどちらの側でも使用できます。したがって，

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

この文は文字 z（ASCII 122）を MyChar に代入します。

変数は手続き型にキャストできます。たとえば，次のように宣言されているとします。

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

この場合は，次の代入が可能です。

```
F := Func(P);       { P の手続き値を F に代入します }
Func(P) := F;       { F の手続き値を P に代入します }
@F := P;            { P のポインタ値を F に代入します }
P := @F;            { F のポインタ値を P に代入します }
N := F(N);          { F を使って関数を呼び出します }
N := Func(P)(N);    { P を使って関数を呼び出します }
```

次の例に示すように，変数型キャストの後に限定子を付けることもできます。

```
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;
```

この例では，TByteRec を使用してワードの下位バイトと上位バイトにアクセスし，TWordRec を使用して Longint 型の下位ワードと上位ワードにアクセスしています。組み込み関数の Lo と Hi を呼び出しても同じことができますが，変数型キャストは代入文の左辺にも使えるという長所があります。

ポインタの型キャストについての詳細は，5-25 ページの「ポインタとポインタ型」を参照してください。クラス型とインターフェース型の型キャストについての詳細は，7-25 ページの「as 演算子」と 10-10 ページの「インターフェースの型キャスト」を参照してください。

宣言と文

uses 節と **implementation** などのようにユニットの各部を分ける予約語を除くと，プログラムは宣言と文のみで構成されており，これらはブロックに分けられています。

宣言

変数，定数，型，フィールド，プロパティ，手続き，関数，プログラム，ユニット，ライブラリ，およびパッケージの名前は識別子と呼ばれます。26057 のような数値定数は識別子ではありません。識別子は使用する前に宣言する必要があります。例外は，コンパイラが自動的に認識する定義済みのいくつかの型，ルーチン，および定数と，関数ブロックでの変数 Result，およびメソッドの実装での変数 Self のみです。

宣言は識別子を定義し，必要に応じて識別子のためのメモリを割り当てます。次に例を示します。

```
var Size: Extended;
```

これは Extended 型（実数）の値を格納する Size という変数を宣言します。

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

これは 2 つの文字列を引数として受け取って整数を返す DoThis という関数を宣言します。宣言はすべてセミコロンで終わります。いくつかの変数，定数，型，またはラベルを一度に宣言するときは，適切な予約語を 1 回しか記述する必要がありません。

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

宣言の構文と配置は，定義する識別子の種類によって異なります。一般に，宣言はブロックの先頭か，ユニットのインターフェース部または実現部の先頭（**uses** 節の後）にのみ記述できます。変数，定数，型，関数などを宣言する場合の固有の規約については，それぞれの章に記載されています。

"hint" 指令の **platform**，**deprecated**，および **library** はすべての宣言の後ろに記述できます。手続きまたは関数の宣言の場合，hint 指令と残りの宣言の間をセミコロンで区切ります。次に例を示します。

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

ソースコードが **{$HINTS ON} {$WARNINGS ON}** 状態でコンパイルされる場合，これらの指令を使って宣言された識別子への参照により，適切なヒントまたは警告が生成されます。特定の動作環境（Windows や Linux など）に固有のアイテムを示すには **platform** を使用し，古いアイテムや下位互換性のためにだけサポートされているアイテムを示すには **deprecated** を使用し，特定のライブラリまたはコンポーネントのフレームワーク（VCL や CLX など）に依存することを示すには **library** を使用します。

文

文はプログラム内のアルゴリズム的処理を定義します。代入や手続き呼び出しなどの単純な文を組み合わせると，ループや条件文などの構造化文を作成できます。

ブロック内の複数の文およびユニットの初期化部と終了処理部の複数の文を区切るには，セミコロンを使用します。

単純文

単純文とは，ほかのいかなる文も含まない文のことです。単純文には，代入，手続きと関数の呼び出し，および **goto** ジャンプがあります。

代入文

代入文は次の形をとります。

```
変数 := 式
```

左辺の変数には，変数，変数型キャスト，逆参照されたポインタ，または構造化型変数の要素を含む任意の変数参照を使用できます。右辺の式には，代入互換性のある任意の式を使用できます。（関数ブロック内では，左辺の変数を定義される関数の名前で置き換えることができます。（第 6 章「手続きと関数」を参照）。**:=** シンボルは代入演算子と呼ばれることもあります。

代入文は，変数の現在の値を式の値で置き換えます。次に例を示します。

```
I := 3;
```

これは 3 という値を変数 I に代入します。代入文の左辺の変数参照は，右辺の式でも使用できます。次に例を示します。

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

これは I の値をインクリメントします。その他の代入文には次のものがあります。

```
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;
```

手続きと関数の呼び出し

手続き呼び出しは，手続き名と，必要な場合はパラメータリストによって構成されます。手続き名には限定子を付ける場合と付けない場合があります。次に例を示します。

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

{$X+} で拡張構文が有効になっている場合，関数呼び出しは，手続きの呼び出しと同様に独立した文として扱うことができます。

```
MyFunction(X);
```

このように関数を使用する場合，戻り値は破棄されます。

手続きと関数についての詳細は，第 6 章「手続きと関数」を参照してください。

goto 文

goto 文は次の形をとります。

```
goto label
```

goto 文は，指定のラベルでマークされた文へプログラムの実行を転送します。文をマークするには，あらかじめラベルを宣言しなければなりません。ラベルを宣言したら，マークする文の先頭にラベルとコロンを付けます。

```
label: statement
```

ラベルは次のように宣言します。

```
label ラベル ;
```

複数のラベルを一度に宣言することもできます。

```
label ラベル1, ..., ラベルn;
```

ラベルには任意の有効な識別子または 0 ～ 9999 の任意の数を使用できます。

ラベルの宣言，マークされた文，および **goto** 文は，同じブロックに存在していなければなりません。(4-27 ページの「ブロックとスコープ」を参照)。このため，ジャンプによって手続きや関数の中に入ったり手続きや関数から出たりすることはできません。同一のブロックの複数の文を同じラベルでマークしてはいけません。

次に例を示します。

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

これは Beep 手続きを繰り返し呼び出す無限ループになります。

構造化プログラミングでは，**goto** 文の使用は一般に好ましくないとされています。ただし，次の例のように，ネストしたループから抜け出る方法として使用されることがあります。

```
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 ... { X, Y, および Z について何かの条件が満たされたら } then
          goto FoundAnAnswer;

  ⋮{ 答えが見つからなかった場合に実行するコード }
  Exit;

  FoundAnAnswer:
    ⋮{ 答えが見つかった場合に実行するコード }
end;
```

goto はネストしたループから出るために使用していることに注意してください。ループまたはその他の構造化文に goto を使用して入ってはいけません。予期しない結果が生じる可能性があります。

構造化文

構造化文はほかの文を使用して構築されている文です。文を順に実行したり，条件付きで実行したり，繰り返し実行したりする場合は，構造化文を使用します。

- 複合文と **with** 文は，それを構成する文を順に実行する
- 条件文（**if** 文または **case** 文）は，指定された判断基準に基づいて，それを構成する文のうち最高で 1 つを実行する
- 繰り返し文には，**repeat** ループ，**while** ループ，および **for** ループがあり，それを構成する文をくり返し順に実行する
- **raise**，**try...except**，および **try...finally** の各構文を含む一連の特殊な文は，例外の生成と処理を行う。例外の生成と処理についての詳細は，7-26 ページの「例外」を参照

複合文

複合文は一連の単純文または構造化文で構成され，これらの文は記述されている順で実行されます。複合文は予約語 **begin** で始まって予約語 **end** で終わります。複合文を構成する文はセミコロンで区切られます。次に例を示します。

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

end の前の最後のセミコロンはオプションです。したがって，これは次のように記述することもできます。

```
begin
```

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

複合文は，Object Pascal の構文によって単一の文が要求される場合に必要です。プログラム，関数，および手続きのブロックのほかに，条件文やループなどの構造化文でも使用します。次に例を示します。

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

1 つの文のみで構成される複合文を記述することもできます。複雑な式でのカッコと同様に，**begin** と **end** を使用することによってあいまいさが解消され，可読性が向上する場合があります。文をまったく含まない複合文を使用して，何も実行しないブロックを作成することもできます。

```
  begin
  end;
```

with 文

with 文は，レコードのフィールドやオブジェクトのフィールド，プロパティ，およびメソッドを参照するための簡便な方法です。**with** 文の構文を次に示します。

with obj do 文

または

with obj_1，...，obj_n do 文

obj はオブジェクトまたはレコードを表す変数参照であり，文は任意の単純文または構造化文です。文の内部では，限定子を付けずに識別子のみで obj のフィールド，プロパティ，およびメソッドを参照することができます。

たとえば，次のように宣言されているとします。

```
  type TDate = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

  var OrderDate: TDate;
```

この場合は，次の **with** 文を記述することができます。

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

```
      Month := Month + 1;
```

これは次と同じです。

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

obj の解釈に配列の添字付けやポインタの逆参照が必要な場合は，複合文が実行される前にそれらの処理が 1 回だけ実行されます。このため，**with** 文は簡潔であるだけでなく効率も優れています。また，このため，**with** 文の実行中は文内部での変数への代入によって obj の解釈が影響を受けることはありません。

with 文内の変数参照とメソッド名は，可能な限り指定されたオブジェクトまたはレコードの要素として解釈されます。同じ名前を持つ別の変数やメソッドに **with** 文からアクセスする場合は，次の例のように限定子を前に付ける必要があります。

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

with の後に複数のオブジェクトまたはレコードを指定すると，文全体が一連のネストした **with** 文のように扱われます。したがって次のコードでは,

with obj_1，obj_2，...，obj_n do 文

したがって，上の宣言と下の宣言は同じ意味になります。

with obj_1 do
  with obj_2 do
     ⋮
     with obj_n do
       文

この場合，文の変数参照またはメソッド名は可能な限り obj_n の要素として解釈されます。解釈できない場合は可能な限り obj_{n-1} の要素として解釈され，以下同様に続きます。同じ規則が obj の解釈にも適用されます。たとえば，obj_n という要素が obj_1 と obj_2 の両方に存在する場合は，$obj_2.obj_n$ と解釈されます。

if 文

if 文には，**if...then** と **if...then...else** の 2 つの形があります。**if...then** 文の構文を次に示します。

```
if 式 then 文
```

式は論理値を返します。式が True である場合は文が実行され，それ以外の場合は実行されません。次に例を示します。

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

if...then...else 文の構文を次に示します。

if 式 then 文$_1$ else 文$_2$

式は論理値を返します。式が True である場合は文$_1$ が実行され，それ以外の場合は文$_2$ が実行されます。次に例を示します。

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

then 節と **else** 節にはそれぞれ 1 つの文が含まれますが，これらの文には構造化文も使用できます。次に例を示します。

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

then 節と **else** の間にはセミコロンを使用しないことに注意してください。**if** 文全体の後にセミコロンを使用してブロック内の次の文から区切ることはできます。しかし，**then** 節と **else** 節の間にはスペースまたは改行以外は何も必要としません。**if** 文で **else** の直前にセミコロンを使用するのは，よくあるプログラミングエラーです。

ネストした **if** 文に関して特殊な問題が発生する場合があります。この問題の原因は，**if** 文に **else** 節があるものとないものがあり，しかもこれらの 2 種類の文の構文が **else** 節の有無を除いて同じであることです。一連のネストした条件文で **else** 節が **if** 文よりも少ない場合は，どの **else** 節がどの **if** に結びつくのかが明らかでない場合があります。たとえば次のような文があるとします。

if 式$_1$ then if 式$_2$ then 文$_1$ else 文$_2$;

この文は，次の 2 通りの方法で解析できます。

if 式$_1$ then [if 式$_2$ then 文$_1$ else 文$_2$];

if 式$_1$ then [if 式$_2$ then 文$_1$] else 文$_2$;

コンパイラは常に 1 番目の方法で解釈します。次に実際のコードで記述します。

```
if ... { 式1 } then
  if ... { 式2 } then
    ... { 文1 }
  else
    ... { 文2 } ;
```

したがって，上の宣言と下の宣言は同じ意味になります。

```
if ... { 式1 } then
begin
  if ... { 式2 } then
    ... { 文1 }
  else
```

```
    ... { 文2 }
end;
```

ネストした条件文の解析はもっとも内側の条件文から行われ，**else** はすべて左側のもっとも近い **if** に結び付けられます。この例を 2 番目の方法でコンパイラに解釈させるには，次のように明示的に記述する必要があります。

```
if ... { 式1 } then
begin
  if ... { 式2 } then
    ... { 文1 }
end
else
  ... { 文2 } ;
```

case 文

case 文を使用すると，複雑にネストした **if** 条件文と同じ処理を読みやすい形式で記述できます。**case** 文は次の形をとります。

```
case セレクタ式 of
  caseList1: statement1;
  ⋮
  caseListn: statementn;
end
```

セレクタ式には順序型の任意の式を使用できます。文字列型は無効です。ケースリストはそれぞれ以下のいずれかです。

- コンパイラがプログラムを実行せずに評価できる数値や宣言済みの定数などの式。セレクタ式と互換性のある順序型でなければならない。したがって，`7`，`True`，`4 + 5 * 3`，`'A'`，および `Integer('A')` はすべてケースリストとして使用できるが，変数とほとんどの関数呼び出しは使用できない。ただし，Hi や Lo など一部の組み込み関数はケースリストで使用できる。5-40 ページの「定数式」を参照
- First..Last の形式の部分範囲。First と Last の両方が前の項目の条件を満たし，First が Last 以下であることが必要
- 項目 $_1$，...，項目 $_n$ の形式のリスト。各項目は前の項目の条件のいずれかを満たしていることが必要

ケースリストが表す値はそれぞれ **case** 文内で一意的なものでなければならず，部分範囲またはリストが重複してはいけません。**case** 文の末尾では **else** 節を使用できます。

```
case セレクタ式 of
  ケースリスト1: 文1;
  ⋮
  ケースリストn: 文n;
else
  文;
end
```

セミコロンで区切られた一連の文の場合は，**case** 文が実行されると，文 $_1$ ～文 $_n$ のうち最大で 1 つが実行されます。セレクタ式と値の等しいケースリストがあれば，そのケースリストの文が使用されます。セレクタ式と値の等しいケースリストがなく，**else** 節がある場合は，**else** 節の文が実行されます。

次の **case** 文があるとします。

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

これは次のネストした条件文と一致します。

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

case 文の例をもう 1 つ示します。

```
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;
```

制御ループ

ループを使用すると，一連の文を繰り返し実行し，制御条件または制御変数を使用して実行を停止することができます。Object Pascal には，**repeat** 文，**while** 文，および **for** 文の 3 種類の制御ループがあります。

標準の Break 手続きと Continue 手続きを使用すると，**repeat** 文，**while** 文，または **for** 文の流れを制御できます。Break はそれが含まれている文の実行を終了します。Continue は次の繰り返しに実行を移します。各手続きについての詳細は，オンラインヘルプを参照してください。

repeat 文

repeat 文の構文を次に示します。

```
repeat 文1; ...; 文n; until 式
```

式は論理値を返します。**until** の前の最後のセミコロンはオプションです。**repeat** 文はそれを構成する一連の文を繰り返し実行し，繰り返しを終了するごとに式をテストします。式が True を返すと **repeat** 文は終了します。式は 1 回目の実行の後まで評価されないため，一連の文は常に最低でも 1 回は実行されます。

repeat 文の例には次のものがあります。

```
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 文

while 文は **repeat** 文に似ていますが，一連の文が 1 回目に実行される前に制御条件が評価される点が異なります。したがって，条件が False であれば，一連の文は 1 度も実行されません。

while 文の構文を次に示します。

```
while 式 do 文
```

式は論理値を返し，文には複合文を使用できます。**while** 文はそれを構成する一連の文を繰り返し実行し，各繰り返しを開始する前に式をテストします。式が True を返す限り実行は継続されます。

while 文の例には次のものがあります。

```
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 文

for 文は，**repeat** 文または **while** 文と異なり，ループを実行する回数を明示的に指定する必要があります。**for** 文の構文を次に示します。

```
for カウンタ := 初期値 to 終値 do 文
```

または

```
for カウンタ := 初期値 downto 終値 do 文
```

各要素の説明は次のとおりです。

- カウンタは，**for** 文が含まれるブロックで宣言される限定子を持たない順序型のローカル変数である
- 初期値と終値はカウンタと代入互換の式である
- 式はカウンタの値を変更しない単純文または構造化文である

for 文は初期値の値をカウンタに代入した後，文を繰り返し実行し，繰り返しのたびにカウンタをインクリメントまたはデクリメントします。**for...to** 構文ではカウンタがインクリメントされ，**for...downto** 構文ではデクリメントされます。カウンタが終値と同じ値を返すと，文がもう一度実行された後，**for** 文は終了します。つまり，初期値から終値までの範囲のすべての値につき 1 回ずつ文が実行されます。初期値と終値が同じ値である場合，文は 1 回だけ実行されます。**for...to** 文で初期値が終値よりも大きい場合，または **for...downto** 文で初期値が終値よりも小さい場合，文は一度も実行されません。**for** 文の終了後，カウンタの値は未定義になります。

ループの実行を制御するために，初期値と終値の各式はループの実行前に一度だけ評価されます。したがって，**for...to** 文は次の **while** 構文にきわめて似ていますが，まったく同じではありません。

```
begin
  カウンタ := 初期値;
  while カウンタ <= 終値 do
  begin
    文;
    カウンタ := Succc( カウンタ );
  end;
end
```

この構文と **for...to** 文の違いは，**while** ループで各繰り返しの前に終値が評価し直される点です。そのため，終値が複雑な式である場合は，実行速度が著しく低下する場合があります。また，文の内部で終値の値を変更するとループの実行に影響を与える可能性があります。

for 文の例には次のものがあります。

```
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 J := 1 to 10 do
  begin
    X := 0;
    for K := 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);
```

ブロックとスコープ

宣言と文はブロックに整理されます。ブロックはラベルと識別子のローカルな名前空間（スコープ）を定義します。このため，同一の識別子（変数名など）がプログラムの部分によって異なる意味を持つことができます。各ブロックはプログラム，関数，または手続きの宣言の一部になっています。プログラム，関数，または手続きの各宣言には対応する 1 つのブロックがあります。

ブロック

ブロックは一連の宣言とその後に続く複合文で構成されます。宣言はすべてブロックの先頭に記述する必要があります。したがって，ブロックは次の形をとります。

```
宣言
begin
  文
end
```

宣言部には，変数，定数（リソース文字列を含む），型，手続き，関数，およびラベルの宣言を任意の順序で記述できます。プログラムブロックの場合は，宣言部に 1 つまたは複数の **exports** 節を記述できます。第 9 章「ライブラリとパッケージ」を参照してください。

たとえば次の関数宣言があるとします。

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

宣言の 1 行目は関数ヘッダーであり，これに続くすべての行がブロックを構成します。Ch，L，Source，および Dest はローカル変数です。これらの変数の宣言は UpperCase 関数のブロックにのみ適用され，プログラムブロックまたはユニットのインターフェース部か実現部で同じ識別子が宣言されている場合は，このブロック内でのみそれらの宣言に優先します。

スコープ

変数名や関数名などの識別子は，宣言のスコープ内でのみ使用できます。スコープは宣言の場所によって決定されます。プログラム宣言，関数宣言，または手続き宣言で宣言された識別子のスコープは，宣言の行われたブロックに限定されます。ユニットのインターフェース部で宣言された識別子のスコープは，宣言の行われたユニットを使用するほかのユニットやプログラムを含みます。スコープの狭い識別子，特に関数や手続きで宣言された識別子は，ローカルな識別子と呼ばれます。スコープの広い識別子はグローバルな識別子と呼ばれます。

識別子のスコープを決定する規則を以下に示します。

識別子が宣言された場所	スコープの範囲
プログラム，関数，または手続きの宣言	識別子が宣言された場所から現在のブロックの末尾まで。このスコープに含まれるすべてのブロックを含む
ユニットのインターフェース部	識別子が宣言された場所からユニットの末尾までと，このユニットを使用するほかのユニットまたはプログラム（第 3 章「プログラムとユニット」を参照）
ユニットの実現部。ただし関数または手続きのブロック以外	識別子が宣言された場所からユニットの末尾まで。この識別子は，ユニット内のすべての関数または手続きから使用でき，（もしあれば）初期化部および終了処理部からも使用できる
レコード型の定義。識別子はレコードのフィールドの名前である	識別子が宣言された場所からレコード型定義の末尾まで（5-21 ページの「レコード型」を参照）
クラスの定義。識別子はクラスのデータフィールドプロパティまたはメソッドの名前である	識別子が宣言された場所からクラス型の定義の末尾まで。宣言が行われたクラスの下位クラスも含み，宣言が行われたクラスとその下位クラスのすべてのメソッドのブロックも含む（第 7 章「クラスとオブジェクト」を参照）

名前の衝突

あるブロックが内部に別のブロックを含んでいる場合，前者を外部ブロック，後者を内部ブロックと呼びます。外部ブロックで宣言された識別子が内部ブロックで宣言し直された場合は，宣言した場所から内部ブロックの末尾まで内部の宣言が外部の宣言に優先し，識別子の意味を決定します。たとえば，ユニットのインターフェース部で MaxValue という変数を宣言し，同じユニットの関数宣言で同名の別の変数を宣言した場合，関数ブロック内でこの識別子が限定子なしで使用されるときは，2 番目のローカルな宣言が識別子の意味を決定します。同じように，関数の内部で別の関数を宣言すると新しい内部スコープが作成され，この内部スコープでは外側の関数で使用されている識別子をローカルに宣言し直すことができます。

複数のユニットを使用する場合は，スコープの定義がさらに複雑になります。**uses** 節に書いた各ユニットは，使われる残りのユニットと，**uses** 節を含むプログラムまたはユニットを包含する新しいスコープを有効にします。**uses** 節の最初のユニットは最も外側のスコープを表し，それに続く各ユニットは前のユニットが表すスコープの内側の新しいスコープを表します。2 つ以上のユニットのインターフェース部で同じ識別子が宣言されている場合，その識別子を限定子なしで参照すると，もっとも内側のスコープの宣言が選択されます。もっとも内側のスコープとは，識別子を参照するユニットのスコープか，そのユニットで識別子が宣言されていない場合は **uses** 節で最後に識別子を宣言したユニットが表すスコープです。

System ユニットはすべてのプログラムおよびユニットによって自動的に使用されます。System の宣言は，コンパイラが自動的に認識する定義済みの型，ルーチン，および定数とともに，常にもっとも外側のスコープを持ちます。

スコープに関するこれらの規則をオーバーライドするには，限定識別子または **with** 文を使用します。限定識別子についての詳細は，4-2 ページの「限定付き識別子」を参照してください。**with** 文についての詳細は，4-20 ページの「with 文」を参照してください。

第5章

データ型，変数，定数

型とは，データの種類の名前です。変数を宣言するときは型を指定しなければならず，その型によって，変数に格納できる値と，変数に対して実行できる操作が決定まります。式は，関数と同じように，すべて特定の型のデータを返します。ほとんどの関数と手続きは，特定の型のパラメータを要求します。

Object Pascal は型チェックの厳密な言語であり，各種のデータ型が識別され，ある型を別の型のかわりに使用することが許容されない場合があります。このため，コンパイラによるデータの適切な処理とコードの精密な検証が可能であり，診断の困難な実行時エラーの発生が予防できるので，通常は大きな利点と言えます。ただし，より柔軟な処理が必要な場合には，厳密な型チェックを回避する機能も使用できます。このような機能には，型キャスト（4-14 ページの「型キャスト」参照），ポインタ（5-25 ページの「ポインタとポインタ型」参照），バリアント（5-30 ページの「バリアント型」参照），レコードの可変部（5-22 ページの「レコードの可変部分」参照），および変数の絶対アドレス参照（5-37 ページの「絶対アドレス」参照）があります。

型について

Object Pascal のデータ型を分類するには，以下の 3 通りの方法があります。

- 一部の型は定義済み（組み込み）の型である。コンパイラは宣言がなくてもこれらの型を自動的に認識する。このマニュアルに記載されている型のほとんどは定義済みである。それ以外の型は宣言によって作成される。このような型には，ユーザー定義の型と，製品のライブラリで定義されている型がある
- 型は基本型と汎用型に分類できる。基本型の範囲と形式は，CPU とオペレーティングシステムに関係なく，Object Pascal のいずれの処理系でも同じである。汎用型の範囲と形式はプラットフォームに依存し，処理系によって異なる可能性がある。定義済みの型のほとんどは基本型であるが，整数型，文字型，文字列型，およびポインタ型の一部は汎用型である。汎用型は最適な処理効率が得られ，可搬性に優れているので，できる限り汎用型を使うことが望ましい。ただし，

汎用型は処理系によって格納形式が異なる可能性があるので，ストリーム処理でファイルにデータを書き込む場合などに互換性の問題が発生する可能性がある

- 型は，単純型，文字列型，構造化型，ポインタ型，手続き型，およびバリアント型に分類できる。さらに，High，Low，SizeOf など一部の関数には型識別子をパラメータとして渡すことができるので，型識別子自体も特殊な型に所属していると見なすことができる

Object Pascal のデータ型の分類を以下に示します。

単純型
- 順序型
  - 整数型
  - 文字型
  - 論理型
  - 列挙型
  - 部分範囲型
- 実数型

文字列型
構造化型
- 集合型
- 配列型
- レコード型
- ファイル型
- クラス型
- クラス参照型
- インターフェース型

ポインタ型
手続き型
バリアント型
型識別子型

標準関数 SizeOf は，すべての変数と型識別子をパラメータに取ることができます。この関数は，指定された型のデータを格納するために必要なメモリ容量をバイト数で表す整数を返します。たとえば，Longint 型の変数は 4 バイトのメモリを使用するので，`SizeOf(Longint)` は 4 を返します。

以下の各節では，型の宣言について説明します。型宣言についての一般的な説明は，5-35 ページの「型の宣言」を参照してください。

単純型

単純型は，順序型と実数型とがあり，順序付けされた値の集合を定義します。

順序型

順序型には，整数型，文字型，論理型，列挙型，および部分範囲型が含まれます。順序型は，順序付けされた値の集合を定義します。この集合では，最初の値を除くすべての値の前の値と，最後の値を除くすべての値の後の値が，それぞれ一意に決まっています。さらに，すべての値が順序値を持ち，これによって型の順序付けが決定されます。ほとんどの場合，ある値の順序値が n であれば，前の値の順序値は n-1 であり，後の値の順序値は n+1 です。

- 整数型の場合，順序値は値そのものです。
- その他の順序型では，デフォルトで最初の値の順序値が 0，次の値の順序値が 1 というように決まっています。列挙型の宣言によって，このデフォルトを明示的にオーバーライドできます。
- 順序型の値と型識別子を処理するいくつかの定義済み関数があります。そのうちもっとも重要なものを以下に示します。

順序型の値と型識別子を処理するいくつかの定義済み関数があります。そのうちもっとも重要なものを以下に示します。

関数	パラメータ	戻り値	説明
Ord	順序型の式	式の値の順序値	Int64 の引数を取らない
Pred	順序型の式	式の値の前の値	
Succ	順序型の式	式の値の後の値	
High	順序型の型識別子または順序型の変数	型のもっとも大きい値	短い文字列型と配列型も処理できる
Low	順序型の型識別子または順序型の変数	型のもっとも小さい値	短い文字列型と配列型も処理できる

たとえば，Byte 型のもっとも大きい値は 255 なので，`High(Byte)` は 255 を返します。2 の後の値は 3 なので，`Succ(2)` は 3 を返します。

標準手続き Inc と Dec は順序型変数の値をインクリメントおよびデクリメントします。たとえば，`Inc(I)` は `I := Succ(I)` と同じであり，I が整数変数である場合は `I := I + 1` と同じです。

整数型

整数型は整数の一部を表します。汎用型の整数型は Integer と Cardinal です。CPU とオペレーティングシステムにとって最適な処理効率が得られるため，通常はできるだけ汎用整数型を使うようにします。次の表に，現在の 32 ビット Object Pascal コンパイラの汎用整数型の範囲と格納形式を示します。

表 5.1　Object Pascal の 32 ビット処理系の汎用整数型

型	範囲	形式
Integer	-2147483648..2147483647	符号付き 32 ビット
Cardinal	0..4294967295	符号なし 32 ビット

基本整数型には，Shortint，Smallint，Longint，Int64，Byte，Word，および Longword があります。

表 5.2　基本整数型

型	範囲	形式
Shortint	-128..127	符号付き 8 ビット
Smallint	-32768..32767	符号付き 16 ビット
Longint	-2147483648..2147483647	符号付き 32 ビット
Int64	$-2^{63}..2^{63}-1$	符号付き 64 ビット
Byte	0..255	符号なし 8 ビット
Word	0..65535	符号なし 16 ビット
Longword	0..4294967295	符号なし 32 ビット

一般に，整数に対する算術演算は Integer 型の値を返します。現在の実装では，これは 32 ビットの Longint と同じです。演算の結果が Int64 型の値となるのは，オペランドが Int64 の場合のみです。したがって，次のコードは不正な結果を返します。

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

この状況で Int64 の戻り値を取得するには，I を Int64 に型キャストします。

```
 ⋮
J := Int64(I) + 1;
```

詳細については，4-6 ページの「算術演算子」を参照してください。

メモ　整数引数を取る標準関数のほとんどは，Int64 を 32 ビットに切り捨てます。ただし，High，Low，Succ，Pred，Inc，Dec，IntToStr，および IntToHex の各ルーチンは Int64 引数を完全にサポートしています。また，Round，Trunc，StrToInt64，および StrToInt64Def の各関数は Int64 の値を返します。Ord など一部のルーチンは Int64 の値をまったく受け取ることができません。

整数型の最後の値をインクリメントするか，最初の値をデクリメントすると，結果はそれぞれ範囲の最初または最後の値に戻ります。たとえば，Shortint 型の範囲は -128..127 です。したがって，次のコードを実行すると，

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

I の値は -128 になります。ただし，コンパイラによる範囲チェックを有効にしている場合は，このコードを実行すると実行時エラーが発生します。

文字型

AnsiChar と WideChar が基本文字型です。AnsiChar の値はサイズが 1 バイト（8 ビット）の文字であり，マルチバイトの場合もあるロケール文字セットに従って順序付けされています。元々 AnsiChar は（名前からも分かるように）ANSI 文字のためにモデル化されたものですが，現在のロケール文字セットに対応するために拡張されました。

WideChar 文字は，複数のバイトを使用してすべての文字を表します。現在の実装では，WideChar はサイズが 1 ワード（16 ビット）の文字であり，Unicode 文字セットに従って順序付けされています（将来の実装では，サイズがさらに大きくなる可能性があります）。Unicode の先頭から 256 文字が ANSI 文字に対応します。

Char が汎用文字型であり，これは AnsiChar と同じです。Char の実装は変更される可能性があるので，サイズの異なる文字を処理する可能性のあるプログラムを作成するときは，ハードコーディングした定数を使用するかわりに標準関数 SizeOf を使用したほうがよいでしょう。

'A' のような長さが 1 の文字列定数は文字値を表すことができます。定義済みの関数 Chr は，AnsiChar または WideChar の範囲の整数に対応する文字値を返します。たとえば，Chr(65) は文字 A を返します。

整数と同様に，範囲の先頭または末尾を超えて文字値をデクリメントまたはインクリメントした場合，範囲チェックを有効にしていなければ，文字値はそれぞれ範囲の末尾または先頭に戻ります。たとえば，次のコードを実行したとします。

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

実行後の Letter の値は A（ASCII 65）です。

Unicode 文字についての詳細は，5-12 ページの「拡張文字セットについて」と 5-13 ページの「ヌルで終わる文字列の処理」を参照してください。

論理型

定義済みの論理型には Boolean，ByteBool，WordBool，および LongBool の 4 種類があります。Boolean が通常使用される型です。ほかの 3 つの型は，ほかの言語やオペレーティングシステムのライブラリとの互換性を提供するために用意されています。

変数のサイズは，Boolean 変数と ByteBool 変数が 1 バイト，WordBool 変数が 2 バイト（1 ワード），LongBool 変数が 4 バイト（2 ワード）です。

論理型の値は定義済みの定数 True と False で表されます。次の関係が成立します。

論理型	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 型，LongBool 型，または WordBool 型の値は，順序値が非ゼロの場合に True と見なされます。Boolean が期待される状況でこれらの値が使用された場合，コンパイラは順序値が非ゼロの値を自動的に True に変換します。

なお，これは論理値の順序値に関する説明であり，論理値そのものに関するものではありません。Object Pascal では論理型の式を整数や実数と等価に見なすことはできません。X が整数変数であるときに，次の文を記述したとします。

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

この文ではコンパイル時にエラーが発生します。この変数を論理型に型キャストしても結果は保証されませんが，以下の方法を使うと問題を解決できます。

```
if X <> 0 then ...;        { 論理型の値を返す式を使用する }
var OK: Boolean            { 論理型の変数を使用する }
 ⋮
if X <> 0 then OK := True;
if OK then ...;
```

列挙型

列挙型では，値を表す識別子を列挙して，順序付けされた値の集合を定義します。値そのものが意味を持つものではありません。列挙型を宣言するには，次の構文を使用します。

type 型名 = (値$_1$，…，値$_n$)

型名と各値には有効な識別子を使用します。たとえば，次の宣言があるとします。

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

これは，Club，Diamond，Heart，または Spade という値を取ることのできる Suit という列挙型を定義します。この場合，Ord(Club) は 0 を返し，Ord(Diamond) は 1 を返すというようになります。

列挙型を宣言すると，各値が型名という型の定数であることが宣言されます。値の識別子が同じスコープ内で別の目的に使用されている場合は，名前の衝突が発生します。たとえば，次の型を宣言したとします。

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

残念ながら，Click は VCL および CLX の TControl とそのすべての下位オブジェクトで定義されているメソッドの名前でもあります。アプリケーションを作成するときに次のようなイベントハンドラを書いたとします。

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

この場合はコンパイル時にエラーが発生します。手続きのスコープ内の Click を，コンパイラが TForm の Click メソッドへの参照として解釈するためです。識別子を限定すると，この問題を回避できます。TSound が MyUnit で宣言されている場合は，次のように指定します。

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

しかし，ほかの識別子と衝突する可能性の少ない定数名を選択するほうが，解決策としては優れています。例を以下に示します。

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

(値 $_1$，...，値 $_n$) という構文は，型の名前と同じように変数宣言で直接使用することができます。

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

しかし，この方法で MyCard を宣言すると，同じ定数識別子を使って同一スコープ内で別の変数を宣言することはできません。したがって次のコードでは，

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

この文ではコンパイル時にエラーが発生します。しかし，

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

これは正しくコンパイルされます。次も同様です。

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

明示的に順序値を割り当てられた列挙型

デフォルトでは，列挙型の値の順序値は 0 から始まり，型宣言で識別子を記述したときと同じ順序になります。これを明示的にオーバーライドするには，宣言において一部またはすべての値に順序値を割り当てます。順序値を割り当てるには，その識別子の後ろに = constantExpression を記述します。ここで constantExpression は整数に評価される定数式です（5-40 ページの「定数式」を参照）。次に例を示します。

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

これは，Small，Medium，および Large という値を取ることのできる Size という型を定義します。この場合，`Ord(Small)` は 5 を返し，`Ord(Medium)` は 10，そして `Ord(Large)` は 15 を返します。

列挙型は本質的には部分範囲型であり，その上限と下限の値は，宣言における定数の順序値の上限と下限の値によって決まります。上記の例で Size 型が取ることのできる値は 11 個で，順序値の範囲は 5 ～ 15 です（したがって，`array[Size] of Char` という型は 11 文字の配列を示します）。これらの値のうち名前が付いているのは 3 つのみです。ほかの値は型キャストやルーチン（Pred，Succ，Inc，Dec など）を介してアクセスできます。次の例では，Size の範囲内の「無名」値が変数 X に割り当てられています。

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

順序値が明示的に割り当てられていない値は，リストの中の直前の値の順序値より 1 つ大きい値になります。最初の値に順序値が割り当てられていない場合は，その順序値は 0 になります。したがって，次のように宣言した場合，SomeEnum が取ることのできる値は 2 つのみです。

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

Ord(e1) は 0 を返し，Ord(e2) と Ord(e3) は両方とも 1 を返します。e2 と e3 は順序値が等しいので同じ値を示します。

部分範囲型

部分範囲型は，別の順序型の値の部分集合を表します。部分集合の基となる順序型は基本型と呼ばれます。Low と High が同じ順序型の定数式であり，Low が High よりも小さい場合，Low..High という形式の構文は Low から High までのすべての値を含む部分範囲型を表します。たとえば，次の列挙型を宣言したとします。

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

この場合は次のような部分範囲型を定義できます。

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

TMyColors には Green，Yellow，Orange，Purple，および White の値が含まれます。

数値定数と文字（長さが 1 の文字列定数）を使用して部分範囲型を定義することもできます。

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

数値定数や文字定数を使用して部分範囲型を定義するときは，指定した範囲を含むもっとも小さい整数型または文字型が基本型になります。

Low..High という構文はそれ自体が型の名前として機能するので，変数の宣言で直接使用できます。次に例を示します。

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

これは 1 ～ 500 の範囲の任意の値を取る整数変数を宣言します。

部分範囲型の各値の順序値には，基本型における順序値が使用されます。1 番目の例では，変数 Color に Green という値が格納されている場合，Color の型が TColors であるか TMyColors であるかに関係なく，Ord(Color) は 2 を返します。部分範囲型の先頭または末尾を超えてインクリメントまたはデクリメントを行った場合は，基本型が整数型または文字型の場合でも，値が部分範囲の末尾または先頭に戻ることはありません。値の型が部分範囲型から基本型に変換されるだけです。次のコードがあるとします。

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

ここではエラーが発生します。

```
 ⋮
I := 99;
Inc(I);
```

ここでは I に 100 が代入されます。ただし，コンパイラの範囲チェックを有効にしている場合を除きます。

部分範囲の定義で定数式を使用すると，構文上の問題が発生する場合があります。型宣言で = の次の最初の意味を持つ文字が左カッコである場合，コンパイラは列挙型が定義されているものと見なします。次のコードがあるとします。

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

ここではエラーが発生します。この問題を回避するには，型宣言を書き直して先頭がカッコにならないようにします。

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

実数型

実数型は，浮動小数点表記で表すことのできる数の集合を定義します。次の表に，基本実数型の範囲と格納形式を示します。

表 5.3　基本実数型

型	範囲	有効桁数	サイズ（バイト数）
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

現在の実装では，汎用型の Real は Double と同じです。

表 5.4　汎用実数型

型	範囲	有効桁数	サイズ（バイト数）
Real	5.0×10^{-324} ..1.7×10^{308}	15 ～ 16	8

メモ　以前のバージョンの Object Pascal では，6 バイトの Real48 型が Real と呼ばれていました。従来の 6 バイト Real 型を使用するコードを再コンパイルする場合は，Real48 に変換することをお勧めします。また，**{$REALCOMPATIBILITY ON}** コンパイラ指令を使用して Real を 6 バイトの型に戻すこともできます。

次の説明は基本実数型に関するものです。

- Real48 は下位互換性のために保持されています。この型の格納形式はインテルの CPU ファミリーに固有の形式でないため，ほかの浮動小数点型よりも演算が低速になります。
- Extended を使用するとほかの浮動小数点型よりも計算精度が高くなりますが，可搬性は低下します。クロスプラットフォームで共有するデータファイルを作成する場合は，Extended の使用に注意してください。

- Comp 型はインテル CPU に固有の形式で，64 ビットの整数を表します。しかし，この型は順序型と動作が異なるので，実数型に分類されています。たとえば，Comp 型の値をインクリメントまたはデクリメントすることはできません。Comp は下位互換性のためにのみ保持されています。Int64 を使用するとより優れた性能が得られます。
- Currency 型は金額計算での丸め誤差が非常に少ない固定小数点データ型です。これは最下位 4 桁が暗黙に小数点以下の桁を表す位取り 64 ビット整数として格納されます。代入文や式でほかの実数型と混在させた場合は，Currency 型の値に対して自動的に 10000 での除算または乗算が行われます。

文字列型

文字列は一連の文字を表します。Object Pascal では以下の定義済み文字列型がサポートされています。

表 5.5　文字列型

型	最大長	必要なメモリ	用途
ShortString	255 文字	2 〜 256 バイト	下位互換性のため
AnsiString	2^{31} 文字まで	4 バイト〜 2GB	8 ビット（ANSI）文字
WideString	2^{30} 文字まで	4 バイト〜 2GB	Unicode 文字 マルチユーザーサーバーとおよび 多言語アプリケーション

AnsiString は長い文字列とも呼ばれ，ほとんどの用途でもっともよく使用される型です。

代入文や式では文字列型を混在させることができます。この場合，コンパイラは必要な変換を自動的に実行します。しかし，**var** パラメータおよび **out** パラメータとして参照渡しで関数または手続きに渡される文字列は，適切な型でなければなりません。文字列は明示的にほかの文字列型に型キャストできます（4-14 ページの「型キャスト」を参照してください）。

予約語 **string** は汎用的な型識別子のように機能します。次に例を示します。

```
var S: string;
```

これは文字列を格納する変数 S を作成します。デフォルトの **{$H+}** 状態で **string** の後に大カッコで囲まれた数字がない場合，string はコンパイラによって AnsiString と解釈されます。**string** を ShortString にするには，**{$H-}** 指令を使用します。

標準関数 Length は文字列にある文字の数を返します。SetLength 手続きは文字列の長さを調節します。詳細は，オンラインヘルプを参照してください。

文字列の比較は，対応する位置にある文字の順序関係によって定義されます。長さが等しくない 2 つの文字列の場合，長い方の文字列の文字に対応する文字が短い方の文字列になければ，長い方の文字列が短い方の文字列よりも大きくなります。たとえば，「AB」は「A」よりも大きく，`'AB' > 'A'` は True を返します。長さがゼロの文字列がもっとも小さい値をとります。

文字列変数には配列と同じように添字を付けることができます。S が文字列変数で i が整数式の場合，`S[i]` は S の i 文字目（正確には i バイト目）を表します。ShortString または AnsiString の場合，`S[i]`

は AnsiChar 型です。WideString の場合，S[i] は WideChar 型です。MyString[2] := 'A'; という文は，MyString の 2 文字目に値 A を代入します。次のコードでは，標準関数 UpCase を使用して MyString を大文字に変換します。

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

このように文字列に添字付けする場合は，文字列の末尾を超えて書き込むとアクセス違反が発生する可能性があるので注意してください。また，長い文字列の添字を **var** パラメータとして渡すとコードの効率が低下するので，このような操作は避けるようにしてください。

文字列定数など，文字列を返す任意の式は，変数に代入できます。文字列の長さは，代入が行われたときに動的に変化します。例を以下に示します。

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';              { スペース }
MyString := ' ';              { 空の文字列 }
```

詳細については，4-4 ページの「文字列」と 4-9 ページの「文字列演算子」を参照してください。

短い文字列

ShortString の長さは 0 〜 255 文字です。ShortString の長さは動的に変化しますが，メモリは 256 バイトが静的に割り当てられます。第 1 バイトには文字列の長さが格納され，残りの 255 バイトに文字を格納できます。S が ShortString 型変数の場合，Ord(S[0]) は Length(S) と同様に S の長さを返します。S[0] に値を代入すると，SetLength の呼び出しと同様に S の長さが変更されます。ShortString は 8 ビットの ANSI 文字を使用し，下位互換性のためにのみ保持されています。

Object Pascal では，長さが 0 〜 255 文字の短い文字列型をサポートしています。これらは ShortString の下位の型として機能します。このような型は，大カッコで囲んだ数字を予約語 **string** の後に追加して表します。次に例を示します。

```
var MyString: string[100];
```

これは最大長が 100 文字の変数 MyString を作成します。これは次の宣言と同じです。

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

このように宣言した型には，必要なメモリだけが割り当てられます。つまり，指定した最大長に 1 バイトを加算したメモリが割り当てられます。定義済みの ShortString 型の変数では 256 バイトが使用されるのに対し，この例の MyString では 101 バイトが使用されます。

短い文字列型変数に値を代入する場合，その型の最大長よりも文字列が長ければ，文字列は切り捨てられます。

標準関数の High と Low は，短い文字列型の識別子と変数にも使えます。High は短い文字列型の最大長を返し，Low はゼロを返します。

長い文字列

AnsiString は長い文字列型とも呼ばれ，動的に割り当てられる文字列であり，最大長は使用可能なメモリによってのみ制限されます。AnsiString では 8 ビット（ANSI）文字が使用されます。

長い文字列型の変数は，4 バイトのメモリを占有するポインタです。変数が空の場合，つまり長さがゼロの文字列が格納されている場合は，ポインタは **nil** であり，ポインタ以外のメモリは使用されません。変数が空でない場合，文字列ポインタは動的メモリブロックを指します。このメモリブロックには，文字列値，文字列の長さを表す 32 ビット値，および 32 ビットの参照カウンタが格納されます。このメモリはヒープに割り当てられますが，メモリ管理はすべて自動的に行われ，ユーザーコードによる管理は不要です。

長い文字列型の変数はポインタであるので，2 つ以上の変数が追加のメモリを使用することなく同じ値を参照することができます。コンパイラはこれを利用してリソースを節約し，代入の実行速度を向上させます。長い文字列型変数が破棄されたり新しい値が代入されたりした場合，元の文字列（変数の以前の値）の参照カウンタがデクリメントされ，新しい値が存在する場合はその参照カウンタがインクリメントされます。文字列の参照カウントがゼロになると，メモリの割り当ては解除されます。この処理は参照のカウントと呼ばれます。添字付けを使って文字列の中にある 1 文字の値を変更するときは，文字列値の参照カウンタが 1 より大きい場合にのみ，文字列値のコピーが作成されます。これは書き込み時コピーと呼ばれます。

WideString

WideString 型は，16 ビット Unicode 文字から成る動的に割り当てられた文字列を表します。この型はほとんどの点で AnsiString に似ています。

Win32 では，WideString は COM の BSTR 型と互換性があります。Borland の開発ツールには，AnsiString 値を WideString 値に変換する機能をサポートしていますが，文字列を WideString に明示的にキャストまたは変換する必要がある場合があります。

拡張文字セットについて

Windows および Linux の両方で，1 バイト文字セット，マルチバイト文字セット，および Unicode がサポートされています。1 バイト文字セット（SBCS）の場合は，文字列の各バイトが 1 つの文字を表します。欧米の多くのオペレーティングシステムで使用されている ANSI 文字セットはシングルバイト文字セットです。

マルチバイト文字セット（MBCS）の場合，一部の文字は 1 バイトで表され，その他の文字は複数バイトで表されます。マルチバイト文字の 1 バイト目はリードバイトと呼ばれます。一般に，マルチバイト文字セットの下位 128 文字は 7 ビットの ASCII 文字にマップし，128 以上の順序値を持つバイトはマルチバイト文字のリードバイトです。ヌル値（#0）を含むのは 1 バイト文字だけです。マルチバイト文字セット，特に 2 バイト文字セット（DBCS）は，アジアの諸言語を表すために広く使用されています。

1 バイト文字セット，マルチバイト文字セットとも，どの言語を表現しているのかを示すエンコーディング規則が定められていないと，他の機種や他の環境との情報交換ができなくなります。エンコーディング規則とは，ある文字を表現するコードを決めるための規則です。言語によってその規則は異なり，歴史的事情から 1 つの言語で複数の規則が存在する場合もあります。日本語の場合には，Windows 環境では Shift-JIS エンコードが使われ，Linux では EUC-JP が使われています。

Unicode 文字セットではすべての文字が 2 バイトで表されます。したがって，Unicode の文字列はバイトの列ではなく，2 バイトのワードの列です。Unicode エンコーディングでは，特定の文字とそれを表すコードは 1 対 1 に定められています。そのため，動作環境に関わらず特定の文字を表現することが可能となります。Unicode の文字と文字列は，ワイド文字およびワイド文字列とも呼ばれます。Unicode の先頭から 256 文字は ANSI 文字セットにマップします。Windows オペレーティングシステムは Unicode（USC-2）をサポートしています。Linux オペレーティングシステムは，UCS-2 の上位セットである UCS-4 をサポートしています。Delphi/Kylix は，両方のプラットフォームで UCS-2 をサポートします。

Object Pascal では，Char，PChar，AnsiChar，PAnsiChar，および AnsiString の各型をとおして 1 バイトとマルチバイトの文字と文字列をサポートしています。マルチバイト文字列では，S[i] が表すのは S の i バイト目であり，それが i 文字目とは限らないので，添字指定を 1 バイト文字列と同様に扱うことはできません。しかし，標準の文字列処理関数には，同じ機能を持つマルチバイト対応の関数があります。これらの関数では，ロケール固有の文字の順序付けも実装されています。通常，マルチバイト関数の名前は Ansi で始まります。たとえば，StrPos のマルチバイトバージョンは AnsiStrPos です。マルチバイト文字のサポートは，オペレーティングシステムに依存し，現在のロケールに基づきます。

Object Pascal では，WideChar 型，PWideChar 型，および WideString 型によって Unicode 文字と文字列をサポートしています。

ヌルで終わる文字列の処理

C や C++ など多くのプログラミング言語には文字列専用のデータ型がありません。これらの言語と，これらの言語で構築されている環境は，ヌルで終わる文字列を使用します。ヌルで終わる文字列は，添字がゼロから始まってヌル文字（#0）で終わる文字配列です。この配列には長さを示すデータが含まれないため，最初のヌル文字が文字列の終端を表します。ヌルで終わる文字列を使用するシステムとデータを共有する必要がある場合は，Object Pascal の構文と SysUtils ユニットの特殊ルーチンを使用してヌルで終わる文字列を処理できます（詳細については第 8 章「標準ルーチンと入出力」を参照してください）。

たとえば，次の型宣言を使用するとヌルで終わる文字列を格納できます。

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

{$X+} で拡張構文が有効になっている場合は，静的に割り当てられた添字ゼロで始まる文字配列には文字列定数を代入できます。動的に割り当てられた配列には文字列定数を代入できません。配列定数を初期化するときに，宣言した配列の長さよりも短い文字列を使用すると，残りの文字は #0 になります。配列についての詳細は，5-17 ページの「配列型」を参照してください。

ポインタ，配列，文字列定数の使用

ヌルで終わる文字列を操作するには，多くの場合，ポインタを使用する必要があります。（5-25 ページの「ポインタとポインタ型」参照）。文字列定数は PChar 型および PWideChar 型と代入互換です。これらの型は，ヌルで終わる Char 値と WideChar 値の配列を指すポインタを表します。次に例を示します。

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

これは，ヌルで終わる「Hello world!」のコピーが格納されているメモリ領域を P が指すようにします。これは次と同じです。

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

また，PChar 型または PWideChar 型の値パラメータまたは **const** パラメータをとる関数に文字列定数を渡すこともできます。たとえば，`StrUpper('Hello world!')` は可能です。PChar への代入と同じように，コンパイラは文字列のヌルで終わるコピーを生成し，このコピーを指すポインタを関数に渡します。PChar または PWideChar の独立した定数または構造化型に含まれる定数を文字列リテラルで初期化することもできます。例を以下に示します。

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

添字がゼロから始まる文字配列は PChar および PWideChar と互換性があります。文字配列をポインタ値のかわりに使うと，コンパイラはその文字配列をポインタ定数に変換します。ポインタ定数の値はその文字配列の最初の要素のアドレスを指します。次に例を示します。

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

このコードは同じ値を使用して SomeProcedure を 2 度呼び出します。

文字ポインタは配列と同じように添字付けすることができます。前の例では，`MyPointer[0]` は H を返します。添字は，逆参照する前にポインタに加算するオフセットを指定します。PWideChar 型の変数の場合，添字には自動的に 2 が乗じられます。P が文字ポインタであれば，P`[0]` は P^ と同じで配列の 1 文字目を指し，P`[1]` は配列の 2 文字目を指します。P`[-1]` は P`[0]` の左の「文字」を指します。これらの添字について，コンパイラによる範囲チェックは行われません。

次の StrUpper 関数では，ヌルで終わる文字列を 1 文字ずつ処理するためにポインタの添字を使う方法を示します。

```
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;
```

Pascal 文字列とヌルで終わる文字列の混在

長い文字列（AnsiString 値）とヌルで終わる文字列（PChar 値）は，式と代入で混在させることができます。長い文字列型のパラメータをとる関数または手続きに PChar 値を渡すこともできます。S が文字列変数，P が PChar 式である場合，S := P という代入を行うと，ヌルで終わる文字列が長い文字列にコピーされます。

二項演算の一方のオペランドが長い文字列型でもう一方が PChar 型の場合，PChar オペランドは長い文字列型に変換されます。

PChar 値は長い文字列に型キャストできます。2 つの PChar 値に対して文字列演算を行う場合に便利です。次に例を示します。

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

長い文字列をヌルで終わる文字列に型キャストすることもできます。以下の規則が適用されます。

- S が長い文字列型の式である場合，`PChar(S)` は S をヌルで終わる文字列に型キャストし，S の 1 番目の文字を指すポインタを返す。

  Windows の場合：たとえば，Str1 と Str2 が長い文字列である場合，Win32 API の MessageBox 関数を次のように呼び出すことができる。

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

  （MessageBox の宣言は Windows インターフェースユニットにある）

  Linux の場合：たとえば，Str が長い文字列である場合，opendir 関数を次のように呼び出すことができる。

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

  （opendir は Libc インターフェースユニットで宣言されている）

- `Pointer(S)` を使用して長い文字列を型なしポインタに型キャストすることもできる。ただし，S が空である場合は **nil** が返される
- 長い文字列型の変数をポインタに型キャストする場合は，変数に新しい値が代入されるまで，または変数がスコープから出るまで，ポインタは有効である。その他の長い文字列型の式をポインタに型キャストする場合，ポインタは型キャストが実行される文の中でのみ有効である

- 長い文字列型の式をポインタに型キャストする場合，通常はそのポインタを読み出し専用と見なす必要がある。以下の条件がすべて満たされている場合のみ，ポインタを使用して長い文字列を安全に変更できる。
  - 型キャストされる式が長い文字列型の変数である
  - 文字列が空でない
  - 文字列がユニークである。つまり，参照カウントが 1 である。文字列を確実にユニークな文字列にするには，SetLength，SetString，または UniqueString のいずれかの手続きを呼び出す
  - 型キャスト以後，文字列が変更されていない
  - 変更されるすべての文字が文字列内にある。範囲外の添字を使用しないように注意する

WideString 値と PWideChar 値を混在させる場合も同じ規則が適用されます。

構造化型

構造化型のインスタンスには，複数の値が格納されます。構造化型には，集合型，配列型，レコード型，ファイル型，クラス型，クラス参照型，およびインターフェース型があります。クラス型とクラス参照型についての詳細は，第 7 章「クラスとオブジェクト」を参照してください。インターフェース型についての詳細は，第 10 章「オブジェクトインターフェース」を参照してください。集合型には順序値のみを格納できますが，それ以外の構造化型にはほかの構造化型を格納することができます。構造のレベル数に制限はありません。

デフォルトでは，構造化型の値はアクセスを速くするためにワード境界またはダブルワード境界にアラインメントされます。構造化型を宣言するときは，予約語 **packed** を使用するとデータを圧縮して格納できます。次に例を示します。

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

packed を使用するとデータへのアクセス速度が低下し，文字配列の場合は型の互換性に影響を与えます。詳細については，第 11 章「メモリ管理」を参照してください。

集合型

集合は，同一の順序型の値の集まりです。値に順序はありません。また，同じ値を集合に 2 度格納することに意味はありません。

集合型の値の範囲は，基本型と呼ばれる特定の順序型のべき集合です。つまり，集合型が取りうる値は基本型の取りうる値のすべての部分集合であり，これには空集合も含まれます。基本型の取りうる値の数が 256 を超えることはできません。また，基本型が取りうる値の順序値は 0 ～ 255 の範囲におさまらなければなりません。次の形式の構文では，

```
set of 基本型
```

基本型は適切な順序型であり，1 つの集合型を識別します。

基本型のサイズに制限があるので，通常，集合型は部分範囲を指定して定義されます。たとえば，次の宣言があるとします。

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

これは 1 ～ 250 の整数の集まりを値とする TIntSet という集合型を作成します。次のように宣言しても同じことができます。

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

この宣言を使用すると，以下の集合を作成できます。

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

set of ... 構文を変数の宣言で直接使用することもできます。

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

集合型の例には，以下のものもあります。

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

in 演算子は，ある値が集合に含まれているかどうかをテストします。

```
if 'a' in MySet then ... { 処理を行う } ;
```

すべての集合型は空集合を保持できます。空集合は [] で表されます。集合型についての詳細は，4-13 ページの「集合構成子」と 4-10 ページの「集合演算子」を参照してください。

配列型

配列型は，基本型と呼ばれる同じ型の要素の添字付きの集まりを表します。配列の要素にはすべて一意的な添字が付けられているため，集合型とは異なり，同一の値を 2 回以上格納することには意味があります。配列は静的にも動的にも割り当てることができます。

静的配列

静的な配列型は，次の形式の構文で表されます。

```
array[ 添字型1, ..., 添字型n] of 基本型
```

添字型は範囲が 2GB を超えない順序型です。添字型は配列の添字付けに使用されるので，配列に格納できる要素の数は各添字型のサイズの積によって決定されます。通常，添字型には整数の部分範囲が使用されます。

もっとも単純な一次元配列の場合，添字型は 1 つだけです。次に例を示します。

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

これは文字値 100 個の配列を格納する MyArray という変数を宣言します。このように宣言されている場合，`MyArray[3]` は MyArray の 3 文字目を表します。作成した静的配列の一部の要素に値を代入しなかった場合，未使用の要素にもメモリは割り当てられます。ただし，初期化されていない変数と同様に，これらの要素の内容はランダムなデータになります。

多次元配列は配列の配列です。次に例を示します。

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

したがって，上の宣言と下の宣言は同じ意味になります。

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

どちらの方法で宣言しても，TMatrix は 500 個の実数値の配列を表します。TMatrix 型の変数 MyMatrix に添字付けする方法は，`MyMatrix[2,45]` と `MyMatrix[2][45]` の 2 種類があります。また，次の宣言があるとします。

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

したがって，上の宣言と下の宣言は同じ意味になります。

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

標準関数の Low と High は，配列型の識別子と変数にも使えます。これらの関数は配列の 1 番目の添字型の下限と上限を返します。標準関数 Length は配列の第 1 次元の要素の数を返します。

Char 値の静的な一次元パック配列は，パック文字列と呼ばれます。パック文字列型は文字列型と互換性があり，要素数が同じであるほかのパック文字列型とも互換性があります。5-33 ページの「型の互換性と同一性」を参照してください。

`array[0..x] of Char` という形式の配列型は，添字がゼロから始まる文字配列と呼ばれます。添字がゼロから始まる文字配列は，ヌルで終わる文字列の格納に使用され，PChar 値と互換性があります。5-13 ページの「ヌルで終わる文字列の処理」を参照してください。

動的配列

動的配列はサイズや長さが固定されていません。動的配列のメモリは，配列に値を代入したとき，または SetLength 手続きに配列を渡したときに割り当てられます。動的な配列型は，次の形式の構文で表されます。

```
array of 基本型
```

次に例を示します。

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

これは実数の一次元配列を宣言します。この宣言は MyFlexibleArray のメモリ割り当てを行いません。メモリ内に配列を作成するには SetLength を呼び出します。前の宣言がある場合，次の文は，

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

添字が 0 〜 19 の実数 20 個の配列を割り当てます。動的配列の添字は常に整数であり，0 から始まります。

動的配列変数は暗黙にポインタであり，長い文字列型と同じ参照カウントの方法を使用して管理されます。動的配列の割り当てを解除するには，配列を参照する変数に **nil** を代入するか，Finalize に変数を渡します。配列がほかで参照されていなければ，これらの方法のどちらでも配列が破棄されま

す。長さが 0 である動的配列の値は **nil** です。動的配列変数に逆参照演算子（^）を使用してはいけません。また，動的配列変数を New 手続きまたは Dispose 手続きに渡してはいけません。

X と Y が同じ動的配列型の変数である場合，X := Y は，X が Y と同じ配列を指すようにします（この処理を行う前に，X 用にメモリを割り当てる必要はありません）。文字列や静的配列と異なり，動的配列が書き込み時に自動的にコピーされることはありません。たとえば，次のコードを実行したとします。

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

実行後の `A[0]` の値は 2 です。A と B が静的配列であれば，`A[0]` は 1 のままです。

添字付けによる動的配列への代入（`MyFlexibleArray[2] := 7` など）を行っても，配列の再割り当ては行われません。範囲外の添字を使用しても，コンパイル時には報告されません。

動的配列変数の比較では，配列の値ではなくポインタが比較されます。たとえば，次のコードを実行したとします。

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

この場合，`A = B` は False を返しますが，`A[0] = B[0]` は True を返します。

動的配列を切り捨てるには，SetLength 手続きまたは Copy 関数に配列変数を渡し，結果を配列変数に代入します（通常は SetLength 手続きの方が高速です）。たとえば，A が動的配列である場合，`A := SetLength(A, 0, 20)` は A の 21 番目以降の要素を切り捨てます。

割り当て済みの動的配列は，標準関数 Length，High，および Low に渡すことができます。Length は配列の要素数を返します。High は配列の添字の上限，つまり Length - 1 を返します。Low は 0 を返します。配列の要素数がゼロの場合，High は -1 を返すので，High < Low という特異な結果になります。

メモ 一部の関数と手続きの宣言では，配列パラメータが「`array of 基本型`」として宣言されており，添字型が指定されていません。次に例を示します。

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

これは，配列のサイズ，添字付けの方法，静的か動的かの区別に関係なく，指定された基本型のすべての配列をその関数が処理できることを示します。6-15 ページの「オープン配列パラメータ」を参照してください。

多次元動的配列

多次元動的配列を宣言するには，**array of ...** 構文を重ねて使用します。次に例を示します。

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

これは文字列の二次元配列を宣言します。この配列をインスタンス化するには，2 つの整数引数を付けて SetLength を呼び出します。たとえば，I と J が整数の値を持つ変数であるとします。

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

これは I × J の配列を割り当てます。`Msgs[0,0]` はこの配列の要素を表します。

特定の次元の配列の長さがすべて同じではない多次元動的配列を作成することもできます。まず SetLength を呼び出し，配列の最初の n 次元を指定するパラメータを渡します。次に例を示します。

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

これは Ints に 10 行を割り当てますが，列は割り当てません。次に，列を 1 つずつ割り当て，それぞれ異なる長さを指定します。次に例を示します。

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

これは Ints の第 3 列の長さを整数 5 つ分にします。この時点で，ほかの列をまだ割り当てていなくても，第 3 列に値を代入できます。たとえば，`Ints[2,4] := 6` とすることができます。

次の例では，動的配列と SysUtils ユニットで宣言されている IntToStr 関数を使用して，三角形の文字列行列を作成します。

```
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;
```

配列の型と代入

配列は同一の型である場合にだけ代入互換です。Pascal では型の名前に基づいて型の一致を判定するので，次のコードはコンパイルできません。

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

代入を行えるようにするには，次のように変数を宣言します。

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

または

```
type IntArray = array[1..10] of Integer;
```

```
var
  Int1: IntArray;
  Int2: IntArray;
```

レコード型

レコードは，ほかの言語では構造体とも呼ばれ，複数の型を持つ要素の集合を表します。各要素はフィールドと呼ばれ，レコード型の宣言では各フィールドの名前と型を指定します。レコード型の宣言の構文を次に示します。

```
type レコード型名 = record
  fieldList1: type1;
  ⋮
  fieldListn: typen;
end
```

レコード型名には有効な識別子を指定し，フィールド型には型を指定し，フィールドリストには有効な識別子またはカンマ区切りの識別子のリストを指定します。末尾のセミコロンはオプションです。

たとえば，次の宣言は TDateRec というレコード型を作成します。

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

TDateRec 型の各変数には，Year という整数値，Month という列挙型の値，および Day という 1 〜 31 の整数値の 3 つのフィールドがあります。Year，Month，および Day は TDateRec のフィールド指定子であり，変数と同じように機能します。ただし，TDateRec の型宣言では Year，Month，および Day の各フィールドのためのメモリは割り当てられません。メモリの割り当ては，次のようにレコードをインスタンス化したときに行われます。

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

この変数宣言では，Record1 および Record2 という TDateRec 型の 2 つのインスタンスが作成されます。

レコードのフィールドにアクセスするには，フィールド指定子をレコード名で限定します。

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

with 文を使用する方法もあります。

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

この時点で Record1 のフィールドの値を Record2 にコピーできます。

```
Record2 := Record1;
```

フィールド指定子のスコープはフィールドが存在するレコードに限定されているので，フィールド指定子とほかの変数の間で名前の衝突が発生することはありません。

レコード型を定義するかわりに，変数の宣言で **record ...** 構文を直接使用することもできます。

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

ただし，レコード型の長所は類似する変数のグループを繰り返しコーディングしなくても済むことであるので，このように宣言するとレコード型を使用する意味がほとんどなくなります。また，このように個別に宣言されたレコード型は，構造がまったく同じであっても，代入互換でなくなります。

レコードの可変部分

レコード型は可変部分を持つことができます。可変部分は **case** 文と似ています。可変部分はレコード宣言でほかのフィールドよりも後に記述しなければなりません。

可変部分を持つレコード型を宣言するには，次の構文を使用します。

```
type レコード型名 = record
  フィールドリスト1: フィールド型1;
  ⋮
  フィールドリストn: フィールド型n;
case タグ : タグフィールド型 of
  定数リスト1: (可変部1);
  ⋮
  定数リストn: (可変部n);
end;
```

宣言のうち，予約語 **case** までの前半部分は，標準的なレコード型と同じです。宣言の残り，つまり **case** から末尾のオプションのセミコロンまでが可変部分と呼ばれます。可変部分には以下の規則が適用されます。

- タグはオプションで，任意の有効な識別子を使用できる。タグを省略する場合は，後のコロン（**:**）も省略する
- タグフィールド型には，順序型の型識別子を指定する
- 定数リストは，タグフィールド型の値を表す定数か，そのような定数のカンマ区切りのリストである。n 個ある定数リストの中で，同じ値を 2 回以上使用してはならない
- 各可変部は，レコード型の固定部の「フィールドリスト：フィールド型」構文に似たカンマ区切りの宣言のリストである。つまり，可変部は次の形式をとる。

```
フィールドリスト1: フィールド型1;
 ⋮
フィールドリストn: フィールド型n;
```

フィールドリストには有効な識別子またはカンマ区切りの識別子のリストを指定し，フィールド型には型を指定する。最後のセミコロンはオプションである。フィールド型は，長い文字列型，動的配列型，Variant 型，またはインターフェース型であってはならない。また，長い文字列型，動的配列型，Variant 型，またはインターフェース型を含む構造体であってもならない。しかし，これらの型へのポインタは許容される

可変部分を持つレコードは，構文上は複雑ですが，その意味は非常に単純です。レコードの可変部分には，メモリの同じ領域を共有する複数の可変部が含まれます。いずれの可変部のいずれのフィールドに対しても，読み取りと書き込みがいつでも可能です。ただし，ある可変部のフィールドに書き込みを行い，次に別の可変部のフィールドに書き込んだ場合は，以前に書き込んだデータを上書きする可能性があります。タグが存在する場合は，レコードの固定部分でタグフィールド型のフィールドとして機能します。

可変部分を使用する目的は 2 つあります。第一に，さまざまな種類のデータをフィールドに格納するレコード型を作成するときに，単一のインスタンスですべてのフィールドを使用する必要が絶対にないことがわかっているとします。次に例を示します。

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

すべての従業員に年俸または時給が存在するが，両方とも存在する従業員はいない，ということに注意してください。したがって，TEmployee のインスタンスを作成するときに，両方のフィールドのためのメモリを割り当てる理由はありません。この場合，2 つの可変部で異なるのはフィールド名のみですが，フィールドの型が異なる場合もあります。より複雑な例を示します。

```
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;
```

レコード型の各インスタンスについて，コンパイラはもっとも大きい可変部のすべてのフィールドを格納できるだけのメモリを割り当てます。オプションのタグと定数リスト（最後の例の Rectangle,

Triangle など）は，コンパイラによるフィールドの保守には影響を与えません。これらはプログラマの利便のためにのみ存在します。

可変部分を使用する 2 つ目の理由は，コンパイラが型キャストを許可しないような場合でも，同じデータを異なる型のデータとして扱うことにあります。たとえば，ある可変部の 1 つ目のフィールドが 64 ビットの Real 型であり，別の可変部の 1 つ目のフィールドが 32 ビットの Integer 型である場合は，Real 型のフィールドに値を割り当ててから Integer 型のフィールドの値を読み取ることで，先頭の 32 ビットを調べることができます。読み取った値は，たとえば整数型のパラメータを取る関数に渡すことができます。

ファイル型

ファイルは，同一の型の要素の順序付けされた集合です。標準の I/O ルーチンでは，定義済みの TextFile 型（Text 型）を使用します。この型は，行単位で整理された文字が含まれるファイルを表します。ファイル入出力についての詳細は，第 8 章「標準ルーチンと入出力」を参照してください。

ファイル型を宣言するには，次の構文を使用します。

```
type ファイル型名 = file of 型
```

ファイル型名には任意の有効な識別子を使用します。ファイルの型には固定サイズの型を指定します。ポインタ型は，暗黙のものでも明示的なものでも許容されません。したがって，動的配列，長い文字列，クラス，オブジェクト，ポインタ，バリアント，ほかのファイル，またはこれらを含む構造型をファイル型に格納することはできません。

次に例を示します。

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

これは名前と電話番号を記録するためのファイル型を宣言します。

file of ... 構文を変数の宣言で直接使用することもできます。次に例を示します。

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

file のみの場合は型なしファイルを表します。

```
var DataFile: file;
```

詳細については，8-4 ページの「型なしファイル」を参照してください。

配列の要素やレコードのフィールドにファイル型を使用することはできません。

ポインタとポインタ型

ポインタはメモリアドレスを指す変数です。ポインタにほかの変数のアドレスが格納されている場合は，メモリ内の変数の場所またはそこに格納されているデータをそのポインタが指しているといいます。配列型などの構造型の場合は，1 番目の要素のアドレスがポインタに格納されます。

ポインタは，指すアドレスに格納されるデータの種類に応じて型付けされます。汎用の Pointer 型は任意の型のデータを指すポインタを表します。その他の特殊化されたポインタ型は，特定の型のデータのみを参照します。ポインタは 4 バイトのメモリを占有します。

ポインタの概要

次の例では，ポインタの機能を示します。

```
1    var
2      X, Y: Integer;   // X と Y は Integer 型の変数
3      P: ^Integer;     // P は Integer を指す
4    begin
5      X := 17;         // X に値を代入する
6      P := @X;         // X のアドレスを P に代入する
7      Y := P^;         // P を逆参照して結果を Y に代入する
8    end;
```

2 行目は X と Y を Integer 型の変数として宣言します。3 行目は Integer 値を指すポインタとして P を宣言します。このため，P は X または Y のアドレスを指すことができます。5 行目は X に値を代入し，6 行目は @X で表される X のアドレスを P に代入します。最後に，7 行目は P が指すアドレスに格納されている値（P^）を取得して Y に代入します。このコードの実行後，X と Y の値はどちらも 17 になります。

ここで変数のアドレスを取得するために使用した **@** 演算子は，関数と手続きに対しても使用できます。詳細については，4-12 ページの「@ 演算子」と 5-29 ページの「文と式での手続き型」を参照してください。

^ シンボルには 2 つの用途があり，この例ではどちらの用途でも使用されています。1 つは，型識別子の前に記述される場合です。

  ^ 型名

この場合は，指定された型の変数を指すポインタ型を表します。もう 1 つは，ポインタ変数の後に記述される場合です。

  ポインタ ^

この場合はポインタを逆参照します。つまり，ポインタが指すメモリアドレスに格納されている値を返します。

この例は，ある変数の値を別の変数にコピーする方法としては回りくどいように思われるかもしれません。単純な代入文でも同じことは実現できます。しかし，ポインタが役に立つ理由はいくつかあります。第一に，ポインタを理解することによって Object Pascal の理解が深まります。多くの場合，コードに明示的にポインタが現れない場合でも，内部的にはポインタが使用されています。大きいメ

モリブロックの動的な割り当てを必要とするデータ型は，すべてポインタを使用します。たとえば，長い文字列型の変数は暗黙のポインタであり，クラス変数も同じです。さらに，一部の高度なプログラミング技法ではポインタの使用を必要とします。

最後に，Object Pascal の厳密な型付けを回避する唯一の方法がポインタである場合もあります。汎用的な Pointer 型で変数を参照し，この Pointer 型を特定の型のポインタに型キャストしてから逆参照すると，変数に格納されているデータを任意の型のデータとして扱うことができます。たとえば，次のコードは実数型の変数に格納されているデータを整数型の変数に割り当てます。

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

もちろん，実数と整数は異なる形式で格納されています。この代入は単に R から I にバイナリデータを変換せずにコピーするものです。

ポインタに値を代入する方法としては，**@** 演算の結果を代入する方法のほかに，いくつかの標準的なルーチンがあります。New 手続きと GetMem 手続きは既存のポインタにメモリアドレスを代入します。Addr 関数と Ptr 関数は，指定されたアドレスまたは変数を指すポインタを返します。

逆参照されたポインタは限定することができ，限定子として機能することもできます。たとえば，`P1^.Data^` とすることができます。

予約語 **nil** はすべてのポインタに代入できる特殊な定数です。**nil** を代入されたポインタは何も参照しません。

ポインタ型

任意の型を指すポインタを宣言するには，次の構文を使用します。

```
type ポインタ型名 = ^型
```

一般に，レコード型などのデータ型を定義するときは，その型を指すポインタも定義します。このようなポインタ型を定義すると，大きいメモリブロックをコピーせずにその型のインスタンスを容易に操作できるようになります。

多くの用途のためのポインタ型が標準で用意されています。もっとも柔軟性が高いのは Pointer 型で，任意の型のデータを指すことができます。ただし，Pointer 変数は逆参照できません。Pointer 変数の後に ^ シンボルを記述すると，コンパイル時にエラーが発生します。Pointer 変数が参照するデータにアクセスするには，ほかのポインタ型に型キャストしてから逆参照してください。

文字ポインタ

基本型 PAnsiChar と PWideChar は，それぞれ AnsiChar 値と WideChar 値を指すポインタを表します。汎用型 PChar は Char を指すポインタを表します。現在の実装では，Char は AnsiChar と同じです。これらの文字ポインタは，ヌルで終わる文字列の操作に使用します。（5-13 ページの「ヌルで終わる文字列の処理」を参照）。

その他の標準のポインタ型

System ユニットと SysUtils ユニットでは，頻繁に使用する標準のポインタ型が多数宣言されています。

表 5.6　System と SysUtils で宣言されているポインタ型（抜粋）

ポインタ型	指す変数の型
PAnsiString，PString	AnsiString
PByteArray	TByteArray。この型は SysUtils で宣言されている。動的に割り当てられたメモリを型キャストして配列にアクセスするために使用される
PCurrency，PDouble，PExtended，PSingle	Currency，Double，Extended，Single
PInteger	Integer
POleVariant	OleVariant
PShortString	ShortString。古い PString 型を使用する従来のコードを移植するときに便利
PTextBuf	TTextBuf。この型は SysUtils で宣言されている。TTextRec ファイルレコードで使われる内部バッファの型
PVarRec	TVarRec。この型は System で宣言されている
PVariant	Variant
PWideString	WideString
PWordArray	TWordArray。この型は SysUtils で宣言されている。2 バイト値の配列のために動的に割り当てられたメモリの型キャストに使用される

手続き型

手続き型を使用すると，手続きや関数を値として扱い，変数に割り当てたりほかの手続きや関数に渡したりすることができます。たとえば，2 つの整数パラメータを受け取って整数を返す Calc という関数を定義したとします。

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

Calc 関数は変数 F に代入できます。

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

手続きまたは関数のヘッダーから **procedure** または **function** の後の識別子を取り除くと，残ったものが手続き型の名前になります。このような型の名前は，前の例のように変数宣言で直接使用したり，新しい型の宣言に使用したりすることができます。

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
```

```
    TStrProc = procedure(const S: string);
    TMathFunc = function(X: Double): Double;
  var
    F: TIntegerFunction;             { F はパラメータなしで整数を返す関数 }
    Proc: TProcedure;                { Proc はパラメータなしの手続き }
    SP: TStrProc;                    { SP は文字列パラメータをとる手続き }
    M: TMathFunc;                    { M は Double 型（実数）パラメータをとって
                                       Double を返す関数 }
  procedure FuncProc(P: TIntegerFunction);  { FuncProc は，整数を返すパラメータなしの関数を
                                              唯一のパラメータとする手続き }
```

これらの変数はすべて手続きポインタ，つまり手続きまたは関数のアドレスを指すポインタです。インスタンスオブジェクトのメソッドを参照する場合は，手続き型の名前に **of object** を追加する必要があります。インスタンスオブジェクトについての詳細は，第 7 章「クラスとオブジェクト」を参照してください。次に例を示します。

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

これらの型はメソッドポインタを表します。メソッドポインタは実際には 2 つのポインタで構成されます。最初のポインタにはメソッドのアドレスが格納され，2 番目のポインタにはメソッドが属しているオブジェクトへの参照が格納されます。次のような宣言が行われているとします。

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

この場合は，次の代入が可能です。

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

2 つの手続き型が次の条件を満たす場合は相互に互換性があります。

- 呼び出し規約が同じである
- 戻り値が同じであるか，または戻り値がない
- パラメータの数が一致し，パラメータの型と順序も一致する。パラメータの名前は関係がない

手続きポインタ型とメソッドポインタ型は常に非互換です。値 **nil** はすべての手続き型に代入できます。

ネストした手続きと関数，つまりほかのルーチンの内部で宣言されているルーチンは，手続き値としては使用できません。定義済みの手続きと関数も使用できません。Length などの定義済みの手続きや関数を手続き値として使うには，ラッパーを記述します。

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

文と式での手続き型

代入文の左辺に手続き型の変数がある場合，コンパイラは右辺に手続き型の値を必要とします。代入により，左辺の変数には右辺の関数または手続きを指すポインタが格納されます。ただし，代入以外の場合に手続き型の変数を使用すると，変数が参照する手続きまたは関数が呼び出されることになります。手続き型変数を使用してパラメータを渡すこともできます。

```
var
  F: function(X: Integer): Integer;
  I: Integer;
function SomeFunction(X: Integer): Integer;
 ⋮
F := SomeFunction;  // F に SomeFunction を代入
I := F(4);          // 関数を呼び出し，結果を I に代入
```

代入文では，左辺の変数の型によって，右辺の手続きポインタまたはメソッドポインタの解釈方法が決定されます。次に例を示します。

```
var
  F, G: function: Integer;
  I: Integer;
function SomeFunction: Integer;
 ⋮
F := SomeFunction;  // F に SomeFunction を代入
G := F;             // G に F をコピー
I := G;             // 関数を呼び出し，結果を I に代入
```

1 番目の文は手続き値を F に代入します。2 番目の文は F の値を別の変数にコピーします。3 番目の文は，参照されている関数を呼び出して結果を I に代入します。I は手続き型ではなく整数型の変数なので，最後の代入では整数を返す関数 SomeFunction が実際に呼び出されます。

場合によっては，手続き型変数の解釈方法がそれほど明確でないこともあります。次の文があるとします。

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

この場合，F の出現箇所で関数が呼び出されます。コンパイラは F が指す関数を呼び出し，次に MyFunction 関数を呼び出して，結果を比較します。手続き型の変数が式に出現した場合，これは常にその変数が参照する手続きまたは関数への呼び出しを表します。結果を返さない手続きを F が参照する場合，またはパラメータを必要とする関数を参照する場合は，コンパイル時にこの文でエラーが発生します。F の手続き値を MyFunction と比較するには，次の構文を使用します。

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

`@F` は F を，アドレスが格納される型なしポインタ変数に変換し，`@MyFunction` は MyFunction のアドレスを返します。

手続き型変数に格納されているアドレスではなく，手続き型変数のメモリアドレスを取得するには，**@@** を使用します。たとえば，`@@F` は F のアドレスを返します。

@ 演算子は型なしポインタ値を手続き型変数に代入するときにも使用できます。次に例を示します。

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

これは GetProcAddress 関数を呼び出し，その結果を StrComp が指すようにします。

値 **nil** はすべての手続き型変数に格納できます。この値は，ポインタが何も指していないことを表します。値が **nil** である手続き型変数を呼び出そうとすると，エラーが発生します。手続き型変数に値が代入されているかどうかを調べるには，標準関数 Assigned を使用します。

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

バリアント型

場合によっては，型が変化するデータまたはコンパイル時に型を決定できないデータの操作が必要なことがあります。このような場合には，Variant 型の変数とパラメータを使用することができます。Variant 型は，実行時に型が変化する値を表します。バリアント型は柔軟性に富んでいる一方，通常の変数より多くのメモリを消費し，バリアント型に対する演算は静的にバインドされた値に対する演算より実行速度が遅くなります。また，通常の変数に対する不正な操作はコンパイル時に捕捉されるのに対し，多くの場合，バリアント型に対する不正な操作は実行時エラーとなります。カスタムのバリアント型を作成することもできます。

デフォルトでは，バリアント型には，レコード，集合，静的配列，ファイル，クラス，クラス参照，およびポインタを除くすべての型の値を格納できます。つまり，バリアント型には構造型とポインタ型を除くすべての値を格納できます。バリアント型には，インターフェースを格納でき，そのメソッドとプロパティにバリアント型を通じてアクセスできます（第 10 章「オブジェクトインターフェース」を参照）。バリアント型には動的配列も格納でき，バリアント配列と呼ばれる特殊な静的配列も格納できます。（5-32 ページの「バリアント配列」を参照）。式と代入文では，バリアント型をほかのバリアント型と組み合わせることができ，整数値，実数値，文字列値，および論理値とも組み合わせることができます。コンパイラは必要な型変換を自動的に実行します。

文字列が格納されているバリアント型は添字付けできません。つまり，バリアント型の V に文字列値が格納されている場合，`V[1]` は実行時エラーとなります。

バリアント型を拡張して任意の値を格納できるようにしたカスタムバリアントを定義することができます。たとえば，インデックス付けを可能にしたり，特定のクラス参照やレコード型，静的配列を格納するバリアント文字列型を定義できます。カスタムバリアント型は，TCustomVariantType クラスの下位クラスを作成することによって定義します。

バリアント型は 16 バイトのメモリを占有し，型コードと値（または値を指すポインタ）で構成されます。値の型は，型コードが表す型です。すべてのバリアント型は作成時に特殊値 Unassigned に初期化されます。特殊値 Null は，未知のデータまたはデータが見つからないことを表します。

標準関数 VarType はバリアントの型コードを返します。varTypeMask 定数は，VarType 関数の戻り値から型コードを抽出するために使用されるビットマスクです。次に例を示します。

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

これは V に Double 型または Double 型の配列が格納されている場合に True を返します。ビットマスクは，バリアントに配列が格納されているかどうかを表す第 1 ビットを隠します。System ユニットで定義されている TVarData レコード型を使ってバリアント型の変数を型キャストすると，その内部

表現へアクセスできます。型コードの一覧は，オンラインヘルプの VarType の説明に記載されています。Object Pascal の将来の実装では新しい型コードが追加される可能性があります。

バリアント型の変換

すべての整数型，実数型，文字列型，文字型，および論理型はバリアント型と代入互換です。式は明示的にバリアント型に型キャストできます。また，標準ルーチンの VarAsType と VarCast を使用すると，バリアントの内部表現を変更できます。次のコードでは，バリアント型の使い方と，バリアント型をほかの型と混在させたときに実行される自動的な変換を示します。

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1; { 整数値 }
  V2 := 1234.5678; { 実数値 }
  V3 := 'Hello world';  { 文字列値 }
  V4 := '1000'; { 文字列値 }
  V5 := V1 + V2 + V4; { 実数値 2235.5678}
  I := V1; { I = 1 (整数値) }
  D := V2; { D = 1234.5678 (実数値) }
  S := V3; { S = 'Hello world!' (文字列値) }
  I := V4; { I = 1000 (整数値) }
  S := V5; { S = '2235.5678' (文字列値) }
end;
```

コンパイラは以下の規則に従って型変換を行います。

表 5.7　バリアントの型変換規則

変換先 変換元	**整数型**	**実数型**	**文字列型**	**文字型**	**論理型**
整数型	整数の形式を変換する	実数に変換する	文字列表現に変換する	文字列（左の項目）と同じ	0 の場合は False，それ以外の場合は True を返す
実数型	もっとも近い整数に丸める	実数の形式を変換する	地域の設定を使用して文字列表現に変換する	文字列（左の項目）と同じ	0 の場合は False，それ以外の場合は True を返す
文字列型	整数に変換する。必要な場合は切り捨てる。文字列が数値でない場合は例外を生成する	地域の設定を使用して実数に変換する。文字列が数値でない場合は例外を生成する	文字列の文字の形式を変換する	文字列（左の項目）と同じ	文字列が「false」の場合，または評価結果が 0 となる数値を表す文字列の場合は False を返す。文字列が「true」の場合，または非ゼロの数値を表す文字列の場合は True を返す。大文字と小文字は区別されない。それ以外の場合は例外を生成する
文字型	文字列（上の項目）と同じ	文字列（上の項目）と同じ	文字列（上の項目）と同じ	文字列から文字列への変換と同じ	文字列（上の項目）と同じ

表 5.7　バリアントの型変換規則（つづき）

論理型	False = 0, True = -1（Byte 型の場合は 255）	False = 0, True = -1	False =「0」, True =「-1」	文字列（左の項目）と同じ	False = False, True = True
Unassigned	0 を返す	0 を返す	空の文字列を返す	文字列（左の項目）と同じ	False を返す
Null	例外を生成する	例外を生成する	例外を生成する	文字列（左の項目）と同じ	例外を生成する

範囲外の値を代入すると，多くの場合，代入先の変数にその変数が取りうる最大の値が代入されます。無効な代入や型キャストでは EVariantError 例外が生成されます。

System ユニットで宣言されている TDateTime 実数型には特殊な変換規則が適用されます。TDateTime をほかの型に変換すると，通常の Double 型として扱われます。整数型，実数型，または論理型の値を TDateTime 型に変換するときは，まず Double 型に変換され，次に日付 / 時刻値として読み取られます。文字列を TDateTime 型に変換するときは，地域の設定を使用して日付 / 時刻値として解釈されます。Unassigned 値を TDateTime 型に変換するときは，実数値または整数値の 0 のように扱われます。Null 値から TDateTime 型への変換では例外が生成されます。

Windows では，COM インターフェースを参照するバリアントの値の変換を試みると，オブジェクトのデフォルトプロパティが読み取られ，そのプロパティ値が指定された型に変換されます。オブジェクトにデフォルトプロパティがない場合は例外が生成されます。

式におけるバリアント

^，**is**，および **in** を除き，いずれの演算子でもバリアント型のオペランドを取ることができます。バリアントに対する演算は Variant 型の値を返します。どちらかまたは両方のオペランドが Null の場合は Null を返します。両方のオペランドが Unassigned である場合は例外が生成されます。二項演算では，一方のオペランドのみがバリアント型の場合，他方のオペランドはバリアント型に変換されます。

演算の戻り値の型はオペランドによって決定されます。一般には，静的にバインドされる型のオペランドに適用されるのと同じ規則がバリアント型にも適用されます。たとえば，V1 と V2 がバリアント型であり，それぞれ整数値と実数値が格納されている場合，`V1 + V2` は実数値を持つバリアント型を返します（4-6 ページの「演算子」を参照）。ただし，バリアント型の場合は，静的に型付けされる式では許容されない値の組み合わせに対して二項演算を行うことができます。不正なバリアントの組み合わせに対し，コンパイラは表 5.7 の規則を使用して可能な限り変換を行います。たとえば，V3 と V4 がバリアント型であり，それぞれ数値を表す文字列と整数が格納されている場合，式 `V3 + V4` は整数値を持つバリアントを返します。演算の前に，数値を表す文字列は整数に変換されます。

バリアント配列

通常の静的配列をバリアントに代入することはできません。かわりに，標準関数 VarArrayCreate または VarArrayOf のどちらかを使用してバリアント配列を作成します。次に例を示します。

```
V: Variant;
  ⋮
```

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

これは，10 個の整数要素を持つバリアント配列を作成し，バリアント型の変数 V に代入します。この配列は `V[0]`，`V[1]` などのように添字付けできますが，バリアント配列の要素を **var** パラメータとして渡すことはできません。バリアント配列は常に整数で添字付けします。

VarArrayCreate の呼び出しの 2 つ目のパラメータは，配列の基本型の型コードです。これらの型コードの一覧は，オンラインヘルプの VarType の説明に記載されています。varString コードを VarArrayCreate に渡してはいけません。文字列のバリアント配列を作成するには，varOleStr を使用します。

バリアント型には，さまざまなサイズ，次元数，および基本型を持つバリアント配列を格納できます。バリアント配列の要素には，ShortString と AnsiString を除き，バリアント型で許容されるいずれの型でも使用できます。基本型が Variant 型である場合は，要素ごとに型が異なることも可能です。バリアント配列のサイズを変更するには，VarArrayRedim 関数を使用します。バリアント配列を処理するその他の標準関数には，VarArrayDimCount，VarArrayLowBound，VarArrayHighBound，VarArrayRef，VarArrayLock，および VarArrayUnlock があります。

バリアント配列が格納されているバリアントを別のバリアントへ代入する場合，または値パラメータとして渡す場合には，配列全体のコピーが作成されます。このような操作はメモリを大量に使用するので，不必要に行わないでください。

OleVariant

OleVariant 型は，Windows と Linux 両方のプラットフォームに存在します。Variant と OleVariant の主な違いは，Variant は現在のアプリケーションで操作方法が分かっているデータ型だけしか保持できないことです。OleVariant は，OLE オートメーションと互換性があるとして定義されたデータ型のみを保持できます。したがって，プログラム間またはネットワーク経由でデータを受け渡すことができ，そのとき相手側がデータの操作方法を分かっているかどうかを考慮する必要がありません。

カスタムデータ（Pascal 文字列や新しいカスタムバリアント型など）を含む Variant を OleVariant に代入すると，ランタイムライブラリはその Variant を OleVariant 標準データ型のいずれかに変換しようとします（たとえば，Pascal 文字列は OLE BSTR 文字列に変換されます)。たとえば，バリアントに AnsiString が含まれている場合，AnsiString は WideString になります。Variant を OleVarient 関数パラメータに渡す場合も同様です。

型の互換性と同一性

どのような式に対してどのような演算が可能であるかを理解するには，型と値の互換性について，いくつかの種類を区別する必要があります。これらの種類には，型の同一性，型の互換性，および代入互換性があります。

型の同一性

型の同一性に複雑な規則はほとんどありません。ある型識別子の宣言に別の型識別子が限定なしで使用されている場合，この 2 つの識別子は同じ型を表します。たとえば，次のように宣言されているとします。

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

T1，T2，T3，T4，および Integer はすべて同一の型を表します。同一でない型を作成するには，宣言で **type** をもう一度使用します。次に例を示します。

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

これは Integer と同一でない TMyInteger という新しい型を作成します。

型名として機能する構文は，出現箇所ごとに別の型を表します。次の宣言があるとします。

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

これは TS1 と TS2 という 2 つの異なる型を作成します。同様に，次の変数宣言があるとします。

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

これは異なる型の変数を 2 つ作成します。同一の型の変数を作成するには，次のように宣言します。

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

または

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

型の互換性

すべての型は同一の型と互換性があります。同一でない 2 つの型が互換性を持つのは，以下の条件の少なくとも 1 つを満たす場合です。

- 両方の型が実数型である
- 両方の型が整数型である
- 一方の型が他方の型の部分範囲である
- 両方の型が同じ型の部分範囲である
- 両方の型が互換性のある基本型を持つ集合型である
- 両方の型が同じ個数の要素を持つパック文字列型である
- 一方の型が文字列型で，他方の型が文字列型，パック文字列型，または Char 型のいずれかである
- 一方の型がバリアント型で，他方の型が整数型，実数型，文字列型，文字型，または論理型のいずれかである

- 両方の型がクラス型，クラス参照型，またはインターフェース型で，一方の型が他方の型から派生している
- 一方の型が PChar 型または PWideChar 型で，他方の型が `array [0..n] of Char` という形式の添字がゼロから始まる文字配列である
- 一方の型が Pointer 型（型なしポインタ型）で，他方の型が任意のポインタ型である
- 両方の型が同じ型を指す型付きポインタであり，**{$T+}** コンパイラ指令が有効である
- 両方の型が手続き型であり，結果の型とパラメータの数が一致し，対応する位置にあるパラメータの間に型の同一性がある

代入の互換性

代入互換性は対称的な関係ではありません。T2 型の式を T1 型の変数に代入できるのは，式の値が T1 の範囲内にあり，以下の条件の少なくとも 1 つが満たされる場合です。

- T1 と T2 が同一の型で，どちらもファイル型でなく，いずれかのレベルにファイル型を含む構造化型でもない
- T1 と T2 が互換性のある順序型である
- T1 と T2 がどちらも実数型である
- T1 が実数型で，T2 が整数型である
- T1 が PChar 型または任意の文字列型で，式が文字列定数である
- T1 と T2 が両方とも文字列型である
- T1 が文字列型で，T2 が Char 型またはパック文字列型である
- T1 が長い文字列型で，T2 が PChar 型である
- T1 と T2 が互換性のあるパック文字列型である
- T1 と T2 が互換性のある集合型である
- T1 と T2 が互換性のあるポインタ型である
- T1 と T2 が両方ともクラス型，クラス参照型，またはインターフェース型で，T2 が T1 から派生した型である
- T1 がインターフェース型で，T2 が T1 を実装するインターフェース型である
- T1 が PChar 型または PWideChar 型で，T2 が `array [0..n] of Char` という形式の添字がゼロから始まる文字配列である
- T1 と T2 が互換性のある手続き型である。一部の代入文では，関数または手続きの識別子は手続き型の式として扱われる。5-29 ページの「文と式での手続き型」を参照
- T1 がバリアント型で，T2 が整数型，実数型，文字列型，文字型，論理型，またはインターフェース型である
- T1 が整数型，実数型，文字列型，文字型，または論理型で，T2 がバリアント型である
- T1 が IUnknown または IDispatch インターフェース型であり，T2 がバリアント型である。T1 が IUnknown の場合，バリアントの型コードは varEmpty，varUnknown，または varDispatch でなければならない。T1 が IDispatch の場合は varEmpty または varDispatch でなければならない

型の宣言

型宣言では，型を表す識別子を指定します。型宣言の構文は次のとおりです。

```
type 新規型名 = 型
```

新規型名には有効な識別子を使用します。たとえば，次の宣言があるとします。

```
type TMyString = string;
```

この場合は，次の変数宣言が可能です。

```
var S: TMyString;
```

ポインタ型を例外として，型識別子のスコープに型宣言自体は含まれません。したがって，たとえばレコード型にその型自体が再帰的に含まれるような宣言はできません。

既存の型と同一の型を宣言した場合，コンパイラは新しい型識別子を元の型のためのエリアスとして扱います。たとえば，次のように宣言されているとします。

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

X と Y は同じ型になります。実行時に TValue と Real を区別する方法はありません。通常，このことは問題になりません。しかし，たとえばのプロパティエディタを特定の型のプロパティに関連付ける場合など，実行時型情報を利用するために新しい型を定義する場合は，名前が異なることと型が異なることの違いが重要になります。このような場合は次の構文を使用します。

```
type 新規型名 = type 型
```

次に例を示します。

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

これは Real とは異なる TValue という新しい型をコンパイラに強制的に作成させます。

変数

変数は，実行時に値を変更できる識別子です。言い方を変えると，変数はメモリ内の位置に付ける名前であり，この名前を使用してその位置に対する読み取りと書き込みを行うことができます。変数はデータの容器のようなものであり，格納されているデータの解釈方法をコンパイラが認識できるように型が付けられています。

変数の宣言

変数宣言の基本的な構文は次のとおりです。

```
var 識別子リスト : 変数型 ;
```

識別子リストには有効な識別子のカンマ区切りのリストを記述し，変数型には任意の有効な型を使用します。次に例を示します。

```
var I: Integer;
```

これは Integer 型の変数 I を宣言します。

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

これは Real 型の変数 X と Y を宣言します。

変数を連続して宣言する場合，予約語 **var** を繰り返す必要はありません。

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

手続きまたは関数の内部で宣言された変数はローカル変数，その他の変数はグローバル変数と呼ばれることがあります。次の構文を使用すると，グローバル変数を宣言と同時に初期化できます。

```
var 識別子 : 変数型 = 初期値 ;
```

初期値には，変数型の任意の値を表す定数式を指定します。定数式についての詳細は，5-40 ページの「定数式」を参照してください。次の宣言があるとします。

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

これは次の宣言および文と一致します。

```
var I: Integer;
 ⋮
I := 7;
```

`var X, Y, Z: Real;` のように複数の変数を宣言する場合，あるいはバリアント型またはファイル型の変数を宣言する場合は，初期値を指定できません。

明示的に初期化されていないグローバル変数はコンパイラによって 0 に初期化されます。一方，ローカル変数は宣言で初期化することができず，値が代入されるまではランダムなデータが含まれています。

変数を宣言するとメモリが割り当てられます。このメモリは，変数が使用されなくなったときに自動的に解放されます。特にローカル変数は，その変数が宣言されている関数または手続きからプログラムが抜け出るまでの間しか存在しません。変数とメモリ管理についての詳細は，第 11 章「メモリ管理」を参照してください。

絶対アドレス

ほかの変数と同じアドレスを持つ新しい変数を作成することができます。そのためには，新しい変数の宣言で型名に続けて **absolute** 指令を記述し，その後ろに既存の（以前に宣言した）変数の名前を指定します。次に例を示します。

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

これは変数 StrLen が Str と同じアドレスから始まることを指定します。短い文字列の第 1 バイトには文字列の長さが格納されるため，Str の長さが StrLen の値になります。

absolute 宣言で変数を初期化したり，または **absolute** をほかの指令と組み合わせたりすることはできません。

動的変数

GetMem 手続きまたは New 手続きを呼び出すと動的変数を作成できます。動的変数はヒープに割り当てられ，自動的な管理は行われません。作成した動的変数のメモリは，プログラマの責任で最終的に解放する必要があります。GetMem で作成した変数の破棄には FreeMem を使用し，New で作成した変数の破棄には Dispose を使用します。動的変数を処理するその他の標準ルーチンには，ReallocMem，Initialize，StrAlloc，および StrDispose があります。

長い文字列，ワイド文字列，動的配列，バリアント，およびインターフェースもヒープに割り当てられる動的変数ですが，これらの変数のメモリは自動的に管理されます。

スレッドローカル変数

スレッドローカル変数（スレッド変数）はマルチスレッドのアプリケーションで使用されます。スレッドローカル変数はグローバル変数に似ていますが，それぞれのスレッドが変数のプライベートなコピーを持ち，ほかのスレッドからはアクセスできない点が異なります。スレッドローカル変数は **var** ではなく **threadvar** で宣言します。次に例を示します。

```
threadvar X: Integer;
```

スレッド変数の宣言には，以下の規則が適用されます。

- 手続きまたは関数の内部には記述できない
- 初期化を含むことができない
- **absolute** 指令を指定できない

ポインタ型や手続き型のスレッド変数を作成してはなりません。また，動的にロードされる可能性のある共有オブジェクト（パッケージ以外）内でスレッド変数を使用してはなりません。

通常はコンパイラが管理する動的変数（長い文字列，ワイド文字列，動的配列，バリアント，およびインターフェースなど）は **threadvar** によって宣言できます。しかし，実行の各スレッドによってヒープに割り当てられたメモリが自動的に解放されることはありません。これらのデータ型をスレッド変数で使用する場合は，明示的にメモリを解放してください。次に例を示します。

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  ...
S := '';  // S が使用しているメモリを解放する
```

（バリアントを解放するには Unassigned に設定し，インターフェースや動的配列を解放するには **nil** に設定する）

宣言された定数

「定数」と呼ばれる構文はいくつかあります。17 のような数値定数もあり，'Hello world!' のような文字列定数もあります。数値定数は数値とも呼ばれ，文字列定数は文字列または文字列リテラルとも呼ばれます。数値定数と文字列定数についての詳細は，第 4 章「構文の要素」を参照してください。列挙型はすべて，その型の値を表す定数を定義します。True，False，**nil** などの定義済みの定数もあります。さらに，変数と同じように宣言によって個別に作成される定数もあります。

宣言された定数は，真の定数であるか，または型付き定数です。この 2 つの定数の種類は一見似ていますが，異なる規則に従っており，異なる用途のために使用されます。

真の定数

真の定数は宣言された識別子で，その値は変化しません。次に例を示します。

```
const MaxValue = 237;
```

これは整数 237 を返す MaxValue という定数を宣言します。真の定数を宣言する構文は次のとおりです。

```
const 定数識別子 = 定数式
```

定数識別子には任意の有効な識別子を指定します。定数式には，コンパイラがプログラムを実行せずに評価できる式を使用します。詳細については，5-40 ページの「定数式」を参照してください。

定数式が順序値を返す場合は，値を型キャストすることで宣言された定数の型を指定することができます。次に例を示します。

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

これは整数 17 を返す Int64 型の定数 MyNumber を宣言します。それ以外の場合は，定数式の型が宣言された定数の型になります。

- 定数式が文字列の場合，宣言された定数は任意の文字列型と互換性があります。文字列の長さが 1 である場合は，任意の文字型とも互換性があります。
- 定数式が実数の場合，その型は Extended です。定数式が整数の場合，その型は次の表に示すように決定されます。

表 5.8　整数定数の型

定数の範囲（16 進）	定数の範囲（10 進）	型
-$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	順序型
$100000000..$7FFFFFFFFFFFFFFF	4294967296..2^{63}-1	Int64

次に定数宣言の例を示します。

```
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;
```

定数式

定数式は，それが含まれるプログラムをコンパイラが実行せずに評価できる式です。定数式には，数値，文字列，真の定数，列挙型の値，および特殊値の True，False，**nil** があります。また，演算子，型キャスト，および集合構成子とこれらの要素のみを使用して作成した式も定数式です。以下の定義済みの関数を除いて，変数，ポインタ，または関数呼び出しを定数式に含むことはできません。

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

このような定数式の定義は，Object Pascal の構文仕様のいくつかの箇所で使用されています。定数式は，グローバル変数の初期化，部分範囲型の定義，列挙型の値への順序値の割り当て，デフォルトのパラメータ値の指定，**case** 文の記述，および真の定数と型付き定数の宣言に必要です。

定数式の例を次に示します。

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

リソース文字列

リソース文字列はリソースとして格納され，実行形式ファイルまたはライブラリにリンクされるので，プログラムを再コンパイルせずに変更することができます。詳細については，オンラインヘルプでアプリケーションのローカライズに関するトピックを参照してください。

リソース文字列はほかの真の定数と同じように宣言されますが，**const** ではなく **resourcestring** を使用する点が異なります。= シンボルの右辺の式は定数式でなければならず，文字列値を返さなければなりません。次に例を示します。

```
resourcestring
  CreateError = 'Cannot create file %s';      { 形式指定子についての詳細は }
  OpenError = 'Cannot open file %s';          { オンラインヘルプで「形式文字列」を参照 }
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks\000\000';
  SomeResourceString = SomeTrueConstant;
```

ライブラリ間でリソース文字列の名前の衝突がある場合は，コンパイラによって自動的に解決されます。

型付き定数

真の定数とは異なり，型付き定数には配列型，レコード型，手続き型，およびポインタ型の値を格納できます。型付き定数は定数式で使用できません。

コンパイラがデフォルトの **{$J-}** 状態の場合，型付き定数には新しい値を代入できません。つまり，読み取り専用の変数のように機能します。しかし，**{$J+}** コンパイラ指令が有効な場合は，型付き定数に新しい値を代入できます。つまり，型付き定数は初期化済みの変数のように機能します。

型付き定数は次のように宣言します。

```
const 定数識別子 : 定数型 = 定数値
```

定数識別子には任意の有効な識別子を使用します。定数型にはファイルとバリアントを除く任意の型を，定数値には定数型の式を使用します。次に例を示します。

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

ほとんどの場合には，定数値は定数式でなければなりません。ただし，定数型が配列型，レコード型，手続き型，またはポインタ型である場合は特殊な規則が適用されます。

配列型定数

配列型定数を宣言するには，配列の各要素の値をカンマ区切りで指定したリストをカッコで囲み，宣言の末尾に記述します。各要素の値は定数式で表さなければなりません。次に例を示します。

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

これは文字の配列を格納する Digits という型付き定数を宣言します。

ゼロで始まる文字配列はヌルで終わる文字列を表すことが多いので，文字列定数を使用して文字配列を初期化することができます。したがって，この宣言は次のように記述することもできます。

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

多次元配列定数を定義するには，各次元の値をカッコで囲み，カッコをカンマで区切ります。次に例を示します。

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

これは Maze という配列を作成します。Maze の内容は次のとおりです。

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

配列型定数は，いずれのレベルにもファイル型の値を含むことができません。

レコード型定数

レコード型定数を宣言するには，各フィールドの値を「フィールド名： フィールド値」の形式で指定し，各フィールドをセミコロンで区切り，全体をカッコで囲んで宣言の末尾に記述します。各値は定数式で表さなければなりません。フィールドはレコード型の宣言と同じ順序で指定しなければならず，タグフィールドが存在する場合は値を指定する必要があります。可変部分があるレコード型の場合は，タグフィールドで選択された可変部にのみ値を指定できます。

例を以下に示します。

```
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);
```

レコード型定数は，いずれのレベルにもファイル型の値を含むことができません。

手続き型定数

手続き型定数を宣言するには，宣言されている定数の型と互換性のある関数または手続きの名前を指定します。次に例を示します。

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

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

このように宣言されている場合は，手続き型定数 MyFunction を関数呼び出しで使用できます。

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

手続き型定数には値 **nil** も代入できます。

ポインタ型定数

ポインタ型定数を宣言するときは，コンパイル時に解決可能な値に初期化しなければなりません。少なくとも，相対アドレスとして解決可能な値に初期化する必要があります。これには，**@** 演算子を使用する方法と **nil** を使用する方法があり，定数が PChar 型の場合は文字列リテラルを使用する方法もあります。たとえば，I が Integer 型のグローバル変数である場合は次のような定数を宣言できます。

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

グローバル変数はコードセグメントに格納されるので，コンパイラはこの宣言を解決できます。関数とグローバル定数も同じです。

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

文字列リテラルはグローバル定数としてメモリを割り当てられるので，文字列リテラルを使用してPChar 型定数を初期化することができます。

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

スタックに割り当てられるローカル変数のアドレスと，ヒープに割り当てられる動的変数のアドレスは，ポインタ変数に割り当てることができません。

第6章

手続きと関数

手続きと関数は，それ自体で完結した文の集合で，プログラム内のほかの場所から呼び出すことができます。手続きと関数は，ルーチンと総称されます。関数は，実行すると値を返すルーチンです。一方，手続きは値を返さないルーチンです。

関数を呼び出すと値が返されるため，関数呼び出しは代入操作や演算操作の中で式として使用できます。次に例を示します。

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

このコードでは，SomeFunction を呼び出して，その結果を I に代入しています。関数呼び出しを代入文の左辺に置くことはできません。

手続き呼び出し，および **{$X+}** で拡張構文が有効になっている場合の関数呼び出しは，どちらも完全な文として使用できます。次に例を示します。

```
DoSomething;
```

このコードでは，DoSomething ルーチンを呼び出しています。DoSomething が関数の場合は，戻り値は破棄されます。

手続きと関数は再帰的に呼び出すことも可能です。

手続きと関数の宣言

手続きまたは関数を宣言するときは，その名前とともに，パラメータの個数および型を指定し，関数の場合は戻り値の型も指定します。宣言の中でこの部分は，プロトタイプ，ヘッディング，あるいはヘッダーと呼ばれることがあります。それに続いて，手続きまたは関数が呼び出されたときに実行するコードを記述します。この部分は，ルーチンの本体またはブロックと呼ばれることがあります。

手続きまたは関数の本体には標準手続き Exit を置くことができます。Exit は，それが置かれたルーチンの実行を停止させ，そのルーチンが呼び出された場所にただちにプログラムの制御を戻します。

手続きの宣言

手続きの宣言は，次のような形式をとります。

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

procedureName には有効な識別子を指定し，statements にはこの手続きが呼び出されたときに実行する一連の文を記述します。(parameterList)，directives;，および localDeclarations; は必要に応じて指定します。

- parameterList についての詳細は，6-11 ページの「パラメータ」を参照してください。
- directives（指令）についての詳細は，6-4 ページの「呼び出し規約」，6-5 ページの「forward 宣言とインターフェース宣言」，6-6 ページの「external 宣言」，6-8 ページの「手続きと関数のオーバーロード」，および 9-3 ページの「動的にロード可能なライブラリの記述」を参照してください。指令を複数指定する場合はセミコロンで区切ります。
- ローカル識別子を宣言する localDeclarations についての詳細は，6-10 ページの「ローカル宣言」を参照してください。

手続き宣言の例を次に示します。

```
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;
```

このように宣言されている場合，NumString 手続きは次のようにして呼び出すことができます。

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

この手続き呼び出しでは，値「17」を文字列型変数の MyString に代入しています。

手続きの文ブロックの中では，localDeclarations の部分で宣言した変数その他の識別子を使用できます。また，パラメータリストで指定したパラメータ名（上の例では N と S）を使用することもできます。なお，パラメータリストではローカル変数を定義するので，パラメータリストで指定したパラメータ名を localDeclarations の部分で再定義しないでください。さらに，手続きの宣言部分が属しているスコープ内にある識別子も使用できます。

関数宣言

関数宣言は手続き宣言に似ていますが，関数宣言では戻り値の型と戻り値を指定するという点が異なります。関数の宣言は，次のような形式をとります。

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

functionName には有効な識別子を指定し，returnType には型を指定します。statements にはこの関数が呼び出されたときに実行する一連の文を記述します。(parameterList)，directives;，および localDeclarations; は必要に応じて指定します。

- parameterList についての詳細は，6-11 ページの「パラメータ」を参照してください。
- directives（指令）についての詳細は，6-4 ページの「呼び出し規約」，6-5 ページの「forward 宣言とインターフェース宣言」，6-6 ページの「external 宣言」，6-8 ページの「手続きと関数のオーバーロード」，および 9-3 ページの「動的にロード可能なライブラリの記述」を参照してください。指令を複数指定する場合はセミコロンで区切ります。
- ローカル識別子を宣言する localDeclarations についての詳細は，6-10 ページの「ローカル宣言」を参照してください。

関数の文ブロックに適用される規則は手続きと同じです。関数の文ブロックの中では，関数の localDeclarations の部分で宣言した変数その他の識別子，パラメータリストで指定したパラメータ名，および関数の宣言部分が属しているスコープ内にある識別子を使用できます。また，定義済み変数 Result と同じように，関数名自体を戻り値を保持した特別な変数として使用できます。

次に例を示します。

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

このコードは，パラメータをとらず，常に整数値 17 を返す WF という定数型の関数を定義しています。このコードと次のコードは同等です。

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

もう少し複雑な関数宣言の例を次に示します。

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

```
    Max := X;
  end;
```

Result または関数名には，文ブロック内で値を何度代入してもかまいません。ただし，代入する値は宣言した戻り値の型と同じ型の値でなければなりません。関数の実行終了時点で Result または関数名に最後に代入された値がその関数の戻り値になります。次に例を示します。

```
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 と関数名は常に同じ値を表します。したがって，

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

この関数は値 11 を返します。しかし，Result と関数名は完全に交換可能なわけではありません。関数名を代入文の左辺に置くと，コンパイラは関数名が戻り値の管理に使用されているとみなし（Result と同様），関数名を文ブロックの中で代入文の左辺以外の部分に置くと，コンパイラはそれを関数の再帰的呼び出しであると解釈します。一方，Result は，演算操作，型キャスト，集合構成子，添字，およびほかのルーチンの呼び出しの中で変数として使用できます。

{$X+} で拡張構文が有効になっていれば，Result はすべての関数で暗黙に宣言されます。Result を再宣言するのは避けてください。

Result または関数名に代入が行われずに関数の実行が終了した場合，関数の戻り値は未定義になります。

呼び出し規約

手続きまたは関数の宣言時には，**register**，**pascal**，**cdecl**，**stdcall**，**safecall** のいずれかの指令を使用して呼び出し規約を指定できます。次に例を示します。

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

呼び出し規約とは，ルーチンにパラメータが渡される順序を決めるものです。また，スタックからパラメータを削除する方法や，パラメータを渡すために使用するレジスタの種類，エラーと例外の処理方法にも関係します。デフォルトの呼び出し規約は **register** です。

- **register** 規約と **pascal** 規約では，パラメータが左から右へ渡されます。つまり，左端のパラメータが最初に評価されて渡され，右端のパラメータが最後に評価されて渡されます。**cdecl** 規約，**stdcall** 規約，および **safecall** 規約では，パラメータが右から左へ渡されます。
- **cdecl** を除くすべての規約では，手続きまたは関数が戻りと同時にスタックからパラメータを削除します。**cdecl** 規約では，呼び出し側が呼び出しの戻り時にスタックからパラメータを削除します。
- **register** 規約では 3 つまでの CPU レジスタを使ってパラメータが渡されますが，その他の規約ではすべてのパラメータがスタックを使用して渡されます。
- **safecall** 規約では，例外 "firewall" が実装されます。Windows では，COM のエラー処理と例外処理が実装されます。

次の表に呼び出し規約の要約を示します。

表 6.1　呼び出し規約

指令	パラメータが渡される順序	パラメータの削除	パラメータの受け渡しにレジスタを使用
register	左から右	ルーチン	あり
pascal	左から右	ルーチン	なし
cdecl	右から左	呼び出し側	なし
stdcall	右から左	ルーチン	なし
safecall	右から左	ルーチン	なし

デフォルトで使用される **register** 規約では，通常，スタックフレームの作成が回避されるのでもっとも効率的です（パブリッシュプロパティのアクセスメソッドには **register** を使わなければなりません）。**cdecl** 規約は，C または C++ で記述された共有ライブラリから関数を呼び出す場合に便利です。一方，外部コードを呼び出す場合は，**stdcall** および **safecall** 規約が推奨されます。Windows では，オペレーティングシステム API は **stdcall** および **safecall** です。他のオペレーティングシステムでは一般的に **cdecl** を使用します（**cdecl** より **stdcall** の方が効率は高いことに注意してください）。

デュアルインターフェースのメソッドの宣言には，**safecall** 規約を使用しなければなりません。**pascal** 規約は，下位互換性を保つ目的で残されています。呼び出し規約についての詳細は，第 12 章「プログラムの制御」を参照してください。

near，**far**，**export** の 3 つの指令は 16 ビット Windows プログラミングにおける呼び出し規約であり，32 ビットアプリケーションに対しては無効です。これらの規約は，下位互換性を保つ目的でのみ残されています。

forward 宣言とインターフェース宣言

ローカル変数宣言や文が含まれるブロックのかわりに **forward** 指令を指定する手続き宣言または関数宣言が **forward** 宣言です。次に例を示します。

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

このコードでは，Calculate という関数を定義しています。このルーチンは，この **forward** 宣言以降のどこかでブロックが含まれる定義宣言を記述し，そこで再宣言しなければなりません。Calculate の定義宣言は，たとえば次のようになります。

```
function Calculate;
 ⋮  { 宣言部 }
begin
 ⋮  { 文ブロック }
end;
```

通常は，定義宣言の中でルーチンのパラメータリストや戻り値の型を再度指定する必要はありませんが，指定する場合は **forward** 宣言での指定と完全に同じでなければなりません。ただし，デフォルトパラメータは省略可能です。**forward** 宣言でオーバーロード手続きまたはオーバーロード関数（6-8 ページの「手続きと関数のオーバーロード」を参照）を指定した場合は，定義宣言の中でもパラメータリストを指定しなければなりません。

forward 宣言からその定義宣言までの間には，ほかの宣言以外には何も記述してはいけません。定義宣言を **external** 宣言か **assembler** 宣言にすることはできますが，定義宣言をさらに **forward** 宣言にすることはできません。

forward 宣言の目的は，手続きまたは関数の識別子のスコープをソースコードの前の方まで広げることです。**forward** 宣言を使用することで，ルーチンをその実際の定義位置よりも前から呼び出すことができるようになります。このように，**forward** 宣言を使用するとコードをより柔軟性のある構成にすることができますが，相互再帰を使用する場合にも **forward** 宣言が必要になることがあります。

forward 指令をユニットのインターフェース部に記述することはできません。しかし，手続きおよび関数のヘッダーをインターフェース部に置くと **forward** 宣言と同じような効果が得られます。手続きおよび関数のヘッダーをインターフェース部に置いた場合は，定義宣言を実現部に記述しなければなりません。インターフェース部で宣言したルーチンは，同じユニット内のほかの場所や，ほかのユニット，そのルーチンが宣言されたユニットを使用するプログラムから呼び出すことができます。

external 宣言

手続きまたは関数の宣言でブロックのかわりに **external** 指令を指定するのが **external** 宣言です。**external** 宣言を行うと，分割コンパイルしたルーチンを呼び出すことができます。外部ルーチンは，オブジェクトファイルまたは動的にロード可能なライブラリから呼び出すことができます。

可変個のパラメータを取る C++ 関数をインポートする場合は，**varargs** 指令を使用します。次に例を示します。

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

varargs 指令は，外部ルーチンおよび **cdecl** 呼び出し規約に対してのみ機能します。

オブジェクトファイルのリンク

分割コンパイルされたオブジェクトファイルに含まれるルーチンを呼び出すには，まず，コンパイラ指令 **$L**（または **$LINK**）を使用してオブジェクトファイルをアプリケーションにリンクします。次に例を示します。

```
Windows の場合 : {$L BLOCK.OBJ}
```

```
Linux の場合：  {$L block.o}
```

このコードを記述すると，これが記述されたプログラムまたはユニットに BLOCK.OBJ（Windows）または block.o（Linux）がリンクされます。次に，呼び出したい関数と手続きを，次のようにして宣言します。

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

これで，BLOCK.OBJ（Windows）または block.o（Linux）に含まれる MoveWord ルーチンと FillWord ルーチンを呼び出すことができるようになります。

上記のような宣言は，アセンブリ言語で記述された外部ルーチンを使用するためによく用いられます。アセンブリ言語ルーチンを Object Pascal のソースコードに直接含めることも可能です。詳細については 第 13 章「インラインアセンブラコード」を参照してください。

ライブラリからの関数のインポート

動的にロード可能なライブラリ（.so または .DLL）からルーチンをインポートするには，次の形式の指令を通常の手続きまたは関数のヘッダーの最後に付加します。

```
external stringConstant;
```

stringConstant には，.DLL ファイルの名前を単引用符で囲んで指定します。次に例を示します。

Windows の場合：

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

このコードでは，strlib.dll に含まれる SomeFunction という関数をインポートしています。

Linux の場合：

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

このコードでは，strlib.so に含まれる SomeFunction という関数をインポートしています。

ルーチンを，DLL 内での名前とは別の名前でインポートすることもできます。これには，次のように **external** 指令の中で元の名前を指定します。

```
external stringConstant1 name stringConstant2;
```

最初の stringConstant には .DLL ファイルの名前を指定し，2 番目の stringConstant にはルーチンの元の名前を指定します。

Windows の場合：たとえば，次の宣言は user32.dll（Windows API の一部）から関数をインポートします。

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

この関数の元の名前は MessageBoxA ですが，これを MessageBox としてインポートしています。

次のように，名前のかわりに，インポートしたいルーチンを番号で指定することもできます。

```
external stringConstant index integerConstant;
```

integerConstant にはエクスポートテーブルに記載されているルーチンのインデックスを指定します。

Linux の場合：たとえば，次の宣言では libc.so.6 から標準システム関数がインポートされます。

```
function OpenFile(const PathName: PChar; Flags: Integer): Integer; cdecl;
  external 'libc.so.6' name 'open';
```

この関数の元の名前は open ですが，これを OpenFile としてインポートしています。

宣言のインポート時には，ルーチン名のスペルと大文字小文字の区別を間違えないようにしてください。インポートしたルーチンを後で呼び出すとき，名前の大文字と小文字が区別されます。

ライブラリについての詳細は，第 9 章「ライブラリとパッケージ」を参照してください。

手続きと関数のオーバーロード

同一のスコープ内で 2 つ以上のルーチンを同じ名前で定義できます。これを「オーバーロードする」といいます。オーバーロードするルーチンは **overload** 指令を使って宣言し，それぞれ異なるパラメータリストを指定しなければなりません。たとえば，次のように宣言されているとします。

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

この 2 つの宣言では，同じ Divide という名前で，とるパラメータの型が違う 2 つの関数を作成しています。Divide が呼び出されると，コンパイラは渡される実パラメータを見て，どちらの関数を呼び出すかを判断します。たとえば，`Divide(6.0, 3.0)` は，実数の引数が指定されているため，最初の Divide 関数が呼び出されます。

オーバーロードルーチンに，そのルーチンのどの宣言で指定された型とも同一でない型のパラメータを渡すことはできますが，1 つ以上の宣言におけるパラメータと代入互換でなければなりません。このような状況が最も多く生じるのは，ルーチンを異なる整数型あるいは異なる実数型でオーバーロードしている場合です。例を示します。

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

このような場合，曖昧さを排除することが可能であれば，コンパイラは呼び出しにおける実パラメータに適合する最も小さな範囲の型をパラメータとするルーチンを呼び出します（具体的な値が指定された定数式は，常に Extended 型になることを覚えておいてください）。

オーバーロードされるルーチンが区別できるように，パラメータの個数または型が異なっていなければなりません。したがって，次のような宣言を行うと，コンパイルエラーになります。

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

しかし，次の宣言は有効です。

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

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

オーバーロードルーチンを **forward** 宣言またはインターフェース部で宣言した場合は，その定義宣言の中でも同じパラメータリストを指定しなければなりません。

コンパイラは，同じパラメータ位置に AnsiString/PChar パラメータを持つオーバーロード関数と WideString/WideChar パラメータを持つオーバーロード関数を区別できます。そのようなオーバーロードの状況で渡された文字列定数または文字列リテラルは，ネイティブな文字列または文字型である AnsiString/PChar に変換されます。

```
procedure test(const S: String);  overload;
procedure test(const W: WideString); overload;

var
  a: string;
  b: widestring;
begin
  a := 'a';
  b := 'b';
  test(a);    // String バージョンが呼び出される
  test(b);    // WideString バージョンが呼び出される
  test('abc');    // String バージョンが呼び出される
  test(WideString('abc'));   // WideString バージョンが呼び出される
end;
```

オーバーロード関数宣言のパラメータとしてバリアントも使用できます。バリアントは，単純型のより汎用的な型と考えられます。望ましいのは，バリアント一致よりも，常に正確に型が一致することです。このようなオーバーロード状況でバリアントが渡され，そのパラメータ位置にバリアントを取るオーバーロード関数である場合，そのパラメータはバリアント型と正確に一致したと見なされます。

これによって，浮動小数点型についてはいくつか副作用が生じることがあります。浮動小数点型はサイズによって一致するかどうかが決まります。オーバーロード呼び出しで渡された浮動小数点変数に正確に一致する定義がなく，バリアントパラメータが使用可能な場合，サイズの小さい浮動小数点型の代わりにそのバリアントが適用されます。

例を示します。

```
procedure foo(i: integer); overload;
procedure foo(d: double);  overload;
procedure foo(v: variant); overload;

var
  v: variant;
begin
  foo(1);       // integer バージョン
  foo(v);       // variant バージョン
  foo(1.2);     // variant バージョン（浮動小数点リテラル → extended 精度）
end;
```

この例では，foo の Double バージョンではなく，Variant バージョンが呼び出されます。これは，定数 1.2 は暗黙的に Extended 型となり，Double に正確には一致しないためです。Extended も Variant に正確には一致しませんが，Variant がより汎用的な型である（Double は Extended よりサイズが小さい）と見なされます。

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

この型キャストは機能しません。このような場合には，次のように型付き定数を使う必要があります。

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

上のコードは正しく動作し，Double バージョンが呼び出されます。

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

このコードも foo の Double バージョンを呼び出します。Single は，Variant と比べて Double により一致すると見なされます。

一連のオーバーロードルーチンを宣言するときに，バリアントへの型の拡張を回避する最良の方法は，Variant バージョンに加えて，各浮動小数点型（Single，Double，Extended）ごとにオーバーロード関数のバージョンを宣言することです。

オーバーロードルーチンでデフォルトパラメータを使用する場合は，パラメータ宣言があいまいにならないよう注意してください。詳細については，6-18 ページの「デフォルトパラメータとオーバーロードルーチン」を参照してください。

呼び出し時に限定子でルーチン名を限定すれば，オーバーロードの効果を制限できます。たとえば，`Unit1.MyProcedure(X, Y)` と指定すると，Unit1 で宣言されたルーチンだけを呼び出すことができます。指定した名前とパラメータリストに一致するルーチンが Unit1 の中にない場合は，エラーになります。

クラス階層の中にオーバーロードメソッドを分散して配置する方法についての詳細は，7-12 ページの「メソッドのオーバーロード」を参照してください。オーバーロードルーチンの共有ライブラリからのエクスポートについての詳細は，9-5 ページの「exports 節」を参照してください。

ローカル宣言

関数または手続きの本体の先頭部分では，その文ブロックで使用するローカル変数を宣言するのが通例です。また，定数，型，その他のルーチンを宣言することもできます。ローカル識別子のスコープは，それが宣言されているルーチン内に限定されます。

ルーチンのネスト

関数および手続きのブロックのローカル宣言部に，ほかの関数や手続きが記述されることがあります。たとえば，次のコードでは DoSomething という手続きを宣言していますが，内部に別の手続きがネストして記述されています。

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

```
begin
  ⋮
  NestedProc(S);
  ⋮
end;
```

ネストされたルーチンのスコープは，それが宣言されている手続きまたは関数内に限定されます。上の例では，NestedProc は DoSomething 内からのみ呼び出すことができます。

ネストされたルーチンの実例は，SysUtils ユニットに含まれる DateTimeToString 手続き，ScanDate 関数などのルーチンを見てください。

パラメータ

ほとんどの手続きおよび関数のヘッダーには，パラメータリストがあります。たとえば，次のヘッダーでは (X: Real; Y: Integer) の部分がパラメータリストになります。

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

パラメータリストとは，パラメータ宣言をセミコロンで区切って並べ，全体をカッコで囲んだものです。それぞれの宣言は，パラメータ名をカンマで区切って並べ，通常，その後にコロンと型識別子が付加された形になっています。また，= によってデフォルト値が指定されることもあります。パラメータ名は有効な識別子でなければなりません。宣言の前には，**var**，**const**，**out** のいずれかの予約語を付加することができます。例を以下に示します。

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

パラメータリストでは，そのルーチンが呼び出されたときにルーチンに渡すパラメータの個数，順序，および型を指定します。パラメータをとらないルーチンの場合は，次のように識別子のリストとカッコを記述しません。

```
procedure UpdateRecords;
begin
 ⋮
end;
```

パラメータ名（上の例では X と Y）は，手続きまたは関数の本体内でローカル変数として使うことができます。パラメータ名を手続きまたは関数の本体のローカル宣言部で再宣言しないでください。

パラメータのセマンティクス（意味論）

パラメータは，以下のようにさまざまな分類ができます。

- すべてのパラメータは，値パラメータ，変数パラメータ，定数パラメータ，out パラメータのいずれかに分類されます。値パラメータがデフォルトです。予約語 **var**，**const**，**out** は，それぞれ変数パラメータ，定数パラメータ，出力パラメータを表します。
- 値パラメータには必ず型を指定します。これに対して，定数パラメータ，変数パラメータ，出力パラメータは，型を指定しても指定しなくてもかまいません。

- 配列パラメータには特別な規則が適用されます。6-15 ページの「配列パラメータ」を参照してください。

ファイルと，ファイルが含まれる構造化型のインスタンスは変数パラメータ（**var**）としてしか渡せません。

値パラメータと変数パラメータ

大部分のパラメータは，値パラメータ（デフォルト）か変数パラメータ（**var**）のどちらかになります。値パラメータは値で渡され，変数パラメータは参照で渡されます。以下の 2 つの関数を例にとって，これを具体的に見てみます。

```
function DoubleByValue(X: Integer): Integer;     // X は値パラメータ
begin
  X := X * 2;
  Result := X;
end;
function DoubleByRef(var X: Integer): Integer;     // X は変数パラメータ
begin
  X := X * 2;
  Result := X;
end;
```

どちらの関数も返す結果は同じですが，渡される変数の値を変更できるのは 2 番目の DoubleByRef 関数だけです。これらの関数を次のように呼び出してみます。

```
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;
```

DoubleByValue に渡した変数 I の値は，このコードの実行後も代入時点と変わりません。しかし，DoubleByRef に渡した変数 V は，代入時とは異なる値になっています。

値パラメータは，手続きまたは関数の呼び出し時に渡す値に初期化されるローカル変数のような働きをします。変数を値パラメータとして渡すと，手続きまたは関数はそのコピーを作成します。したがって，コピーに変更を加えても元の変数の値は変わらず，プログラムの実行が呼び出し側に戻った時点でコピーの内容は失われてしまいます。

一方，変数パラメータはコピーではなくポインタのような働きをします。関数または手続きの本体部分でパラメータに加えた変更は，プログラムの実行が呼び出し側に戻り，パラメータ名自体がスコープ外になっても失われません。

同一の変数が複数の変数パラメータに渡された場合であっても，コピーは作成されません。これを次の例を使って説明します。

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

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

このコードの実行後，I の値は 3 になります。

宣言の中で変数パラメータが指定されたルーチンを呼び出すときは，変数，型付き定数（**{$J+}** が指定されている場合），逆参照ポインタ，フィールド，添字付きの変数など，代入可能な式を渡さなければなりません。前述の例でいくと，`DoubleByRef(7)` はエラーになりますが，`DoubleByValue(7)` は正しく処理されます。

添字またはポインタの逆参照は，`DoubleByRef(MyArray[I])` のように変数パラメータ（**var**）で渡された場合，ルーチンの実行前に一度評価されます。

定数パラメータ

定数パラメータ（**const**）は，ローカル定数や読み出し専用変数のようなものです。定数パラメータは値パラメータに似ていますが，手続きまたは関数の本体で定数パラメータに値を代入したり，定数パラメータを別のルーチンに変数パラメータ（**var**）として渡したりすることはできません。ただし，オブジェクトの参照を定数パラメータとして渡すときは，そのオブジェクトのプロパティに変更を加えることは可能です。

構造化型パラメータや文字列型パラメータに対して **const** を指定すると，コンパイラはコードを最適化できます。また，定数パラメータを使用することで，パラメータが別のルーチンに誤って参照渡しされるのを回避できます。

例として，SysUtils ユニットに含まれる CompareStr 関数のヘッダーを次に示します。

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

S1 と S2 の値は CompareStr の本体で変更されないので，どちらも定数パラメータとして宣言できます。

out パラメータ

out パラメータは，変数パラメータと同じように参照により渡されます。しかし，**out** パラメータの値は，呼び出し先ルーチンに渡る前にいったん破棄されます。つまり，呼び出し先に **out** パラメータを使って値を渡すことはできません。**out** パラメータは結果を受け取ることしかできません。呼び出し先ルーチンの処理結果を受け取るための領域を確保するために使われます。

次のヘッダーを見てください。

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

GetInfo 手続きを呼び出すときには，SomeRecordType 型の変数を渡す必要があります。

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

この MyRecord は GetInfo 手続きにデータを渡すことはできません。MyRecord は，GetInfo が生成する情報を格納する入れ物です。GetInfo 手続きを呼び出すと，呼び出し側で MyRecord の内容が解放され，すべてのフィールドが未定義の状態に戻されます。GetInfo 手続きは，メモリの確保を含め，値を返すために必要な作業を自ら行います。

out パラメータは，COM や CORBA などの分散オブジェクトモデルで多用されます。また，初期化されていない変数を関数または手続きに渡す場合にも **out** パラメータを使用することができます。

メモ **out** パラメータの値は必ずしもゼロにクリアされるわけではありません。レジスタに置かれた値は，そのまま呼び出し先に渡ることがあります。しかし，**out** パラメータ経由で渡された値を使った場合は，予期できないことが起こりえるため，使わないようにしてください。

型なしパラメータ

変数パラメータ，定数パラメータ，および **out** パラメータの宣言時には，型の指定を省略できます（値パラメータは必ず型を指定しなければなりません）。次に例を示します。

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

このコードでは，任意の型のパラメータを受け付ける TakeAnything という手続きを宣言しています。このようなルーチンを呼び出すときに，数値や型なしの数値定数を渡すことはできません。

手続きまたは関数の本体内では，型なしパラメータはどのような型とも互換性がありません。型なしパラメータに対して操作を行うときはキャストしなければなりません。通常，コンパイラは型なしパラメータに対する操作が有効かどうかを確認できません。

次の例では，指定したバイト数の任意の 2 つの変数を比較する関数 Equal で型なしパラメータを使用しています。

```
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;
```

次のような宣言が行われているとします。

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

この場合，以下のような呼び出しが可能です。

```
Equal(Vec1, Vec2, SizeOf(TVector))            // Vec1 と Vec2 を比較
Equal(Vec1, Vec2, SizeOf(Integer) * N)        // Vec1 と Vec2 の最初の N 要素を比較
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5)  // Vec1 の最初の 5 要素と最後の 5 要素を比
較
Equal(Vec1[1], P, 4)                          // Vec1[1] と P.X を比較し，
                                                 Vec1[2] と P.Y を比較
```

文字列パラメータ

短い文字列型のパラメータをとるルーチンを宣言するとき，そのパラメータ宣言で長さを指定することはできません。次の例を見てください。

```
procedure Check(S: string[20]);   // 構文エラー
```

この文ではコンパイルエラーが発生します。しかし，

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

この文は有効です。長さの変化する短い文字列型のパラメータをとるルーチンの宣言には，次のように，特別な識別子である OpenString が使用できます。

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

コンパイラ指令 **{$H-}** と **{$P+}** がどちらも有効になっている場合は，パラメータの宣言において **string** は OpenString と同じ意味を持ちます。

短い文字列，OpenString，**$H**，および **$P** は，下位互換性を保つ目的でのみ残されています。長い文字列を使用すれば，以上のような点を考慮する必要がなくなります。

配列パラメータ

配列型のパラメータをとるルーチンを宣言するとき，そのパラメータ宣言で添字の型を指定することはできません。次の例を見てください。

```
procedure Sort(A: array[1..10] of Integer);   // 構文エラー
```

この宣言はコンパイルエラーになります。しかし，次の宣言は有効です。

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

しかし，多くの場合，オープン配列パラメータを使用した方が望ましいといえます。

オープン配列パラメータ

オープン配列パラメータを使うと，サイズの異なる配列を同じ手続きまたは関数に渡すことができます。オープン配列パラメータを使用するルーチンを定義するには，パラメータ宣言で `array[X..Y] of` type という構文のかわりに `array of` type という構文を使用します。次に例を示します。

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

このコードでは，任意のサイズの文字配列をとり，整数を返す Find という関数を宣言しています。

メモ　オープン配列パラメータの構文は動的配列の構文に似ていますが，両者は同じではありません。上記の関数は，動的配列を含め，Char 型の要素を持つ配列をとります。動的配列だけをとるパラメータを宣言するには，次のように型識別子を指定する必要があります。

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

動的配列についての詳細は，5-18 ページの「動的配列」を参照してください。

ルーチンの本体内では，オープン配列パラメータに対して以下の規則が適用されます。

- 添字は必ず 0 から始まります。したがって，1 番目の要素は 0 で，2 番目の要素は 1 になります。標準関数の Low と High では，それぞれ 0 と 長さ - 1 が返されます。SizeOf 関数では，ルーチンに渡される実際の配列のサイズが返されます。
- オープン配列パラメータは，要素単位でしかアクセスできません。オープン配列パラメータ全体に対して一度に代入を行うことはできません。
- ほかの手続きおよび関数には，オープン配列パラメータか型なしの変数パラメータとしてのみ渡すことができます。オープン配列パラメータは SetLength には渡せません。
- 配列のかわりに，オープン配列パラメータの基本型の変数を渡すことができます。この場合，長さ 1 の配列として扱われます。

配列をオープン配列の値パラメータとして渡すと，コンパイラはルーチンのスタックフレームにそのローカルコピーを作成します。このため，大きな配列を渡す場合は，スタックがオーバーフローしないように注意してください。

次の例では，オープン配列パラメータを使って，実数型配列の各要素に 0 を代入する Clear という手続きと，実数型配列の全要素の合計を計算する Sum という関数を定義しています。

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

オープン配列パラメータを使用したルーチンには，オープン配列コンストラクタを渡すことができます。6-19 ページの「オープン配列コンストラクタ」を参照してください。

型可変オープン配列パラメータ

型可変オープン配列パラメータを使用すると，同一の手続きまたは関数にさまざまな型の式が格納された配列を渡すことができます。型可変オープン配列パラメータを使用するルーチンを定義するには，パラメータの型として **array of const** を指定します。次に例を示します。したがって次のコードでは,

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

さまざまな型の要素が格納された配列を操作できる DoSomething という手続きを宣言しています。

array of const は，**array of** TVarRec と同じ意味を持ちます。TVarRec は System ユニットで宣言されており，整数，論理値，文字，実数，文字列，ポインタ，クラス，クラス参照，インターフェース，バリアントの各型の値を格納できる可変部分を持つレコードを表します。TVarRec の VType フィールドは，配列の各要素の型を表すフィールドです。型によっては値ではなくポインタとして渡されま

す。特に，長い文字列は Pointer 型として渡され，したがって **string** 型に型キャストしなければなりません。詳細については，オンラインヘルプの「TVarRec」を参照してください。

次の例では，渡された各要素について文字列表現を作成し，その結果を連結して 1 つの文字列にする関数で型可変オープン配列パラメータを使用しています。この関数の中で使用している文字列処理ルーチンは，SysUtils で定義されているものです。

```
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 High(Args) 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;
```

この関数は，オープン配列コンストラクタを使用して呼び出すことができます（6-19 ページの「オープン配列コンストラクタ」を参照）。次に例を示します。

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

この場合，「test100 T3.14159TForm」という文字列が返されます。

デフォルトパラメータ

手続きまたは関数のヘッダーで，デフォルトのパラメータ値を指定できます。デフォルト値を指定できるのは，定数パラメータと値パラメータだけです。デフォルト値を指定するには，パラメータ宣言の末尾に = に続けてパラメータに代入可能な型の定数式を指定します。

たとえば，次のような宣言があるとします。

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

次の 2 つの手続き呼び出しは，どちらも同じ意味になります。

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

複数のパラメータを一度に宣言する場合には，デフォルト値は指定できません。したがって，次の宣言は有効ですが，

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

次の宣言はエラーになります。

```
function MyFunction(X, Y: Real = 3.5): Real;   // 構文エラー
```

デフォルト値を指定するパラメータは，パラメータリストの末尾に置かなければなりません。つまり，あるパラメータにデフォルト値を指定したら，それ以降のパラメータにはすべてデフォルト値を指定しなければならないということです。したがって，次の宣言は無効です。

```
procedure MyProcedure(I: Integer = 1; S: string);   // 構文エラー
```

手続き型で指定したデフォルト値は，実際のルーチンの中で指定したデフォルト値よりも優先されます。たとえば，次のように宣言されているとします。

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

この場合，次の 2 つの文の結果は，

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

Resizer に渡される値（N, 1.0）になります。

デフォルトパラメータとして指定できるのは，定数式で指定可能な値だけです（5-40 ページの「定数式」を参照）。したがって，動的配列，手続き，クラス，クラス参照，インターフェースの各型のパラメータに対しては，**nil** 以外のデフォルト値は指定できません。レコード，バリアント，ファイル，静的配列，オブジェクトの各型のパラメータに対しては，デフォルト値は指定できません。

デフォルトのパラメータ値を使ってルーチンを呼び出す方法についての詳細は，6-19 ページの「手続きと関数の呼び出し」を参照してください。

デフォルトパラメータとオーバーロードルーチン

オーバーロードルーチンでデフォルトのパラメータ値を使用する場合は，パラメータ宣言があいまいにならないよう注意してください。次の例を見てください。

```
procedure Confused(I: Integer); overload;
 ⋮
procedure Confused(I: Integer; J: Integer = 0); overload;
 ⋮
Confused(X);    //  どちらの手続きが呼び出されるか？
```

実際にはどちらの手続きも呼び出されません。このコードではコンパイルエラーが発生します。

forward 宣言とインターフェース宣言でのデフォルトパラメータ

forward 宣言されているルーチンまたはユニットのインターフェース部で宣言されているルーチンでデフォルトのパラメータ値を指定する場合は，デフォルトのパラメータ値は **forward** 宣言部分またはインターフェース部で指定しなければなりません。このような場合，定義宣言部分（実現部）ではデフォルト値を指定する必要はありませんが，指定する場合は，**forward** 宣言部分またはインターフェース部での指定と完全に同じ値を指定しなければなりません。

手続きと関数の呼び出し

手続きまたは関数が呼び出されると，呼び出した部分からそのルーチン本体へとプログラムの制御が移ります。ルーチンは，その宣言名（必要に応じて限定子を付加）か，ルーチンを指し示す手続き型変数を使って呼び出します。どちらの方法で呼び出す場合も，パラメータが宣言されているルーチンを呼び出す場合は，指定された型のパラメータをルーチンのパラメータリストで指定されているとおりの順序で渡さなければなりません。ルーチンに渡すパラメータは実パラメータといい，ルーチンの宣言部分で指定されたパラメータは仮パラメータといいます。

ルーチンの呼び出しに際しては，以下の点に留意してください。

- 型付きの定数パラメータおよび値パラメータを渡す式は，対応する仮パラメータと代入の互換性がなければなりません。
- 変数パラメータおよび **out** パラメータを渡す式は，対応する仮パラメータが型なしでない限り，対応する仮パラメータと型が同一でなければなりません。
- 変数パラメータおよび **out** パラメータを渡すのに使える式は，代入可能な式だけです。
- ルーチンの仮パラメータが型なしの場合は，数値および数値が格納された真の（型付き定数でない）定数を実パラメータとして使用することはできません。

デフォルトのパラメータ値が指定されたルーチンを呼び出すとき，デフォルト値をどれかの実パラメータで採用したら，それ以降のすべての実パラメータでデフォルト値を採用しなければなりません。したがって，`SomeFunction(,,X)` という形式の呼び出しは無効です。

ルーチンにすべてデフォルトパラメータだけを渡す場合は，カッコを省略できます。たとえば，次のような手続きがあるとします。

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

この場合，次の 2 つの呼び出しは，どちらも同じ意味になります。

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

オープン配列コンストラクタ

オープン配列コンストラクタを使うと，手続きや関数の呼び出し時に配列を直接作成することができます。オープン配列コンストラクタは，オープン配列パラメータか型可変オープン配列パラメータとしてのみ渡すことができます。

オープン配列コンストラクタは，集合構成子と同じように，式をカンマで区切って全体を大カッコで囲んだものです。たとえば，次のように宣言されているとします。

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

この場合，次のような文で Add 手続きを呼び出すことができます。

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

これは次と同じです。

```
var Temp: array[0..3] of Integer;
```

```
  ⋮
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

オープン配列コンストラクタは，値パラメータか定数パラメータとしてのみ渡すことができます。オープン配列コンストラクタで指定する式は，配列パラメータの基本型と代入の互換性がなければなりません。型可変オープン配列パラメータの場合は，式の型がパラメータごとに異なっていてもかまいません。

第 7 章

クラスとオブジェクト

クラス（クラス型）とは，フィールド，メソッド，およびプロパティから構成される構造を定義したものです。クラス型のインスタンスをオブジェクトといいます。クラスのフィールド，メソッド，およびプロパティは，クラスのコンポーネントまたはメンバーと呼ばれます。

- フィールドは，本質的にはオブジェクトの構成要素である変数です。レコードのフィールドと同じように，クラスのフィールドはクラスのインスタンスの中に存在するデータ項目を表します。
- メソッドとは，クラスに関連付けられた手続きまたは関数を指します。大部分のメソッドは，クラスのインスタンスであるオブジェクトを操作するものです。メソッドの中にはクラス型そのものを操作するメソッド（クラスメソッド）もあります。
- プロパティとは，オブジェクトに関連付けられたデータ（一般にフィールドに格納）にアクセスするときの仲立ちとなるものです。プロパティには，そのデータの読み書き方法を決めるアクセス指定子があります。プログラムのほかの部分（オブジェクトの外）からは，たいていの場合プロパティはフィールドと同じように見えます。

オブジェクトは動的に割り当てられたメモリブロックで，その構造はクラス型により決まります。各オブジェクトはクラスで定義されている全フィールドのコピーをそれぞれ持っていますが，クラスのすべてのインスタンスで同じメソッドが共有されています。オブジェクトは，コンストラクタおよびデストラクタと呼ばれる特別なメソッドにより作成および破棄されます。

クラス型の変数は，実際にはオブジェクトを参照するポインタです。したがって，同じオブジェクトを複数の変数で参照することもできます。ほかのポインタと同じように，クラス型の変数には **nil** 値を格納できます。しかし，ポインタが指し示すオブジェクトにアクセスするのに，クラス型の変数を明示的に逆参照する必要はありません。たとえば，`SomeObject.Size := 100` という式で，SomeObject により参照されるオブジェクトの Size プロパティに値 100 が代入されます。これを `SomeObject^.Size := 100` と記述する必要はありません。

クラス型

クラス型は，インスタンス化する前に宣言し，名前を付けておかなければなりません。なお，クラス型の定義を変数の宣言と同時に行うことはできません。クラスの宣言は手続き宣言または関数宣言の中では行えず，プログラムまたはユニットの最も外側のスコープでのみ行えます。

クラス型の宣言は，次のような形式をとります。

```
type className = class (ancestorClass)
  memberList
 end;
```

className には有効な識別子を指定します。(ancestorClass) はオプションです。memberList では，このクラスのメンバー（フィールド，メソッド，およびプロパティ）を宣言します。(ancestorClass) の指定を省略すると，クラスは定義済みクラス TObject から直接派生されます。(ancestorClass) を指定した場合で，memberList が空の場合は，end は省略してもかまいません。クラス型の宣言の中で，そのクラスで実装するインターフェースを列挙することも可能です。10-4 ページの「インターフェースの実装」を参照してください。

クラスの宣言の中では，メソッドは関数または手続きのヘッダーという形で記述します。本体は記述しません。各メソッドの定義宣言は，プログラム内のほかの場所に記述します。

例として，Delphi の VCL の ComCtrls ユニットに含まれる TListColumns クラスの宣言を次に示します。

```
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;
```

この TMemoryStream は Classes ユニットに含まれる TStream から派生されており，TCollection の大部分のメンバーを継承していますが，コンストラクタメソッド Create など一部のメソッドとプロパティについては独自に定義（再定義）しています。デストラクタ Destroy は TObject からそのまま継承しているため，再定義されていません。各メンバーは **private**，**protected**，**public** のいずれかの可視性で定義されています。**published** として定義されたメンバーはこのクラスにはありません。可視性については 7-4 ページの「クラスメンバーの可視性」を参照してください。

上記の宣言から，次のようにして TMemoryStream を作成できます。

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

継承とスコープ

クラスの宣言時には，その直接の派生元となるクラスを指定できます。次に例を示します。

```
type TSomeControl = class(TControl);
```

このコードでは，TControl の派生クラスである TSomeControl というクラスを宣言しています。クラス型には，その直接の派生元となるクラスのすべてのメンバーが自動的に継承されます。各クラスで新しいメンバーの宣言と継承したメンバーの再定義を行うことはできますが，上位クラスで定義されているメンバーを削除することはできません。したがって，TSomeControl は，TControl とその上位クラスで定義されているメンバーすべてを持つことになります。

メンバーの識別子のスコープは，それが宣言された部分から始まってそのクラス定義の終りまで続き，さらに，そのクラスの派生クラスすべてと，そのクラスとその派生クラスで定義されたすべてのメソッドのブロックにまで及びます。

TObject と TClass

System ユニットで定義されている TObject クラスは，それ以外のすべてのクラスの親になります。TObject では，基本的なコンストラクタやデストラクタなど，ごくわずかなメソッドしか定義されていません。System ユニットでは，Tobject 以外に，次のようにクラス参照型である TClass も定義されています。

```
TClass = class of TObject;
```

TObject についての詳細は，オンラインヘルプを参照してください。クラス参照型についての詳細は，7-23 ページの「クラス参照」を参照してください。

クラス型の宣言時に上位クラスの指定を省略した場合，そのクラスは TObject から直接派生します。したがって次のコードでは，

```
type TMyClass = class
  ⋮
end;
```

上の宣言と下の宣言は同じ意味になります。

```
type TMyClass = class(TObject)
  ⋮
end;
```

しかし，読みやすさの点から，後者の宣言をお勧めします。

クラス型の互換性

クラス型は，その上位クラスと代入の互換性があります。したがって，クラス型の変数で，それよりも下位のクラス型のインスタンスを参照できます。たとえば，次のように宣言されているとします。

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TSquare = class(TRectangle);
var
  Fig: TFigure;
```

この場合，Fig 変数には TFigure，TRectangle，TSquare の 3 つの型の値を代入できます。

オブジェクト型

次の構文を使用すると，クラス型のかわりにオブジェクト型を宣言できます。

```
type objectTypeName = object (ancestorObjectType)
  memberList
end;
```

objectTypeName には有効な識別子を指定します。(ancestorObjectType) はオプションです。memberList では，フィールド，メソッド，およびプロパティを宣言します。(ancestorObjectType) の指定を省略すると，上位オブジェクトを持たないオブジェクト型が作成されます。オブジェクト型ではパブリッシュメンバーを定義することはできません。

オブジェクト型は TObject を継承していないので，オブジェクト型には組み込みのコンストラクタやデストラクタなどのメソッドはありません。オブジェクト型のインスタンスは，New 手続きで作成し，Dispose 手続きで破棄できます。また，レコードを定義するときのように，オブジェクト型の変数だけを定義することもできます。

オブジェクト型は，下位互換性を保つ目的でのみ残されています。そのため，オブジェクト型の使用はお勧めできません。

クラスメンバーの可視性

クラスのメンバーはすべて可視性と呼ばれる属性を持っています。可視性は，**private**，**protected**，**public**，**published**，**automated** のいずれかの予約語で指定します。次に例を示します。

```
published property Color: TColor read GetColor write SetColor;
```

このコードでは，Color というパブリッシュプロパティを宣言しています。可視性により，メンバーにどこからどのようにアクセスできるかが決まります。アクセスできる範囲が最も限定されるのがプライベートメンバーで，アクセスできる範囲がもう少し広いのがプロテクトメンバー，アクセスできる範囲が最も広いのがパブリックメンバー，パブリッシュメンバー，および自動化メンバーです。

可視性指定子が指定されていないメンバーには，1 つ前のメンバーと同じ可視性が与えられます。クラス宣言の先頭で定義されているメンバーで可視性が指定されていないものについては，そのクラスを **{$M+}** 状態でコンパイルした場合，または，**{$M+}** 状態でコンパイルされたクラスから派生させた場合は，デフォルトでパブリッシュの可視性が与えられます。それ以外の場合，このようなメンバーにはパブリックの可視性が与えられます。

コードを読みやすくするため，クラス宣言は，プライベートメンバーを最初にまとめて宣言し，次にすべてのプロテクトメンバーを宣言するというような形で構成するのが望ましいといえます。このような構成にすれば，それぞれの可視性指定子は，クラス宣言の中の 1 つのセクションの始まりを表すという形で一度だけ現れることになります。したがって，典型的なクラス宣言は次のような形式になります。

```
type
  TMyClass = class(TControl)
  private
   ⋮{ プライベート宣言はここに記述 }
  protected
   ⋮{ プロテクト宣言はここに記述 }
```

```
public
  ：{ パブリック宣言はここに記述 }
published
  ：{ パブリッシュ宣言はここに記述 }
end;
```

下位クラスでは，メンバーの可視性を再宣言により広げることはできますが，狭めることはできません。たとえば，プロテクトプロパティを下位クラスでパブリックにすることはできますが，プライベートにすることはできません。なお，パブリッシュメンバーを下位クラスでパブリックにすることはできません。詳細については，7-22 ページの「プロパティのオーバーライドと再宣言」を参照してください。

プライベートメンバー，プロテクトメンバー，パブリックメンバー

プライベートメンバーは，それが属するクラスが宣言されているユニットまたはプログラムの外からは見えません。言い換えれば，プライベートメソッドを別のモジュールから呼び出したり，プライベートフィールドやプライベートプロパティを別のモジュールから読み書きしたりすることはできません。関連するクラス宣言はすべて同じモジュールに置くようにすれば，可視性を広くしなくても，それぞれのクラスのプライベートメンバーにアクセスできるようになります。

プロテクトメンバーは，それが属するクラスが宣言されているモジュール内のどこからでも見え，さらに下位クラスからも見えます。下位クラスがどのモジュールに属しているかは関係ありません。言い換えれば，プロテクトメソッドは，それが宣言されているクラスから派生したクラスに属するメソッドの定義部分から呼び出すことができ，同様に，プロテクトフィールドやプロテクトプロパティは，それが宣言されているクラスから派生したクラスに属するメソッドの定義部分から読み書きできます。派生クラスの実装でのみ使用されるメンバーは，通常，プロテクトメンバーにします。

パブリックメンバーは，それが属するクラスを参照できる場所からであれば，どこからでも見えます。

パブリッシュメンバー

パブリッシュメンバーの可視性はパブリックメンバーと同じです。パブリックメンバーとの違いは，パブリッシュメンバーに対しては実行時型情報（RTTI）が生成されるという点です。RTTI を使用することで，アプリケーションからオブジェクトのフィールドとプロパティの問い合わせを動的に行ったり，オブジェクトのメソッドを探したりすることができます。RTTI は，フォームファイルの保存時およびロード時にプロパティ値にアクセスするときに 使用されるほか，オブジェクトインスペクタでプロパティを表示するときや，特定のメソッド（イベントハンドラ）を特定のプロパティ（イベント）に関連付けるときに使用されます。

published が指定できるプロパティは特定のデータ型に限られます。順序型，文字列型，クラス型，インターフェース型，およびメソッドポインタ型は **published** が指定できます。したがって，集合型は，基本型の上限と下限が 0 ～ 31 の範囲の順序値（バイト，ワード，ダブルワードのいずれかに収まる集合）であれば **published** が指定できます。実数型は，Real48 を除いて **published** が指定できます。配列型のプロパティ（以下に説明する配列プロパティとは異なる）には **published** を指定することはできません。

published を指定できるプロパティの中には，ストリーミングシステムによるサポートが完全ではないものがあります。レコード型のプロパティ，**published** の指定が可能なすべての型の配列プロパティ（7-19 ページの「配列プロパティ」を参照），無名の値を含む列挙型のプロパティ（5-7 ページ

の「明示的に順序値を割り当てられた列挙型」を参照）などがそうです。このようなプロパティに **published** を指定した場合，オブジェクトインスペクタはプロパティを正しく表示できず，ストリーミング処理でディスクにオブジェクトが書き込まれる際にプロパティの値が保護されません。

すべてのメソッドは **published** を指定できますが，同じ名前が付けられた複数のオーバーロードメソッドに対して **published** を指定することはできません。フィールドは，クラス型かインターフェース型の場合のみ **published** を指定できます。

パブリッシュメンバーを含めることができるのは，そのクラスを **{$M+}** 状態でコンパイルする場合，または **{$M+}** 状態でコンパイルされたクラスから派生させた場合のみです。パブリッシュメンバーを持つ大部分のクラスは，**{$M+}** 状態でコンパイルされている TPersistent から派生しています。したがって，**$M** 指令の指定が必要になることはあまりありません。

自動化メンバー

自動化メンバーの可視性はパブリックメンバーと同じです。パブリックメンバーとの違いは，自動化メンバーに対しては，オートメーションサーバーにより必要とされるオートメーション型情報が生成されることです。通常，自動化メンバーは Windows のクラスにのみ存在し，Linux プログラミングでは推奨されません。予約語 **automated** は，現在は下位互換性のために残されています。ComObj ユニットの TAutoObject クラスでは，**automated** は使用されていません。

automated を使って宣言されたメソッドとプロパティには，以下の制限が課されます。

- プロパティ，配列プロパティのパラメータ，メソッドのパラメータ，および関数の戻り値の型は，すべてオートメーション可能な型でなければなりません。オートメーション可能な型は，Byte，Currency，Real，Double，Longint，Integer，Single，Smallint，AnsiString，WideString，TDateTime，Variant，OleVariant，WordBool の各型と，すべてのインターフェース型です。
- メソッド宣言には **register** の呼び出し規約（デフォルト）を使わなければなりません。メソッド宣言で **virtual** を指定することはできますが，**dynamic** を指定することはできません。
- プロパティ宣言でアクセス指定子（**read** と **write**）を指定することはできますが，それ以外の指定子（**index**，**stored**，**default**，および **nodefault**）を指定することはできません。アクセス指定子の後には，デフォルトの呼び出し規約である **register** を使用したメソッド識別子を列挙しなければなりません。フィールド識別子を列挙することはできません。
- プロパティ宣言では型を指定しなければなりません。プロパティをオーバーライドすることはできません。

自動化メソッドと自動化プロパティの宣言では，**dispid** 指令を指定しなければなりません。すでに使われているディスパッチ ID を **dispid** 指令の中に指定すると，コンパイルエラーになります。

Windows では，この指令の後に，オートメーションディスパッチ ID を指定する整数定数を指定しなければなりません。そうでない場合，コンパイラは自動的に，そのクラスおよびそのすべての上位クラスのメソッドとプロパティで使用されている最大のディスパッチ ID より 1 大きい値を，そのメンバーのディスパッチ ID として割り当てます。オートメーションの詳細については，10-10 ページの「オートメーションオブジェクト（Windows のみ）」を参照してください。

forward 宣言と相互に依存するクラス

クラス型の宣言が，次のように予約語 **class** とセミコロンで終わっていて，

```
type className = class;
```

class の後に上位クラスもクラスのメンバーも指定されていない場合，その宣言は **forward** 宣言です。**forward** 宣言は同じ型宣言部の中にある同じクラスの定義宣言により解決しなければなりません。つまり，前方参照宣言とその定義宣言の間に，ほかの型宣言以外のものを含めることはできません。

forward 宣言を使用することで，相互に依存したクラスを使用できるようになります。次に例を示します。

```
type
  TFigure = class;  // forward 宣言
  TDrawing = class
    Figure: TFigure;
      ⋮
  end;
  TFigure = class;  // 定義宣言
    Drawing: TDrawing;
      ⋮
  end;
```

TObject から派生させ，クラスのメンバーをまったく宣言していない完全な型の宣言と，**forward** 宣言とを混同しないでください。

```
type
  TFirstClass = class;           // これは forward 宣言
  TSecondClass = class           // これは完全なクラス宣言
    end;                         //
  TThirdClass = class(TObject);  // これは完全なクラス宣言
```

フィールド

フィールドとは，オブジェクトに属する変数のようなものです。フィールドは，クラス型も含め，どのような型にでもできます。つまり，フィールドにオブジェクト参照を格納することも可能です。フィールドには，通常，**private** を指定します。

クラスのフィールドメンバーを定義するには，変数の場合と同じようにフィールドを宣言します。すべてのフィールドは，プロパティやメソッドより前に宣言しなければなりません。次のコードでは，TObject から継承したメソッド以外には Int という整数型フィールドしか持たない，TNumber というクラスを宣言しています。

```
type TNumber = class
  Int: Integer;
end;
```

フィールドは静的に結合されます。したがって，フィールドへの参照はコンパイル時に解決されます。以下のコードを例にとって，これを具体的に見てみます。

```
type
  TAncestor = class
```

```
    Value: Integer;
  end;
  TDescendant = class(TAncestor)
    Value: string;  // 継承された Value フィールドは隠蔽される
  end;
var
  MyObject: TAncestor;
begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!';  // error
  TDescendant(MyObject).Value := 'Hello!';  // OK
end;
```

MyObject には Tdescendant のインスタンスが格納されますが，MyObject は TAncestor として宣言されています。したがって，コンパイラは MyObject.Value を TAncestor で宣言されているフィールド（整数型）への参照と解釈します。しかし，どちらのフィールドも TDescendant オブジェクトに存在します。継承された Value は，新しい Value によって隠蔽されていますが，型キャストによりアクセス可能です。

メソッド

メソッドとは，クラスに関連付けられた手続きまたは関数を指します。メソッドの呼び出し時には，操作対象となるオブジェクト（クラスメソッドの場合はクラス）を指定します。次に例を示します。

```
SomeObject.Free
```

このコードでは，SomeObject で定義されている Free メソッドを呼び出しています。

メソッドの宣言と実装

メソッドは，クラス宣言の中では **forward** 宣言と同じような働きをする手続きまたは関数のヘッダーという形で記述します。メソッドは，同じモジュール内の，クラス宣言よりも後のどこかで定義宣言によって実装しなければなりません。たとえば，次のように TMyClass の宣言の中で DoSomething というメソッドが呼び出されているとします。

```
type
  TMyClass = class(TObject)
      ⋮
    procedure DoSomething;
      ⋮
  end;
```

DoSomething の定義宣言は，次のような形式で，同じモジュール内のこれよりも後の方に記述しなければなりません。

```
procedure TMyClass.DoSomething;
begin
 ⋮
end;
```

クラスの宣言はユニットのインターフェース部でも実現部でも行えますが，クラスのメソッドの定義宣言は実現部に記述しなければなりません。

定義宣言のヘッダーでは，メソッド名は必ずメソッドが属するクラスの名前で限定します。定義宣言のヘッダーには，クラス宣言で指定したパラメータリストを再度指定してもかまいませんが，その場合は，パラメータの順序，型，および名前をクラス宣言での指定と完全に同じにしなければなりません。また，メソッドが関数の場合は，戻り値の型も一致させなければなりません。

メソッド宣言には，ほかの関数や手続きでは使われない固有の指令を含めることができます。指令は定義宣言ではなくクラス宣言でのみ記述し，常に次の順序で指定しなければなりません。

`reintroduce`；`overload`；binding；calling convention；`abstract`；warning

ここで，binding は **virtual**，**dynamic**，または **override**，calling convention は **register**，**pascal**，**cdecl**，**stdcall**，または **safecall**，そして warning は **platform**，**deprecated**，または **library** です。

inherited

予約語 **inherited** は，多態性の動作を実現する上で特別な役割を演じます。**inherited** はメソッドの定義時に使用し，後に識別子を伴う場合と伴わない場合とがあります。

inherited の後にメンバーの名前を指定した場合，これは通常のメソッド呼び出し，またはプロパティやフィールドへの参照になります。ただし，この場合，参照先メンバーの検索は，記述中のメソッドのクラスの直接の派生元から始まります。たとえば，次の行をメソッドの定義部分に置くと，

```
inherited Create(...);
```

継承された Create が呼び出されます。

inherited の後に識別子を指定しない場合は，記述中のメソッドと同じ名前を持つ親メソッドが参照されます。この場合，**inherited** はパラメータを明示的には取りませんが，親メソッドには，記述中のメソッドが呼び出されるときと同じパラメータが渡されます。次に例を示します。

```
inherited;
```

これはコンストラクタの実装で頻繁に使用され，派生コンストラクタに渡されたパラメータと同じパラメータで，親コンストラクタが呼び出されます。

Self

メソッドの実装部分では，Self 識別子はそのメソッドが呼び出されたオブジェクトを参照します。例として，Classes ユニットで定義されている TCollection の Add メソッドを次に示します

```
function TCollection.Add: TCollectionItem;
begin
  Result := FItemClass.Create(Self);
end;
```

この Add メソッドでは，FItemClass フィールドによって参照されるクラス（常に TCollectionItem の派生クラス）の Create メソッドを呼び出しています。TCollectionItem.Create は TCollection 型のパラメータを 1 つとるので，Add では TCollectionItem.Create に自分自身が呼び出されたインスタンスオブジェクト TCollection を渡しています。これを次のコードを使って説明します。

```
var MyCollection: TCollection;
 ⋮
```

```
MyCollection.Add  // MyCollection は TCollectionItem.Create メソッドに渡される
```

Self が便利であるのにはさまざまな理由があります。たとえば，あるクラス型で宣言されているメンバー識別子が，そのクラスのメソッドのブロックで再宣言されているとします。このような場合，Self.Identifier と指定することで元のメンバー識別子にアクセスできます。

クラスメソッドで使用された場合の Self については，7-26 ページの「クラスメソッド」を参照してください。

メソッドの結合

メソッドは，静的（デフォルト），仮想，動的のいずれかとして宣言できます。仮想メソッドと動的メソッドはオーバーライドでき，また，抽象メソッドにすることもできます。このような指定は，あるクラス型の変数に下位クラス型の値が格納されたときに意味を持ちます。これらの指定により，メソッドが呼び出されたとき，どの実装が起動されるかが決まります。

静的メソッド

メソッドは，デフォルトでは静的メソッドになります。静的メソッドが呼び出されると，メソッド呼び出しで使われたクラス変数またはオブジェクト変数の宣言時（コンパイル時）の型によって，どの実装を起動するかが決まります。次の例で，Draw メソッドは静的メソッドです。

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

このように宣言されていることを前提として，静的メソッドを呼び出したときの効果を下のコードを使って説明します。Figure.Draw の 2 番目の呼び出しでは，Figure 変数は TRectangle クラスのオブジェクトを参照していますが，Figure 変数の宣言された型が TFigure であるため，TFigure 内の Draw の実装が呼び出されます。

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw;  // calls TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;  // TFigure.Draw を呼び出す
  TRectangle(Figure).Draw;  // TRectangle.Draw を呼び出す
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;  // TRectangle.Draw を呼び出す
  Rectangle.Destroy;
end;
```

仮想メソッドと動的メソッド

メソッドを仮想メソッドにするには，その宣言時に **virtual** 指令を指定し，動的メソッドにするには **dynamic** 指令を指定します。静的メソッドと違って，仮想メソッドと動的メソッドは下位クラスでオーバーライドできます。オーバーライドされたメソッドの呼び出し時には，メソッド呼び出しで指定されたクラス変数またはオブジェクト変数の宣言時の型ではなく，実際の（実行時の）型によって，どのメソッド実装を起動するかが決まります。

メソッドをオーバーライドするには，**override** 指令を使ってメソッドを再宣言します。オーバーライド宣言するときは，そのパラメータの型と指定の順序を，上位クラスでの宣言と合わせなければなりません。戻り値がある場合は，その型も合わせます。

次の例では，TFigure で宣言されている Draw メソッドが 2 つの下位クラスでオーバーライドされています。

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

このように宣言されていることを前提として，実行時に実際の型が変化する変数を使用して仮想メソッドを呼び出したときの効果を下のコードを使って説明します。

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;  // TRectangle.Draw を呼び出す
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;  // TEllipse.Draw を呼び出す
  Figure.Destroy;
end;
```

オーバーライドできるのは仮想メソッドと動的メソッドだけです。しかし，どのメソッドでもオーバーロードはできます。「メソッドのオーバーロード」を参照してください。

仮想メソッドと動的メソッドの比較

仮想メソッドと動的メソッドは意味的には等価です。実行時におけるメソッド呼び出しのディスパッチの実現方法が異なるだけです。仮想メソッドを使用すると実行速度を最適化できるのに対し，動的メソッドを使用するとコードのサイズを最適化できます。

一般に，多態性の動作を実装する一番効率の良い方法は仮想メソッドを使うことです。オーバーライド可能なメソッドが基本クラスで多数宣言されており，それらがアプリケーション内で多くの下位クラスにより継承されているが，オーバーライドされるのはそれほど頻繁ではないという場合は，動的メソッドを使用すると便利です。

オーバーライドと隠蔽

継承されたメソッドと同じメソッド識別子とパラメータ宣言をメソッドの宣言で指定しても，**override** を指定しなかった場合は，継承されたメソッドが新しいメソッド宣言によって隠蔽されるだけで，オーバーライドはされません。どちらのメソッドも，メソッド名が静的に結合された下位クラスに存在します。次に例を示します。

```
type
  T1 = class(TObject)
    procedure Act; virtual;
  end;
  T2 = class(T1)
    procedure Act;  // Act が再宣言されているがオーバーライドはされない
  end;
var
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act;  // T1.Act が呼び出される
end;
```

reintroduce

reintroduce 指令を指定すると，前に宣言された仮想メソッドが隠蔽されたことを警告するコンパイラのメッセージを表示しないようにすることができます。次に例を示します。

```
procedure DoSomething; reintroduce;  // 上位クラスにも DoSomething メソッドがある
```

継承された仮想メソッドを新しいメソッドで隠蔽したい場合は，**reintroduce** を使用してください。

抽象メソッド

抽象メソッドとは，それが宣言されているクラスで実装が行われていない仮想メソッドまたは動的メソッドを指します。抽象メソッドの実装は派生クラスで行われます。抽象メソッドは，**virtual** 指令または **dynamic** 指令の後に **abstract** 指令を指定して宣言しなければなりません。次に例を示します。

```
procedure DoSomething; virtual; abstract;
```

抽象メソッドは，その抽象メソッドをオーバーライドしたクラス，または，そのようなクラスのインスタンスでのみ呼び出すことができます。

メソッドのオーバーロード

メソッドは，**overload** 指令を使用して再宣言できます。**overload** 指令を使用して再宣言すると，再宣言されたメソッドのパラメータ宣言が元のメソッドと異なる場合，継承されたメソッドは隠蔽されるのではなくオーバーロードされます。下位クラスでこのメソッドを呼び出すと，呼び出し時に使用されたパラメータに合致する実装が起動されます。

仮想メソッドをオーバーロードする場合は，それを下位クラスで再宣言するときに **reintroduce** 指令を使用してください。次に例を示します。

```
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!');  // T2.Test が呼び出される
SomeObject.Test(7);         // T1.Test が呼び出される
```

同じクラス内で，同じ名前を持つ複数のオーバーロードメソッドに対して **publish** を指定することはできません。実行時型情報の管理のため，パブリッシュメンバーにはそれぞれ異なる名前が付けられていなければなりません。

```
type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer  // エラー
     ⋮
```

プロパティの **read** または **write** 指定子として機能するメソッドは，オーバーロードすることはできません。

オーバーロードメソッドの実装部分では，クラス宣言部分で指定したパラメータリストと同じパラメータリストを指定しなければなりません。オーバーロードについての詳細は 6-8 ページの「手続きと関数のオーバーロード」を参照してください。

コンストラクタ

コンストラクタは，インスタンスオブジェクトの作成と初期化を行う特別なメソッドです。コンストラクタの宣言は手続きの宣言に似ていますが，コンストラクタの宣言は予約語 **constructor** で始まります。例を以下に示します。

```
constructor Create;
constructor Create(AOwner: TComponent);
```

コンストラクタの宣言には，デフォルトの呼び出し規約である **register** を使わなければなりません。コンストラクタの宣言では戻り値を指定しませんが，コンストラクタが作成したオブジェクトまたはコンストラクタを呼び出したオブジェクトへの参照が返されます。

クラスに複数のコンストラクタを持たせることも可能ですが，通常は 1 つだけです。Create という名前のコンストラクタを呼び出すのが慣例になっています。

オブジェクトを作成するには，クラス型の中でコンストラクタメソッドを呼び出します。次に例を示します。

```
MyObject := TMyClass.Create;
```

このコードにより，新しいオブジェクト用の記憶域がヒープに割り当てられ，すべての順序型フィールドの値がゼロになり，すべてのポインタ型フィールドとクラス型フィールドが **nil** になり，すべての文字列フィールドが空になります。この後，コンストラクタの実装部分で指定されたほかの処理が行われます。コンストラクタにパラメータとして渡された値を基にオブジェクトを初期化するというのが一般的な処理です。最後に，コンストラクタは新たに割り当てられ初期化されたオブジェクトへの参照を返します。戻り値の型はコンストラクタ呼び出しで指定したクラス型と同じです。

クラス参照により呼び出されたコンストラクタの実行中に例外が生成された場合，Destroy デストラクタが自動的に呼び出されて未完成のオブジェクトが破棄されます。

クラス参照ではなくオブジェクト参照によりコンストラクタが呼び出された場合は，オブジェクトは作成されません。この場合に行われることは，指定されたオブジェクトの操作であり，コンストラクタの実装部分で指定された文だけが実行され，そのオブジェクトへの参照が返されます。一般に，コンストラクタをオブジェクト参照で呼び出すのは，予約語 **inherited** を使って親コンストラクタを実行する場合だけです。

クラス型とそのコンストラクタの例を次に示します。

```
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);  // 継承された部分を初期化
  Width := 65;  // 継承したプロパティを変更
  Height := 65;
  FPen := TPen.Create;  // 新しいフィールドを初期化
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

通常，コンストラクタはまず親コンストラクタを呼び出して，オブジェクトの継承されたフィールドを初期化します。次に，コンストラクタは下位クラスで追加されたフィールドを初期化します。コンストラクタは新しいオブジェクト用に割り当てた記憶域を必ずクリアするため，すべてのフィールドにゼロ（順序型），**nil**（ポインタ型とクラス型），空文字列（文字列型），Unassigned（バリアント型）のいずれかの値が割り当てられます。したがって，ゼロ以外の値や空文字列以外の値を割り当てるのでなければ，コンストラクタの実装部分でフィールドの初期化を行う必要はありません。

クラス型の識別子を使って呼び出す場合，**virtual** として宣言したコンストラクタは静的コンストラクタと同じです。ただし，クラス参照型を使うと，多態性を考慮したオブジェクトの作成，つまりコンパイル時に型がわからないオブジェクトの作成が仮想コンストラクタでできるようになります（7-23 ページの「クラス参照」を参照）。

デストラクタ

デストラクタは，自分自身を呼び出したオブジェクトを破棄し，そのメモリを解放する特別なメソッドです。デストラクタの宣言は手続きの宣言に似ていますが，デストラクタの宣言は予約語 **destructor** で始まります。例を以下に示します。

```
destructor Destroy;
destructor Destroy; override;
```

デストラクタの宣言には，デフォルトの呼び出し規約である **register** を使わなければなりません。クラスに複数のデストラクタを持たせることも可能ですが，デストラクタは継承した Destroy メソッドを各クラスでオーバーライドするようにし，それ以外のデストラクタは宣言しないようにすることをお勧めします。

デストラクタを呼び出すときは，インスタンスオブジェクトを参照しなければなりません。次に例を示します。

```
MyObject.Destroy;
```

デストラクタが呼び出されると，デストラクタの実装部分で指定された処理がまず実行されます。通常，ここで行う処理は，埋め込みオブジェクトの破棄と，オブジェクトにより割り当てられたリソースの解放です。その後，オブジェクト用に割り当てられた記憶域を破棄します。

デストラクタの例を次に示します。

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

通常，デストラクタの実装コードの最後の処理は，継承したデストラクタを呼び出してオブジェクトの継承フィールドを破棄することです。

オブジェクトの作成中に例外が生成された場合は，Destroy が自動的に呼び出され，未完成のオブジェクトが破棄されます。つまり，部分的にしか完成していないオブジェクトの破棄処理のために Destroy を用意しておかなければなりません。コンストラクタは新しいオブジェクトのすべてのフィールドをゼロまたは空に設定してからそれ以外の処理を実行するため，部分的にしか完成していないオブジェクトのクラス型フィールドとポインタ型フィールドは必ず **nil** になります。したがって，クラス型フィールドやポインタ型フィールドの操作をする前に，**nil** 値でないかどうかをチェックするのが望ましいといえます。Destroy のかわりに Free メソッド（TObject で定義）を呼び出すと，オブジェクトを破棄する前に **nil** 値のチェックができるので便利です。

メッセージメソッド

メッセージメソッドは，動的にディスパッチされるメッセージへの応答処理を実装したメソッドです。メッセージメソッドの構文は，すべてのプラットフォームでサポートされます。VCL では，メッセージメソッドを使って Windows メッセージに応答します。CLX では，システムイベントへの応答にメッセージメソッドは使用しません。

メッセージメソッドは，メソッド宣言で **message** 指令を指定し，その後にメッセージ ID（1 ～ 49151 の範囲の整数定数）を指定するという方法で作成します。VCL コントロールでのメッセージメソッドでは，この整数定数は，Messages ユニットで対応するレコード型とともに定義されている Windows メッセージ ID のいずれかを使用できます。メッセージメソッドは，単一の **var** パラメータを取る手続きでなければなりません。

次に Windows の例を示します。

```
type
  TTextBox = class(TCustomControl)
    private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ⋮
  end;
```

Linux またはクロスプラットフォームプログラミングでは，たとえば次のようにメッセージを処理できます。

```
const
  ID_REFRESH = $0001;
type
  TTextBox = class(TCustomControl)
  private
    procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
    ...
  end;
```

メッセージメソッドに **override** 指令を指定して，継承されたメッセージメソッドをオーバーライドする必要はありません。オーバーライドするメソッドと同じメソッド名やパラメータ型を指定する必要もありません。メッセージ ID は単に，メソッドがどのメッセージに応答するかと，オーバーライドであるかどうかを決めるだけです。

メッセージメソッドの実装

次の例に示すように，メッセージメソッドの実装部分から親メッセージメソッドを呼び出すことができますの（Windows の場合）。

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;
```

Linux またはクロスプラットフォームプログラミングでは，同じ例を次のように書くことができます。

```
procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
  if Chr(Message.Code) = #13 then
    ...
  else
    inherited;
end;
```

inherited 文は，クラス階層を逆方向にたどって現在のメソッドと同じ ID が指定された最初のメッセージメソッドを探し，そのメッセージメソッドにメッセージレコードを自動的に渡してこれを呼び出します。当該 ID を処理するメッセージメソッドを実装した上位クラスがない場合は，TObject で定義されている DefaultHandler が呼び出されます。

TObject での DefaultHandler の実装は何の処理もせずにただ戻るだけです。DefaultHandler をオーバーライドすることによって，クラスはメッセージのデフォルト処理を独自に実装できます。Windows で

は，VCL コントロール用の DefaultHandler メソッドは，Windows の DefWindowProc 関数を呼び出します。

メッセージのディスパッチ

メッセージメソッドが直接呼び出されることはめったにありません。かわりに，メッセージは TObject から継承された Dispatch メソッドでオブジェクトにディスパッチされます。

```
procedure Dispatch(var Message);
```

Dispatch に渡す Message パラメータはレコードでなければならず，そのレコード内の最初のエントリは，メッセージ ID を保持する Cardinal 型のフィールドでなければなりません。

Dispatch は自分が呼び出されたオブジェクトのクラスを出発点としてクラス階層を逆方向に検索し，渡された ID を処理する最初のメッセージメソッドを呼び出します。当該 ID を処理するメッセージメソッドが見つからない場合は，DefaultHandler が呼び出されます。

プロパティ

プロパティは，フィールドに似ており，オブジェクトの属性を定義するものです。しかし，フィールドが単にデータの格納場所で，その内容を参照したり変更することしかできないのに対し，プロパティでは，そのデータの読み出し操作または変更操作に対して特定の処理を関連付けることができます。プロパティは，オブジェクトの属性へのアクセスを制御する働きを持ち，属性の計算も可能にします。

プロパティの宣言では名前と型を指定し，さらに，アクセス指定子を最低 1 つ指定します。プロパティ宣言の構文を次に示します。

```
property propertyName[indexes]: type index integerConstant specifiers;
```

各要素の説明は次のとおりです。

- propertyName には有効な識別子を指定します。
- [indexes] はオプションで，パラメータ宣言をセミコロンで区切って指定します。各パラメータ宣言は，$identifier_1$，...，$identifier_n$：type という形式をとります。詳細については，7-19 ページの「配列プロパティ」を参照してください。
- type はあらかじめ定義または宣言されたデータ型でなければなりません。したがって，**property** Num: 0..9 ... のようなプロパティ宣言は無効です。
- index integerConstant 節はオプションです。詳細については，7-21 ページの「インデックス指定子」を参照してください。
- specifiers には，**read**，**write**，**stored**，**default**（または **nodefault**），**implements** の各指定子を列挙します。プロパティ宣言では，最低限 **read** か **write** のいずれかのアクセス指定子を指定しなければなりません。**implements** についての詳細は，10-6 ページの「委任によるインターフェースの実装」を参照してください。

プロパティは，そのアクセス指定子により定義されます。フィールドとは違い，プロパティは変数パラメータとして渡すことはできません。また，プロパティに対してアドレス演算子を使用することも

できません。これは，プロパティは必ずメモリ上に存在しているとは限らないからです。しかし，たとえば **read** メソッドを使って，データベースから値を取り出したり無作為な値を生成したりすることは可能です。

プロパティのへアクセス

すべてのプロパティには，**read** 指定子か **write** 指定子，またはその両方を指定します。これらの指定子はアクセス指定子と呼ばれ，次のような形式で指定します。

```
read fieldOrMethod
write fieldOrMethod
```

fieldOrMethod には，プロパティと同じクラスまたは上位クラスで宣言されているフィールドまたはメソッドの名前を指定します。

- fieldOrMethod を同じクラスで宣言する場合は，プロパティ宣言より前で宣言しなければなりません。fieldOrMethod を上位クラスで宣言している場合は，下位クラスからも可視でなければなりません。つまり，別のユニットで宣言されている上位クラスのプライベートフィールドやプライベートメソッドを fieldOrMethod に指定することはできません。
- fieldOrMethod にフィールドを指定する場合は，プロパティと同じ型のフィールドでなければなりません。
- fieldOrMethod がメソッドである場合，オーバーロードすることはできません。また，パブリッシュプロパティのアクセスメソッドには，**register** 呼び出し規約（デフォルト）を使わなければなりません。
- **read** 指定子の後の fieldOrMethod にメソッドを指定する場合，メソッドは，戻り値の型がプロパティの型と同じで，パラメータをとらない関数でなければなりません。
- **write** 指定子の後の fieldOrMethod にメソッドを指定する場合，メソッドは，プロパティと同じ型の値パラメータまたは定数パラメータを 1 つとる手続きでなければなりません。

たとえば，次のような宣言があるとします。

```
property Color: TColor read GetColor write SetColor;
```

GetColor メソッドは次のように宣言されていなければなりません。

```
function GetColor: TColor;
```

さらに，SetColor メソッドの定義は次のいずれかでなければなりません。

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

ただし，SetColor のパラメータの名前は Value でなくてもかまいません。

プロパティを式の中で参照すると，**read** 指定子の後に指定されたフィールドまたはメソッドを使ってその値が読み出されます。プロパティを代入文の中で参照すると，**write** 指定子の後に指定されたフィールドまたはメソッドを使ってその値が書き込まれます。

下の例では，Heading というパブリッシュプロパティを持つ TCompass というクラスを宣言しています。Heading の値は，FHeading フィールドから読み出され，SetHeading 手続きによって書き込まれます。

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    ⋮
  end;
```

上のように定義されている場合，次のコードは

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

次のように記述するのと同等です。

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

TCompass クラスでは，Heading プロパティの読み出し操作に対して何の処理も関連付けられていません。したがって，このプロパティの読み出し操作で行われることは，FHeading フィールドに格納されている値を取り出すことです。一方，Heading プロパティへの値の代入操作は SetHeading メソッドの呼び出しに変換されます。SetHeading メソッドの処理内容は，新しい値を FHeading フィールドに格納するとともに，それ以外のなんらかの処理を行うことです。たとえば，SetHeading は次のように実装できます。

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;  // ユーザーインターフェースを更新して新しい値を反映
  end;
end;
```

プロパティ宣言に **read** 指定子しかないプロパティは読み出し専用プロパティで，**write** 指定子しかないプロパティは書き込み専用プロパティです。読み出し専用プロパティに値を代入したり，書き込み専用プロパティを式の中で使うとエラーになります。

配列プロパティ

配列プロパティはインデックス付き（添字付き）プロパティです。配列プロパティを使うと，リスト中の項目や，コントロールの子コントロール，ビットマップのピクセルなどを表現できます。

配列プロパティの宣言には，インデックスの名前と型を指定するパラメータリストが含まれます。次に例を示します。

```
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;
```

パラメータ宣言を囲むのにカッコ ( ) のかわりに大カッコ [ ] を使うことを除いて，インデックスパラメータリストの形式は手続きや関数のパラメータリストの場合と同じです。配列では順序型のインデックスだけしか指定できませんが，配列プロパティではどのような型のインデックスでも指定できます。

配列プロパティでは，アクセス指定子の後にはフィールドではなくメソッドを指定しなければなりません。**read** 指定子で指定するメソッドは，プロパティのインデックスパラメータリストと同じ数と型のパラメータを同じ順番でとる関数でなければならず，その関数の戻り値の型はプロパティの型と一致しなければなりません。**write** 指定子で指定するメソッドは，プロパティのインデックスパラメータリストと同じ数と型のパラメータを同じ順番でとり，それに加えてプロパティと同じ型の値パラメータまたは定数パラメータを 1 つとる手続きでなければなりません。

たとえば，上記の配列プロパティで使用するアクセスメソッドは以下のように宣言できます。

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

配列プロパティは，プロパティ識別子にインデックスを指定することでアクセスします。たとえば，次のコードは，

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

次のように記述するのと同等です。

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

Linux では，上の例の 'C:\DELPHI\BIN' ではなく，'/usr/local/bin' などのようなパスを使用します。

配列プロパティの定義の後には **default** 指令を続けることができ，その場合，その配列プロパティはクラスのデフォルトプロパティになります。次に例を示します。

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    ⋮
  end;
```

クラスにデフォルトプロパティがある場合は，object[index] という省略形でプロパティにアクセスできます。これは，object.property[index] と指定するのと同じです。たとえば，上記のように宣言されている場合，`StringArray.Strings[7]` を `StringArray[7]` と省略して指定できます。クラスは，デフォルトプロパティを 1 つだけ持つことができます。コンパイラは常にデフォルトプロパティを静的に決定するので，下位クラスでデフォルトプロパティを変更または隠蔽すると，予期しない結果が生じることがあります。

インデックス指定子

インデックス指定子を使うと，複数のプロパティで同じアクセスメソッドを共有できます。その場合でも，各プロパティは異なる値を表すことができます。インデックス指定子を指定するときは，**index** 指令の後に -2147483647 〜 2147483647 の範囲の整数定数を指定します。次に例を示します。

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Longint;
    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;
```

インデックス指定子を持つプロパティのアクセスメソッドは，Integer 型の追加の値パラメータを 1 つとる必要があります。この値パラメータは，読み出し関数の場合は最終パラメータとして指定しなければなりません。書き込み手続きの場合は最後から 2 番目のパラメータとして指定しなければなりません（プロパティの値を指定するパラメータを最後に指定）。インデックス指定子を持つプロパティにアクセスすると，プロパティの整数定数がアクセスメソッドに自動的に渡されます。

上記のように宣言されている場合，Rectangle の型が TRectangle 型だとすると，次のコードは，

```
Rectangle.Right := Rectangle.Left + 100;
```

次のように記述するのと同等です。

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

格納指定子

オプションの指令である **stored**，**default**，**nodefault** の 3 つの指令は格納指定子と呼ばれます。格納指定子はプログラムの動作に影響を及ぼすものではなく，実行時型情報（RTTI）の管理方法を制御するものです。具体的には，格納指定子により，パブリッシュプロパティの値をフォームファイルに保存するかどうかが決まります。

stored 指令の後には，True，False，Boolean 型のフィールドの名前，論理値を返す，パラメータをとらないメソッドの名前のいずれかを指定しなければなりません。次に例を示します。

```
property Name: TComponentName read FName write SetName stored False;
```

プロパティに **stored** 指令が指定されていない場合，そのプロパティは `stored True` が指定されているものとして扱われます。

default 指令の後には，プロパティと同じ型の定数を指定しなければなりません。次に例を示します。

```
property Tag: Longint read FTag write FTag default 0;
```

継承された **default** 値を新しい値を指定せずにオーバーライドするには，**nodefault** 指令を使います。**default** 指令と **nodefault** 指令は，集合の基本型の上限と下限が 0 〜 31 の範囲の順序値である順序型と集合型のみサポートしています。継承されたプロパティが **default** または **nodefault** の指定なしで宣言されている場合，そのプロパティ宣言は **nodefault** が指定されているものとして扱われます。実数型，ポインタ，および文字列型では，暗黙の **default** 値は，それぞれ 0，**nil**，および ''（空文字列）です。

コンポーネントの状態の保存時には，そのコンポーネントのパブリッシュプロパティの格納指定子がチェックされます。プロパティの現在の値がそのデフォルト値と異なり（あるいは **default** 値がない），かつ，**stored** 指定子が True の場合，そのプロパティの値は保存されます。そうでない場合は，そのプロパティの値は保存されません。

メモ 格納指定子は，配列プロパティはサポートしていません。**default** 指令は，配列プロパティの宣言で使用した場合は意味が異なります。7-19 ページの「配列プロパティ」を参照してください。

プロパティのオーバーライドと再宣言

型が指定されていないプロパティ宣言は，プロパティオーバーライドと呼ばれます。プロパティをオーバーライドすることにより，継承されたプロパティの可視性と指定子を変更できます。最も簡単なオーバーライド方法は，予約語 **property** の後に，継承するプロパティの識別子を指定することです。この形式は，プロパティの可視性を変更するときに使われます。たとえば，あるプロパティが上位クラスで **protected** として宣言されている場合，派生クラスの **public** 部または **published** 部でそのプロパティを再宣言するといったことができます。プロパティオーバーライドでは，**read**，**write**，**stored**，**default**，**nodefault** の各指令を指定できます。これらの指令は，継承された指令のうち対応するものをオーバーライドします。オーバーライドにより，継承されたアクセス指定子を別のアクセス指定子に変えたり，足りない指定子を追加したり，プロパティの可視性を広げたりすることはできますが，アクセス指定子を削除したりプロパティの可視性を狭めたりすることはできません。オーバーライドでは，継承されたインターフェースを削除せずに，実装されたインターフェースのリストに追加する **implements** 指令を指定できます。

プロパティオーバーライドの使い方を，次のコードを例にとって説明します。

```
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 をオーバーライドしている行では，**write** 指定子を追加してプロパティの値を変更できるようにしています。Text と Color をオーバーライドしている行では，この 2 つのプロパティの可視性を **protected** から **published** に変更しています。Color をオーバーライドしている行では，その値が clBlue でない場合には値を保存することを指定しています。

プロパティの再宣言で型識別子が指定されている場合は，継承されたプロパティはオーバーライドされるのではなく隠蔽されます。これは，継承されたプロパティと同じ名前のプロパティが新たに作成されることを意味します。型を指定したプロパティ宣言は完全な宣言でなければなりません。したがって，型を指定したプロパティ宣言では，アクセス指定子を最低 1 つは指定しなければなりません。

プロパティが派生クラスで隠蔽された場合でもオーバーライドされた場合でも，プロパティの参照は常に静的になります。つまり，オブジェクトの識別に使用される変数の宣言時（コンパイル時）の型により，そのプロパティ識別子の解釈が決まるということです。したがって，下のコードの実行後，MyObject には TDescendant のインスタンスが格納されますが，それでも，MyObject.Value の値を読み出すと Method1 が呼び出され，MyObject.Value に値を書き込むと Method2 が呼び出されます。しかし，MyObject を TDescendant にキャストして，下位クラスのプロパティとそのアクセス指定子にアクセスすることは可能です。

```
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;
```

クラス参照

通常，操作はクラスのインスタンス（オブジェクト）に対して行われますが，クラスそのものに対して操作が行われることもあります。コンストラクタメソッドをクラス参照を使って呼び出す場合などがそうです。クラスは常にその名前により参照できますが，値としてクラスをとる変数やパラメータを宣言しなければならない状況では，クラス参照型が必要になります。

クラス参照型

メタクラスと呼ばれることもあるクラス参照型は，次の形式の構文で指定します。

```
class of type
```

type にはクラス型を指定します。type 識別子自体は，型が `class of type` である値を表します。$type_1$ が $type_2$ の上位クラスである場合，`class of` $type_2$ は `class of` $type_1$ と代入の互換性があります。したがって次のコードでは，

```
type TClass = class of TObject;
var AnyObj: TClass;
```

任意のクラスへの参照を格納できる AnyObj という変数を宣言しています（クラス参照型は変数宣言時に直接宣言したり，パラメータリスト内で直接宣言することはできません）。**nil** 値は，任意のクラス参照型の変数に代入できます。

クラス参照型がどのように使用されるかを学ぶため，次の TCollection のコンストラクタの宣言を見てください（Classes ユニットで定義）。

```
type TCollectionItemClass = class of TCollectionItem;
  ⋮
constructor Create(ItemClass: TCollectionItemClass);
```

この宣言は，TCollection のインスタンスオブジェクトを作成するためには，TCollectionItem の派生クラスの名前をコンストラクタに渡さなければならないことを示しています。

クラス参照型は，その実際の型がコンパイル時にわからないクラスまたはオブジェクトでクラスメソッドまたは仮想コンストラクタを呼び出したい場合に便利です。

コンストラクタとクラス参照

コンストラクタは，クラス参照型の変数を使って呼び出すことができます。これにより，コンパイル時にその型がわからないオブジェクトの作成が可能になります。次に例を示します。

```
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;
```

どのような種類のコントロールを作成するかを伝えるため，CreateControl 関数にはクラス参照パラメータを渡す必要があります。CreateControl 関数は，このパラメータを使用してクラスのコンストラクタを呼び出します。クラス型の識別子はクラス参照の値を表すため，CreateControl を呼び出すときに，インスタンスを作成したいクラスの識別子を指定できます。次に例を示します。

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

クラス参照により呼び出されるコンストラクタは，通常，仮想コンストラクタです。どのコンストラクタの実装が呼び出されるかは，指定するクラス参照の実行時の型によって決まります。

クラス演算子

すべてのクラスは，ClassType と ClassParent という TObject のメソッドを継承しています。この 2 つのメソッドは，それぞれ，オブジェクトのクラスへの参照と，オブジェクトの直接の派生元への参照を返します。どちらのメソッドも，より具体的な型にキャスト可能な TClass 型（`TClass = class of TObject`）の値を返します。また，すべてのクラスは，呼び出し元のオブジェクトが指定したクラスから派生されたものであるかどうかをチェックする InheritsFrom というメソッドも継承しています。これらのメソッドは **is** 演算子と **as** 演算子によって使用され，直接呼び出さなければならないことはほとんどありません。

is 演算子

is 演算子は動的型チェックを行う演算子で，オブジェクトの実行時における実際のクラスを調べるときに使用します。次の式は,

object `is` class

object が class で表されるクラスまたはその派生クラスのインスタンスであれば True を返し，そうでない場合は False を返します（object が **nil** の場合は False を返します)。object の宣言時の型が class と無関係である場合，つまり，両者の型が異なり，一方の型が他方の上位型でもない場合は，コンパイルエラーになります。次に例を示します。

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

この文では，ActiveControl 変数が参照しているオブジェクトが TEdit またはその派生クラスのインスタンスかどうかをチェックした後，この変数を TEdit にキャストしています。

as 演算子

as 演算子はチェック付きの型キャストを行います。次の式は,

object `as` class

object と同じオブジェクトへの参照を返しますが，型は class によって指定された型になります。実行時には，object は class で表されるクラスまたはその派生クラスのインスタンスか **nil** でなければなりません。そうでない場合は，例外が生成されます。object の宣言時の型が class と無関係である場合，つまり，両者の型が異なり，一方の型が他方の上位型でもない場合は，コンパイルエラーになります。次に例を示します。

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

演算子の優先順位の規則により，**as** による型キャストをカッコで囲まなければならない場合があります。次に例を示します。

```
(Sender as TButton).Caption := '&Ok';
```

クラスメソッド

クラスメソッドとは，オブジェクトではなくクラスを操作するメソッド（コンストラクタ以外）です。クラスメソッドを定義するときは，予約語 **class** を先頭に付けなければなりません。次に例を示します。

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ⋮
  end;
```

クラスメソッドの定義宣言にも，先頭に **class** を付けなければなりません。次に例を示します。

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  ⋮
end;
```

クラスメソッドの定義宣言では，Self 識別子はクラスメソッドを呼び出すクラスか，クラスメソッドが定義されているクラスの派生クラスを表します。クラスメソッドがクラス C で呼び出された場合は，Self の型は `class of C` になります。したがって，Self を使ってフィールド，プロパティ，および通常のメソッド（オブジェクトメソッド）にアクセスすることはできませんが，コンストラクタやその他のクラスメソッドを呼び出すことは可能です。

クラスメソッドはクラス参照またはオブジェクト参照によって呼び出せます。クラスメソッドがオブジェクト参照によって呼び出された場合は，そのオブジェクトのクラスが Self の値になります。

例外

例外は，エラーやその他のイベントによりプログラムの通常の実行が中断された場合に生成されます。例外が生成されると例外ハンドラに制御が移ります。例外ハンドラを使用することで，通常のプログラムロジックとエラー処理を分離できます。例外はオブジェクトであるため，継承により例外を階層構造に分類することができ，また，既存のコードに影響を与えることなく新しい例外を追加することができます。例外を利用することで，例外が生成された位置から例外が処理される位置まで，エラーメッセージなどの情報を運ぶことができます。

アプリケーションで SysUtils ユニットを使用すれば，すべての実行時エラーが自動的に例外に変換されます。メモリ不足，ゼロによる除算，一般保護例外など，SysUtils ユニットを使用しなければアプリケーションを終了させてしまうようなエラーは，SysUtils ユニットを使用することで捕捉して処理することが可能になります。

例外処理の使用

例外は，プログラムを停止せずに，煩雑な条件文を使用しないで実行時エラーをトラップできる優れた方法を提供します。しかし，Object Pascal の例外処理メカニズムはその複雑さゆえに効率が低下し

てしまうので，適切な判断によって使用しなければなりません。例外はどのような問題に対しても生成可能であり，**try...except** または **try...finally** 文に含めることによってほとんどすべてのコードのブロックを保護することができますが，実際には特殊な場面で使用されます。

例外処理は，エラーの発生率が低いか，または発生率の査定が困難であるにも関わらず，そのエラーがアプリケーションのクラッシュなど致命的な障害を引き起こすような場合や，エラー条件を **if...then** 文でテストするのが複雑または困難であるような場合に最適です。また，オペレーティングシステムによる例外や，制御できないソースコードで書かれたルーチンによって生成された例外に対して応答しなければならない場合にも適しています。一般的に，例外はハードウェア，メモリ，I/O，およびオペレーティングシステムのエラーに使用されます。

多くの場合，条件文を使用するのがエラーをテストする最良の方法です。たとえば，ファイルを開く前に，そのファイルが存在することを確認したい場合は，次のように行います。

```
try
  AssignFile(F, FileName);
  Reset(F);  // ファイルが見つからない場合は EInOutError 例外を生成する
except
  on Exception do ...
end;
```

下記の条件文では例外を生成せずに処理します。

```
if FileExists(FileName) then  // ファイルが見つからない場合は False を返す . 例外は生成しない
begin
  AssignFile(F, FileName);
  Reset(F);
end;
```

別の方法として，Assert 手続きを使用してソースコードの任意の場所でブール論理条件をテストすることもできます。Assert 文が失敗するとプログラムは停止しますが，SysUtils ユニットを使用している場合は EAssertionFailed 例外が生成されます。Assert 手続きを使用するのは，発生してほしくない条件をテストする場合のみです。詳細については，標準手続き Assert に関するオンラインヘルプを参照してください。

例外型の宣言

例外型は，ほかのクラスとまったく同じように宣言します。実際には，どのようなクラスのインスタンスでも例外オブジェクトとして使用できますが，例外は SysUtils で定義されている Exception から派生させるようにすることをお勧めします。

例外は，継承を利用することで関連するグループに分けることができます。たとえば，SysUtils に含まれる次の宣言では，数値演算エラーを扱う例外型のグループが定義されています。

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

EMathError の例外ハンドラを 1 つ定義し，その中で EInvalidOp，EZeroDivide，Eoverflow，および Eunderflow も処理することができます。

例外クラスの中で，エラーに関する付加情報を提供するフィールド，メソッド，プロパティを定義することもあります。次に例を示します。

```
type EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

例外の生成と処理

例外オブジェクトを作成するには，**raise** 文で例外クラスのコンストラクタを呼び出します。次に例を示します。

```
raise EMathError.Create;
```

通常，**raise** 文の形式は次のようになります。

raise object at address

object と at address はどちらもオプションです。object を指定しない場合は，現在の例外が再生成されます。7-31 ページの「例外の再生成」を参照してください。address を指定する場合は，通常は手続きまたは関数へのポインタを指定します。エラーが実際に発生した位置よりスタック上の前の位置から例外を生成したい場合にこの方法を使います。

例外が生成，つまり，**raise** 文で例外が参照されると，例外は特別な例外処理ロジックの管理下に入ります。**raise** 文は，通常の文のようには制御を返さず，当該クラスの例外を処理可能な最も内側の例外ハンドラに制御を移します（最も内側の例外ハンドラとは，最後に入って，まだ抜け出していない **try...except** ブロックです）。

次に示すのは文字列を整数に変換する関数ですが，変換結果の値が指定範囲外の場合は ERangeError 例外を生成します。

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);  // StrToInt は SysUtils で定義されている
  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 メソッドが **raise** 文の中で呼び出されている点に注意してください。例外とその派生クラスには，これ以外にも例外メッセージとコンテキスト ID を作成する特別なコンストラクタがあります。詳細は，オンラインヘルプを参照してください。

生成された例外は，処理された後，自動的に破棄されます。生成された例外を手動で破棄することは避けてください。

メモ ユニットの初期化部で例外を生成させると，意図した結果が得られないことあります。例外サポートは SysUtils ユニットで定義されていますが，SysUtils ユニットの初期化後でないと例外サポートは有効になりません。初期化中に例外が発生すると，SysUtils を含め，初期化されたすべてのユニットの終了処理が行われ，同じ例外が再生成されます。その後，通常はプログラムの実行を中断することにより，その例外が捕捉され，処理されます。

try..except 文

例外は **try...except** 文の内部で処理されます。次に例を示します。

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

この文では Y を Z で割りますが，EZeroDivide 例外が生成されると HandleZeroDivide というルーチンが呼び出されます。

try...except 文の構文を次に示します。

`try` statements `except` exceptionBlock `end`

statements には文をセミコロンで区切って指定し，exceptionBlock には次のいずれかを指定します。

- 別の文
- 例外ハンドラ。必要に応じて次のような **else** 節を指定

  `else` statements

例外ハンドラは，次のような形式をとります。

`on` identifier: type `do` statement

identifier: はオプションで，指定する場合は任意の有効な識別子を指定できます。type には例外を表す型を指定し，statement には任意の文を指定します。

try...except 文では，まず，最初の statements で指定された文が実行されます。例外が生成されなかった場合は例外ブロック（exceptionBlock）は無視され，プログラムの次の部分に制御が渡されます。

最初の statements で指定された文の実行中に例外が生成されると，statements に記述された **raise** 文か，statements から呼び出される手続きまたは関数により，以下のように例外の処理が試みられます。

- 例外に一致するハンドラが例外ブロック内にあれば，例外に一致する最初のハンドラに制御が渡されます。例外ハンドラが例外に「一致」するのは，例外ハンドラで指定された type がその例外のクラスかその上位クラスである場合です。
- 例外に一致するハンドラが見つからない場合は，**else** 節が存在すれば，**else** 節の statement に制御が移されます。
- 例外ブロックに例外ハンドラが存在せず，文だけが記述されている場合は，その最初の文に制御が移されます。

以上の条件がいずれも満たされない場合は，1 つ前に入り，まだ抜け出していない **try...except** 文の例外ブロックで検索が続行されます。該当するハンドラ，**else** 句，または文リストがそこでも見つからない場合は，さらにその 1 つ前に入った **try...except** 文に移り，検索が続行されます。最も外側の **try...except** 文に到達し，そこでも例外が処理されなかった場合は，プログラムの実行が終了します。

例外が処理される場合は，例外処理が行われる **try...except** 文が含まれる手続きまたは関数にスタックをさかのぼる形で戻り，実行される例外ハンドラ，**else** 節，文リストのいずれかに制御が渡されます。このプロセスでは，例外処理が行われる **try...except** 文に入った後に行われたすべての手続き呼び出しと関数呼び出しは破棄されます。次に，Destroy デストラクタの呼び出しにより例外オブジェ

クトは自動的に破棄され，**try...except** 文の次の文に制御が渡されます。なお，Exit，Break，Continue のいずれかの標準手続きの呼び出しにより制御が例外ハンドラ内に残された場合であっても，例外オブジェクトは自動的に破棄されます。

下の例では，最初の例外ハンドラがゼロ除算例外を処理し，2 番目の例外ハンドラがオーバーフロー例外を処理し，最後の例外ハンドラがその他のすべての数値演算例外を処理します。EMathError は例外ブロックの最後に記述されていますが，これは，EMathError がほかの 2 つの例外クラスの上位クラスだからです。もし EMathError を先頭に記述すると，ほかの 2 つのハンドラは呼び出されなくなってしまいます。

```
try
 ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

例外ハンドラでは，例外クラス名の前に識別子を指定できます。例外クラス名の前に識別子を指定すると，**on...do** の次の文の実行中，この識別子は例外オブジェクトを表すようになります。この識別子のスコープは **on...do** の次の文に限定されます。次に例を示します。

```
try
 ⋮
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

例外ブロックに **else** 節を指定すると，例外ブロックの例外ハンドラで処理できない例外を **else** 節で処理できます。次に例を示します。

```
try
 ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

この **else** 節では，EMathError 以外の例外が処理されます。

例外ハンドラをまったく含まず，文のリストだけで構成される例外ブロックでは，すべての例外が処理されます。次に例を示します。

```
try
 ⋮
except
  HandleException;
end;
```

この例では，**try** から **except** の間にある文を実行した結果発生した例外は，HandleException ルーチンによって処理されます。

例外の再生成

オブジェクト参照を指定せずに予約語 **raise** だけを単独で例外ブロック内に記述すると，その例外ブロックで処理される例外を生成することができます。こうすることにより，例外ハンドラでエラーへの対処を限定的に行った後，例外を再生成することができます。例外の発生後，手続きまたは関数でクリーンアップ処理を行わなければならないが，その手続きまたは関数では例外を完全には処理できないといった場合に，例外を再生成すると便利です。

たとえば，次に示す GetFileList 関数（TStringList オブジェクトを割り当て，指定の検索パスと一致するファイル名をそのオブジェクトに入れる関数）で考えてみます。

```
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 は TStringList オブジェクトを作成した後，FindFirst 関数と FindNext 関数（SysUtils で定義）を使用してこのオブジェクトを初期化します。検索パスが無効である，文字列リストに入れるだけの十分なメモリがない，などの理由で文字列リストの初期化に失敗した場合は，新しく割り当てた文字列リストを破棄する必要があります。呼び出し元がまだその文字列リストの存在を認識していないからです。そのため，この文字列リストの初期化は **try...except** 文の中で実行します。例外が発生した場合は，**try...except** 文の例外ブロックによりその文字列が破棄され，同じ例外が再生成されます。

例外のネスト

例外ハンドラで実行されるコードは，それ自体が例外を生成して処理することができます。例外ハンドラで生成される例外も例外ハンドラの内部で処理される限り，元の例外が影響を受けることはありません。ただし，例外ハンドラで生成された例外がそのハンドラを越えて伝わった場合は，元のハンドラが失われます。この様子を次の Tan 関数で示します。

```
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 の実行中に EMathError 例外が発生すると，例外ハンドラにより ETrigError が生成されます。Tan には ETrigError を処理するハンドラがないため，ETrigError 例外は元の例外ハンドラを越えて伝わり，その結果 EMathError 例外は破棄されます。呼び出し元には，Tan 関数が ETrigError 例外を生成したように見えます。

try..finally 文

ある処理を行ったら，処理が例外によって中断されたかどうかにかかわらず，処理の特定の部分を確実に完了させたい場合があります。たとえば，あるルーチンで特定のリソースの制御権を得た場合，そのルーチンが正常に終了したかどうかにかかわらず，たいていはそのリソースを解放する必要があります。このような場合は，**try...finally** 文を使用します。

次に示すのは，ファイルを開いてそれを処理するコードの例ですが，このコードでは，実行中にエラーが発生した場合でもファイルを最終的に確実に閉じることができます。

```
Reset(F);
try
 ⋮ // ファイル F を処理
finally
  CloseFile(F);
end;
```

try...finally 文の構文を次に示します。

try $statementList_1$ finally $statementList_2$ end

statementList には文をセミコロンで区切って指定します。**try...finally** 文は，$statementList_1$（**try** 節）に指定された文を実行します。$statementList_1$ の実行中に例外が生成されなかった場合は，$statementList_2$（**finally** 節）が実行されます。$statementList_1$ の実行中に例外が生成された場合は，$statementList_2$ に制御が移り，$statementList_2$ がその実行を終えたら，例外が再生成されます。Exit，Break，Continue のいずれかの手続きの呼び出しにより制御が $statementList_1$ 内に残された場合は，$statementList_2$ が自動的に実行されます。このように，**try** 節がどのような状態で終了したかに関係なく，**finally** 節は常に実行されます。

生成された例外が **finally** で処理されない場合，その例外は **try...finally** 文を越えて伝わり，**try** 節で既に生成されている例外は失われます。したがって，ほかの例外の伝播の妨げにならないよう，ローカルに生成された例外はすべて **finally** 節で処理するようにしてください。

標準の例外クラスと例外ルーチン

SysUtils ユニットでは，ExceptObject，ExceptAddr，ShowException など，例外を処理するさまざまな標準ルーチンが宣言されています。また，SysUtils などのユニットには，たくさんの例外クラスが含まれています。これらの例外クラスは，OutlineError を除いてすべて Exception からの派生クラスです。

Exception クラスには Message というプロパティと HelpContext というプロパティがあり，これらのプロパティを使うと，エラーの説明と状況感知型オンラインヘルプのコンテキスト ID を渡すことができます。また，各種の方法でエラーの説明とコンテキスト ID を指定できるコンストラクタメソッドも多数定義されています。詳細は，オンラインヘルプを参照してください。

第 8 章

標準ルーチンと入出力

この章では，テキストおよびファイルの入出力について説明します。また，標準ライブラリルーチンについてもまとめてあります。この章で取り上げる手続きと関数の多くは System ユニットに定義されているものです。System ユニットは，暗黙のうちに必ずアプリケーションに組み込まれてコンパイルされます。その他の手続きと関数はコンパイラに組み込まれていますが，その扱いは System ユニットに定義されているものと同様です。

標準ルーチンの中には，SysUtils などのユニットに入っているルーチンがありますが，このようなルーチンを使う場合は，ユニット名を **uses** 節に指定しなければなりません。ただし，**uses** 節に System を含めることはできません。また，System ユニットの内容を変更したり，明示的に再構築してはいけません。

この章で取り上げるルーチンの詳細については，オンラインヘルプを参照してください。

ファイルの入出力

次の表は，入出力関連のルーチンの一覧です。

表 8.1　入出力手続きと関数

手続き / 関数	説明
Append	追加のために既存のファイルを開く
AssignFile	外部ファイルの名前をファイル変数に割り当てる
BlockRead	型なしファイルからレコードを読み出す
BlockWrite	型なしファイルにレコードを書き込む
ChDir	カレントディレクトリを変更する
CloseFile	開いているファイルを閉じる
Eof	ファイルについてファイルの終わりの状態を返す
Eoln	テキストファイルについて行の終わりの状態を返す
Erase	外部ファイルを消去する
FilePos	型付きファイルまたは型なしファイルの現在のファイル位置を返す

表 8.1　入出力手続きと関数（つづき）

手続き / 関数	説明
FileSize	ファイルの現在のサイズを返す（テキストファイルには使えない）
Flush	出力テキストファイルのバッファをフラッシュする
GetDir	指定されたドライブのカレントディレクトリを返す
IOResult	最後に実行された入出力関数の状態を表す整数値を返す
MkDir	サブディレクトリを作成する
Read	ファイルから値を読み出して変数に格納する
Readln	Read と同じ処理をして，テキストファイルの次の行の先頭に移動する
Rename	外部ファイルの名前を変更する
Reset	既存のファイルを開く
Rewrite	新しいファイルを作成して開く
RmDir	空のサブディレクトリを削除する
Seek	型付きファイルまたは型なしファイルの現在のファイル位置を指定された要素に移動する（テキストファイルには使えない）
SeekEof	テキストファイルについてファイルの終わりの状態を返す
SeekEoln	テキストファイルについて行の終わりの状態を返す
SetTextBuf	入出力バッファをテキストファイルに割り当てる
Truncate	型付きファイルまたは型なしファイルを現在のファイル位置で切り捨てる
Write	値をファイルに書き込む
Writeln	Write と同じ処理をして，行の終わりを表すマーカーをテキストファイルに書き込む

ファイル変数はファイル型の変数です。ファイルは，型付き，テキスト，型なしの 3 種類に分類されます。ファイル型の宣言構文については，5-24 ページの「ファイル型」を参照してください。

ファイル変数を使うには，AssignFile 手続きを呼び出して外部ファイルにファイル変数を関連付けなければなりません。外部ファイルは通常は名前付きのディスクファイルですが，キーボードやディスプレイなどのデバイスである場合もあります。外部ファイルは書き込まれた情報を保存したり，読み出される情報を提供したりします。

外部ファイルに関連付けたファイル変数は，入出力が行えるように「開く」必要があります。既存のファイルは Reset 手続きで開きます。新しいファイルは Rewrite 手続きで作成して開きます。Reset で開いたテキストファイルは読み出し専用で，Rewrite や Append で開いたテキストファイルは書き込み専用です。型付きファイルと型なしファイルは，Reset と Rewrite のどちらで開いた場合でも常に読み出しと書き込みの両方ができます。

どのファイルも線形に並んだ要素のシーケンスで，各要素にファイルの要素型（レコード型）が割り当てられています。この要素にはゼロから始まる番号が付けられます。

ファイルのアクセスは通常はシーケンシャルアクセスです。つまり，要素を標準手続き Read によって読み出したり標準手続き Write によって書き込むと，現在のファイル位置がファイル要素の番号順に移動します。ただし，型付きファイルと型なしファイルの場合は標準手続き Seek によるランダムアクセスもできます。Seek は指定された要素に現在のファイル位置を移動します。標準関数の FilePos と FileSize を使うと，現在のファイル位置とファイルサイズがわかります。

プログラムのファイル処理が終わったら，標準手続き CloseFile によってファイルを閉じなければなりません。ファイルが閉じると，関連付けられている外部ファイルが更新されます。その後で，ファイル変数はほかの外部ファイルに関連付けることができます。

デフォルトでは，入出力に関する標準の手続きや関数のすべての呼び出しで，エラーが自動的にチェックされます。エラーが発生すると例外が生成されます（例外処理が有効になっていない場合はプログラムが終了します)。この自動チェックはコンパイラ指令の **{$I+}** と **{$I-}** で有効または無効にできます。入出力チェックが無効な場合，つまり手続きまたは関数の呼び出しを **{$I-}** 状態でコンパイルした場合，入出力エラーが発生しても例外は生成されません。そのため，入出力操作の結果を確認するには標準関数 IOResult を呼び出す必要があります。

エラーが発生したときは，関係のないエラーであっても，必ず IOResult 関数を呼び出してエラーをクリアしなければなりません。そうしないと，**{$I+}** が現在の状態の場合，IOResult のエラーが残るため，次の入出力関数の呼び出しは失敗します。

テキストファイル

この節では，標準の型である Text 型のファイル変数を使った入出力について概説します。

テキストファイルが開くと，その外部ファイルは特別な方法で解釈されます。つまり，テキストファイルは行単位で構成された文字の並びを表し，各行の最後に行の終わりを表すマーカー（復帰文字（CR）の後に改行文字（LF）が続くことが多い）があるものとみなされます。Text 型は `file of Char` 型と区別されます。

テキストファイルに対して特別な形式の Read や Write を使うと，Char 型でない値の読み書きができます。このような値が文字型の表現に，または文字型の表現がこのような値に自動的に変換されます。たとえば，Integer 型の変数 I があるときに `Read (F, I)` とすると，数字の並びが読み出されて 10 進整数と解釈され，I に格納されます。

標準のテキストファイル変数には，Input と Output の 2 つがあります。標準のファイル変数 Input は，オペレーティングシステムの標準入力（通常はキーボード）に関連付けられた読み出し専用ファイルです。標準のファイル変数 Output はオペレーティングシステムの標準出力（通常はディスプレイ）に関連付けられた書き込み専用ファイルです。アプリケーションを実行するとその前に，次の文が実行されたものとして Input と Output は常に自動的に開かれます。

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

メモ テキスト用の入出力はコンソールアプリケーションでのみ使用可能です。つまり，コンソールアプリケーションは，［プロジェクトオプション］ダイアログボックスの［リンカ］ページで［コンソールアプリケーションの作成］をチェックするか，**-cc** コマンドラインコンパイラオプションを付けてコンパイルして作成します。GUI（非コンソール）アプリケーションでは，Input と Output を使った読み書きは入出力エラーになります。

テキストファイル用の標準の入出力ルーチンには，ファイル変数をパラメータとして明示的に指定しなくてもよいものがあります。ファイルのパラメータを省略すると，手続きまたは関数が入力用か出

力用かによって，Input または Output がデフォルトで想定されます。たとえば，`Read(X)` は `Read(Input, X)` に相当し，`Write(X)` は `Write(Output, X)` に相当します。

テキストファイル用の入出力ルーチンのいずれかを呼び出すときにファイルを指定する場合，AssignFile を使ってファイルを外部ファイルに関連付け，Reset，Rewrite，Append のいずれかを使ってそのファイルを開かなければなりません。Reset で開いたファイルを出力用の手続きまたは関数に渡すと例外が生成されます。また，Rewrite または Append で開いたファイルを入力用の手続きまたは関数に渡した場合にも例外が生成されます。

型なしファイル

型なしファイルは低レベル入出力チャネルで，主に型や構造を考慮せずにディスクファイルに直接アクセスするのに利用されます。型なしファイルは予約語 **file** だけで宣言します。次に例を示します。

```
var DataFile: file;
```

型なしファイルに対して Reset 手続きと Rewrite 手続きを使うと，追加パラメータによりデータ転送に使うレコードサイズを指定できます。歴史的理由によってデフォルトのレコードサイズは 128 バイトです。レコードサイズを 1 にすることが，ファイルの正確なサイズを正しく反映する唯一の方法です。レコードサイズが 1 の場合，部分的なレコードは存在しません。

Read と Write を除き，型付きファイルに使えるすべての標準手続きと標準関数は型なしファイルにも使えます。Read と Write のかわりに，BlockRead と BlockWrite という 2 つの手続きを高速データ転送に使います。

テキストファイルデバイスドライバ

プログラム用に独自のテキストファイルデバイスドライバを定義できます。テキストファイルデバイスドライバは 4 つの関数からなり，この 4 つの関数が Object Pascal のファイルシステムとデバイスとの間のインターフェースを完全に実現します。

各デバイスドライバを定義する 4 つの関数とは Open，InOut，Flush，Close です。各関数のヘッダーを次に示します。

```
function DeviceFunc(var F: TTextRec): Integer;
```

ここで，DeviceFunc は関数（つまり Open, InOut, Flush, または Close) の名前となります。TtextRec 型についての詳細は，オンラインヘルプを参照してください。デバイスインターフェース関数の戻り値は IOResult が返す値になります。操作に成功した場合，戻り値は 0 になります。

デバイスインターフェース関数を特定のファイルに関連付けるには，独自の Assign 手続きを作成しなければなりません。この Assign 手続きは 4 つのデバイスインターフェース関数のアドレスをテキストファイル変数の 4 つの関数ポインタに代入しなければなりません。さらに，「マジック」定数 fmClosed を Mode フィールドに格納し，テキストファイルのバッファのサイズを BufSize に，テキストファイルのバッファへのポインタを BufPtr に格納して，Name の文字列をクリアします。

たとえば，4 つのデバイスインターフェース関数の名前を DevOpen，DevInOut，DevFlush，DevClose とすると，Assign 手続きは次のようになります。

```
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;
```

デバイスインターフェース関数は，ファイルレコードの UserData フィールドを使って専用の情報を格納できます。このフィールドは Borland 開発ツールのファイルシステムによって変更されることはありません。

デバイス関連関数

テキストファイルデバイスドライバを構成する関数を次に示します。

Open 関数

Open 関数はデバイスに関連付けられたテキストファイルを開くときに標準手続きの Reset，Rewrite，Append によって呼び出されます。Open 関数が呼び出された時点で Mode フィールドには fmInput，fmOutput，fmInOut のいずれかが入っており，Open 関数が Reset，Rewrite，Append のどれから呼び出されたかを示します。

Open 関数は Mode の値に従って入力用または出力用にファイルを用意します。Mode の指定が fmInOut の場合（Open が Append から呼び出されたことを示します），Open が戻る前に Mode を fmOutput に変えなければなりません。

Open は必ずほかのデバイスインターフェース関数より先に呼び出されます。したがって，AssignDev は OpenFunc フィールドだけを初期化し，残りのベクタの初期化を Open に任せることができます。Mode に基づき，Open は入力用または出力用の関数へのポインタを設定できます。これにより，InOut 関数，Flush 関数，CloseFile 手続きが現在のモードを決めなくてもすみます。

InOut 関数

InOut 関数は，デバイスで入出力が必要なときに Read，Readln，Write，Writeln，Eof，Eoln，SeekEof，SeekEoln，CloseFile といった標準ルーチンから呼び出されます。

Mode が fmInput の場合，InOut 関数は最大 BufSize 個の文字を `BufPtr^` に読み込み，読み込んだ文字の数を BufEnd に返します。BufPos には 0 を格納します。入力要求の結果として InOut 関数が BufEnd に 0 を返した場合，そのファイルに対して Eof は True になります。

Mode が fmOutput の場合，InOut 関数は `BufPtr^` から BufSize 個の文字を書き出し，BufPos には 0 を返します。

Flush 関数

Flush 関数は Read，Readln，Write，Writeln のそれぞれの処理の終わりに呼び出されます。Flush でテキストファイルのバッファをフラッシュできます。

Mode が fmInput の場合，Flush 関数は BufPos と BufEnd に 0 を格納して，バッファに残っている（読まれていない）文字を消去できます。この機能を使うことはあまりありません。

Mode が fmOutput の場合，Flush 関数は InOut 関数と同じようにバッファの内容を書き出すことができます。これにより，デバイスに書き出されたテキストがすぐにデバイスに表示されることが保証されます。Flush が何もしない場合，バッファがいっぱいになるかファイルが閉じるまでテキストはデバイスに表示されません。

Close 関数

Close 関数はデバイスに関連付けられたテキストファイルを閉じるときに標準手続きの CloseFile によって呼び出されます。Reset，Rewrite，Append の各手続きも，開こうとしたファイルがすでに開いている場合は Close を呼び出します。Mode が fmOutput の場合，Close を呼び出す前に Object Pascal のファイルシステムは InOut 関数を呼び出し，すべての文字がデバイスに書き出されたことを保証します。

ヌルで終わる文字列

Object Pascal の拡張構文により，添字がゼロから始まる文字配列に対して標準手続きの Read, Readln，Str，Val が使え，添字がゼロから始まる文字配列と文字ポインタの両方に対して標準手続きの Write，Writeln，Val，AssignFile，Rename が使えます。また，ヌルで終わる文字列は，次に示す関数を使って処理できます。ヌルで終わる文字列についての詳細は，5-13 ページの「ヌルで終わる文字列の処理」を参照してください。

表 8.2　ヌルで終わる文字列の関数

関数	説明
StrAlloc	与えられたサイズの文字バッファをヒープ上に割り当てる
StrBufSize	StrAlloc または StrNew を使って割り当てた文字バッファのサイズを返す
StrCat	2 つの文字列を結合する
StrComp	2 つの文字列を比較する
StrCopy	文字列をコピーする
StrDispose	StrAlloc または StrNew を使って割り当てた文字バッファを破棄する
StrECopy	文字列をコピーし，その文字列の終わりへのポインタを返す
StrEnd	文字列の終わりへのポインタを返す
StrFmt	1 つまたは複数の値を書式化された文字列に変換する
StrIComp	大文字と小文字を区別せずに 2 つの文字列を比較する
StrLCat	指定された長さを上限として 2 つの文字列を結合する
StrLComp	指定された長さを上限として 2 つの文字列を比較する
StrLCopy	指定された長さを上限として文字列をコピーする
StrLen	文字列の長さを返す
StrLFmt	指定された長さを上限として，1 つまたは複数の値を書式化された文字列に変換する

表 8.2　ヌルで終わる文字列の関数（つづき）

関数	説明
StrLIComp	指定された長さを上限として，大文字と小文字を区別せずに 2 つの文字列を比較する
StrLower	文字列を小文字に変換する
StrMove	文字のブロックを 1 つの文字列から別の文字列へ移動する
StrNew	ヒープ上に文字列を割り当てる
StrPCopy	Pascal 文字列をヌルで終わる文字列へコピーする
StrPLCopy	指定された長さを上限として，Pascal 文字列をヌルで終わる文字列へコピーする
StrPos	与えられた部分文字列が文字列内で最初に出現する位置へのポインタを返す
StrRScan	与えられた文字が文字列内で最後に出現する位置へのポインタを返す
StrScan	与えられた文字が文字列内で最初に出現する位置へのポインタを返す
StrUpper	文字列を大文字に変換する

標準の各文字列処理関数には，地域固有の文字の並び順に対応したマルチバイト版の関数も用意されています。マルチバイト版の関数名は，Ansi- で始まります。たとえば，StfPos のマルチバイト版は AnsiStrPos になります。マルチバイト文字のサポートは，オペレーティングシステムに依存し，現在のロケールによって決まります。

ワイド文字列

System ユニットには WideCharToString，WideCharLenToString，StringToWideChar という 3 つの関数があり，ヌルで終わるワイド文字列を 1 バイトまたは 2 バイトの長い文字列に変換できるようになっています。

ワイド文字列についての詳細は，5-12 ページの「拡張文字セットについて」を参照してください。

その他の標準ルーチン

次の表に，Borland 開発ツールのライブラリの中にあるうち，使用頻度が高い手続きと関数を示します。この一覧は，標準ルーチンの全部ではありません。ここに示したものも含め，各ルーチンの詳細については，オンラインヘルプを参照してください。

表 8.3　その他の標準ルーチン

手続き / 関数	説明
Abort	エラーを報告せずに処理を終了する
Addr	指定したオブジェクトへのポインタを返す
AllocMem	メモリブロックを割り当て，各バイトをゼロに初期化する
ArcTan	指定した数のアークタンジェント（逆正接）を算出する
Assert	論理式が True かどうか評価する
Assigned	**nil**（未代入）ポインタまたは手続き変数かどうかを評価する
Beep	コンピュータのスピーカを使って，標準の警告音（ビープ）を鳴らす
Break	**for**，**while**，または **repeat** 文を終了させる
ByteToCharIndex	文字列の中における，指定したバイトを含む文字の位置を返す

表 8.3　その他の標準ルーチン（つづき）

手続き / 関数	説明
Chr	指定した値に対応する文字を返す
Close	ファイル変数と外部ファイルの関連付けを解除する
CompareMem	2 つのメモリイメージをバイナリレベルで比較する
CompareStr	大文字と小文字を区別して文字列を比較する
CompareText	大文字と小文字を区別しないで文字列を比較する
Continue	**for**，**while**，または **repeat** 文の次の繰り返し処理に制御を移行させる
Copy	文字列または動的配列の一部を返す
Cos	角のコサイン（余弦）を算出する
CurrToStr	通貨型の変数を文字列に変換する
Date	現在の日付を返す
DateTimeToStr	TDateTime 型の変数を文字列に変換する
DateToStr	TDateTime 型の変数を文字列に変換する
Dec	順序型変数の値をデクリメントする
Dispose	動的変数に割り当てられていたメモリを解放する
ExceptAddr	現在の例外が発生したアドレスを返す
Exit	現在の手続きを終了する
Exp	X の指数を算出する
FillChar	連続したバイトを指定の値で埋める
Finalize	動的に割り当てた変数の初期化を解除する
FloatToStr	浮動小数点数値を文字列に変換する
FloatToStrF	浮動小数点数値を，指定の書式で文字列に変換する
FmtLoadStr	リソース書式文字列を使って書式付きの出力を返す
FmtStr	一連の配列から書式付き文字列を構築する
Format	書式文字列および一連の配列から文字列を構築する
FormatDateTime	日付 / 時刻値の表示形式を設定する
FormatFloat	浮動小数点数値の表示形式を設定する
FreeMem	動的変数を破棄する
GetMem	動的変数，およびそのブロックのアドレスへのポインタを作成する
GetParentForm	指定したコントロールが属しているフォームまたはプロパティページを返す
Halt	プログラムを異常終了させる
Hi	式の上位バイトを符号なしの値として返す
High	型，配列，文字列の範囲内で最上位の値を返す
Inc	順序型変数の値をインクリメントする
Initialize	動的に割り当てられた変数を初期化する
Insert	部分文字列を文字列中の指定した場所に挿入する
Int	実数の整数部分を返す
IntToStr	整数を文字列に変換する
Length	文字列または配列の長さを返す
Lo	式の下位バイトを符号なしの値として返す
Low	型，配列，文字列の範囲内で最下位の値を返す
LowerCase	ASCII 文字列を小文字に変換する
MaxIntValue	整数の配列の中で最大の符号付き値を返す

表 8.3　その他の標準ルーチン（つづき）

手続き / 関数	説明
MaxValue	配列の中で最大の符号付き値を返す
MinIntValue	整数の配列の中で最小の符号付き値を返す
MinValue	配列の中で最小の符号付き値を返す
New	動的変数を新規作成し，指定したポインタがその変数を指すようにする
Now	現在の日付 / 時刻を返す
Ord	順序型の式の順序値を返す
Pos	文字列の中における，指定した部分文字列の 1 文字目の位置を返す
Pred	順序値の前の値を返す
Ptr	指定したアドレスをポインタに変換する
Random	指定した範囲内で乱数を発生させる
ReallocMem	動的変数の再割り当てを行う
Round	実数値を一番近い整数値に変換して返す
SetLength	文字列型変数か文字列型配列の動的長さを設定する
SetString	与えられた文字列の内容と長さを設定する
ShowException	例外メッセージをアドレスとともに表示する
ShowMessage	書式なし文字列と［OK］ボタンを含むメッセージボックスを表示する
ShowMessageFmt	書式付き文字列と［OK］ボタンを含むメッセージボックスを表示する
Sin	角のサイン（正弦）をラジアンで返す
SizeOf	変数または型が占めるバイト数を返す
Sqr	指定した数の 2 乗を返す
Sqrt	指定した数の平方根を返す
Str	文字列の表示形式を設定して変数に返す
StrToCurr	文字列を通貨型の値に変換する
StrToDate	文字列を日付形式（TDateTime）に変換する
StrToDateTime	文字列を TDateTime に変換する
StrToFloat	文字列を浮動小数点数値に変換する
StrToInt	文字列を整数に変換する
StrToTime	文字列を時刻形式に変換する（TDateTime）
StrUpper	文字列を大文字に変換して返す
Succ	順序値の次の値を返す
Sum	配列中の要素の合計を返す
Time	現在の時刻を返す
TimeToStr	TDateTime 型の変数を文字列に変換する
Trunc	実数を切り捨てて整数にする
UniqueString	文字列の参照が 1 つだけであるようにする（その文字列をコピーして単一の参照を作成できる）
UpCase	文字を大文字に変換する
UpperCase	文字列を大文字に変換して返す
VarArrayCreate	バリアント配列を作成する
VarArrayDimCount	バリアント配列の次元数を返す
VarARrayHighBound	バリアント配列の次元の上限を返す
VarArrayLock	バリアント配列をロックし，データへのポインタを返す

表 8.3　その他の標準ルーチン（つづき）

手続き / 関数	説明
VarArrayLowBound	バリアント配列の次元の下限を返す
VarArrayOf	1 次元のバリアント配列を作成し，値を入れる
VarArrayRedim	バリアント配列のサイズを変更する
VarArrayRef	渡されたバリアント配列への参照を返す
VarArrayUnlock	バリアント配列のロックを解除する
VarAsType	指定した型にバリアントを変換する
VarCast	指定した型にバリアントを変換し，結果を変数に格納する
VarClear	バリアントをクリアする
VarCopy	バリアントをコピーする
VarToStr	バリアントを文字列に変換する
VarType	指定したバリアントの型コードを返す

書式文字列についての詳細は，オンラインヘルプの「形式文字列」を参照してください。

第 II 部

上級機能

第 II 部の各章では，言語の専門的な機能と上級プログラマ向けの事項について説明しています。各章は次の通りです。

- 第 9 章「ライブラリとパッケージ」
- 第 10 章「オブジェクトインターフェース」
- 第 11 章「メモリ管理」
- 第 12 章「プログラムの制御」
- 第 13 章「インラインアセンブラコード」

第9章

ライブラリとパッケージ

動的にロード可能なライブラリは，Windows ではダイナミックリンクライブラリ（DLL），Linux では共有オブジェクトライブラリです。これは，アプリケーションから，または他の DLL あるいは共有オブジェクトから呼び出せるルーチンのコレクションです。ユニットと同じように，動的にロード可能なライブラリには共有可能なコードまたはリソースが入っています。ただし，この形式のライブラリはプログラムとは別にコンパイルされる実行形式ファイルで，ライブラリを使用するプログラムに実行時にリンクされます。

単独で機能する通常の実行可能ファイルと区別するため，Windows ではコンパイル済み DLL のファイルの拡張子は .DLL です。Linux では，共有オブジェクトファイルの拡張子は .so です。Object Pascal プログラムは他の言語で記述された DLL または共有オブジェクトを呼び出すこともでき，Object Pascal で作成した DLL または共有オブジェクトを他の言語で記述したアプリケーションから呼び出して使うこともできます。

動的にロード可能なライブラリの呼び出し

オペレーティングシステムルーチンを直接呼び出すことはできますが，オペレーティングシステムルーチンは実行時までアプリケーションにリンクされません。そのため，このようなライブラリはプログラムのコンパイル時には必要ありません。また，ルーチンのインポートを試みるコンパイル時の検証も行われません。

共有オブジェクトの中に定義されているルーチンを呼び出すには，最初にインポートしなければなりません。共有オブジェクトのルーチンをインポートするには，**external** 宣言を使うか，またはオペレーティングシステムを直接呼び出します。どちらの場合も，共有オブジェクトルーチンへのリンクは実行時に行われます。したがって，プログラムのコンパイル時に共有オブジェクトが必要となることはありません。

Object Pascal では，共有ライブラリからの変数のインポートはサポートされていません。

静的ロード

手続きまたは関数をインポートする場合は，**external** 指令を使って宣言する方法が一番簡単です。次に例を示します。

```
Windows の場合： procedure DoSomething; external 'MYLIB.DLL';
Linux の場合：   procedure DoSomething; external 'mylib.so';
```

この宣言を記述すると，プログラムの起動時に MYLIB.DLL（Windows）または mylib.so（Linux）がロードされます。プログラムが実行している間，識別子 DoSomething は常に同じ共有ライブラリの同じエントリポイントを参照します。

インポートするルーチンの宣言は，それを呼び出すプログラムまたはユニットの中に直接配置できます。保守管理の観点から，一連の **external** 宣言は，ライブラリとのインターフェースに必要となる定数や型と一緒に「インポートユニット」にまとめておくとよいでしょう。ほかのモジュールもこのインポートユニットを使用すれば，中で宣言されているルーチンを自由に呼び出すことができます。

external 宣言についての詳細は，6-6 ページの「external 宣言」を参照してください。

動的ロード

ライブラリのルーチンにアクセスする方法としては，OS のライブラリ関数（LoadLibrary, FreeLibrary，GetProcAddress など）を直接呼び出す方法もあります。Windows では，これらの関数は Windows.pas で宣言されています。Linux では，これらの関数は互換性のために SysUtils.pas で実装されていますが，実際の Linux OS ルーチンは dlopen，dlclose，および dlsym です（これらは Kylix の Libc ユニットで宣言されています。詳細については，man ページを参照してください）。この場合は，インポートしたルーチンの参照に手続き型の変数を使ってください。

Windows または Linux での例を示します。

```
uses Windows, ...;   {Linux では , Windows を SysUtils に置き換える }
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('DATETIME.DLL');
  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;
```

この方法でルーチンをインポートした場合，ライブラリは，LoadLibrary への呼び出しを含むコードが実行された時点で初めてロードされます。ライブラリのアンロードは，FreeLibrary の呼び出しによって行われます。この方法の利点は，メモリの消費を節約できることと，一部のライブラリが存在していなくてもプログラムの実行を開始できることです。

上の例は，Linux では次のように書くことができます。

```
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;
```

この場合は，ルーチンをインポートしたとき，ライブラリは，dlopen への呼び出しを含むコードが実行された時点で初めてロードされます。ライブラリのアンロードは，dlclose の呼び出しによって行われます。この方法の利点も，メモリの消費を節約できることと，一部のライブラリが存在していなくてもプログラムの実行を開始できることです。

動的にロード可能なライブラリの記述

動的にロード可能なライブラリのメインソースは，プログラムのメインソースとほとんど変わりませんが，その開始には，予約語 **program** ではなく，**library** を使います。

ほかのライブラリやプログラムによるインポートに使用できるのは，ライブラリが明示的にエクスポートできるルーチンのみです。次に，エクスポートされた 2 つの関数（Min と Max）を持つライブラリの例を示します。

```
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.
```

ほかの言語で書かれたアプリケーションからもライブラリを利用できるようにするには，できる限り，エクスポートする関数の宣言に **stdcall** を指定してください。言語によっては，Object Pascal のデフォルトの **register** 呼び出し規約をサポートしていないものもあります。

ライブラリは，複数のユニットから作成することもできます。その場合，ライブラリのソースファイルは一般的に，**uses** 節，**exports** 節，初期化コードだけで構成します。次に例を示します。

```
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.
```

exports 節は，ユニットのインターフェース部または実現部に指定することができます。このようなユニットが **uses** 節に含まれているライブラリでは，そのユニットの **exports** 節に指定されているルーチンが自動的にエクスポートされます。ライブラリでさらに **exports** 節を指定する必要はありません。

ルーチンをエクスポート不可としてマークする **local** 指令は，プラットフォーム固有の指令で，Windows プログラミングには何の影響もありません。

Linux では，**local** 指令によって，ライブラリにエクスポートされずにコンパイルされるルーチンのパフォーマンスがわずかに最適化されます。この指令は，単独で機能する手続きや関数に対して指定できますが，メソッドに対しては指定できません。**local** によって宣言されたルーチンの例を次に示します。

```
function Contraband(I: Integer): Integer; local;
```

このルーチンは EBX レジスタをリフレッシュしません。その結果次のようになります。

- ライブラリからエクスポートできない。
- ユニットの **interface** セクションで宣言できない。
- 手続き型の変数に対してアドレスが割り当てられない。
- 純粋なアセンブラルーチンである場合は，呼び出し側が EBX を設定しない限り，ほかのユニットから呼び出すことはできない。

exports 節

ルーチンをエクスポートするには，次の形式に従って **exports** 節に指定します。

`exports` entry$_1$, ..., entry$_n$;

各エントリには，手続き，関数，または変数の名前を記述し，その後に（オーバーロードされているルーチンをエクスポートする場合は）パラメータリストを，さらにその後に任意で **name** 識別子を記述します。手続き，関数，変数は，**exports** 節の前に宣言しておく必要があります。手続きまたは関数の名前は，ユニット名を使って限定することもできます。

（エントリには **resident** 指令を入れることもできますが，この指令は下位互換性のためにのみ残されており，コンパイラには無視されます）。

Windows では，**index** 指定子は，**index** 指令とその後に続く 1 ～ 2,147,483,647 の間の数値定数で構成されます（プログラムの効率を高めるには，低いインデックス値を使用します）。エントリに **index** 指定子がない場合，そのルーチンには自動的にエクスポートテーブル内の数値が割り当てられます。

メモ **index** 指定子は下位互換性のためにのみサポートされており，使用すると他の開発ツールでは問題が生じることがあります。

name 指定子は，**name** 指令と文字列定数から構成されます。**name** 指定子を省略した場合，ルーチンは，オリジナルの宣言名と同じスペル（大文字 / 小文字の区別あり）によってエクスポートされます。**name** 節は，ルーチンを別の名前でエクスポートしたいときに使います。次に例を示します。

```
exports
  DoSomethingABC name 'DoSomething';
```

オーバーロードした関数または手続きを，動的にロード可能なライブラリからエクスポートする場合には，**exports** 節にパラメータリストを指定しなければなりません。次に例を示します。

```
exports
  Divide(X, Y: Integer) name 'Divide_Ints',
  Divide(X, Y: Real) name 'Divide_Reals';
```

Windows では，オーバーロードルーチンのエントリには，**index** 指定子を含めてはなりません。

exports 節は，プログラムまたはライブラリの宣言部内，あるいはユニットの **interface** 部または **implementation** 部内で，どこにでもまた何回でも指定できます。プログラムに **exports** 節が含まれることはめったにありません。

ライブラリ初期化コード

ライブラリのブロックの実行部には，ライブラリの初期化コードを記述します。このコードは，ライブラリがロードされるたびに一度実行されます。通常，ウィンドウクラスの登録や変数の初期化といった処理を行います。ライブラリ初期化コードには，ExitProc 変数を使って終了手続きを組み込むこともできます（この詳細については，12-4 ページの「終了手続き」を参照してください）。この終了手続きは，ライブラリのアンロード時に実行されます。

ライブラリ初期化コードでは，ExitCode 変数をゼロ以外の値に設定することによって，エラーを通知できます。ExitCode は System ユニット内に宣言されており，デフォルト値は初期化の成功を示す 0 です。初期化コードで ExitCode をゼロ以外の値に設定すると，ライブラリがメモリからアンロードされたとき，呼び出し元アプリケーションに処理失敗が通知されます。処理できない例外がライブラリ初期化コードの実行中に発生した場合，呼び出し元アプリケーションへライブラリのロード失敗が通知されます。

初期化コードと終了手続きを持つライブラリの例を次に示します。

```
library Test;
var
  SaveExit: Pointer;
procedure LibExit;
begin
  ⋮  // ライブラリの終了コード
  ExitProc := SaveExit;  // 終了手続きのチェーンを復元します
end;
begin
  ⋮  // ライブラリ初期化コード
  SaveExit := ExitProc;  // 終了手続きのチェーンを保存します
  ExitProc := @LibExit;  // LibExit 終了手続きをインストールします
end.
```

DLL がアンロードされると，ExitProc が **nil** になるまで ExitProc に格納されたアドレスの呼び出しを繰り返すことによって，ライブラリの終了手続きが実行されます。ライブラリが使うユニットの各初期化部はライブラリの初期化コードの前に実行され，これらのユニットの終了部はライブラリの終了手続きの後に実行されます。

ライブラリ内のグローバル変数

共有ライブラリで宣言されているグローバル変数を Object Pascal のアプリケーションにインポートすることはできません。

ライブラリは一度に複数のアプリケーションが使う可能性があり，各アプリケーションは自分のプロセス空間にライブラリのコピーを持って，その中でグローバル変数を管理します。複数のライブラリ，またはライブラリの複数のインスタンスがメモリを共有するためには，メモリマップドファイルを使用する必要があります。詳細については，システムのマニュアルを参照してください。

ライブラリとシステム変数

System ユニットで定義されている変数の中には，ライブラリのプログラミングにおいて重要なものがあります。IsLibrary 変数は，コードがアプリケーションとライブラリのどちらで実行されているのかを示します。アプリケーションの場合は常に False，ライブラリの場合は常に True となります。HInstance は，ライブラリの実行中の間，そのインスタンスハンドルを保持します。CmdLine 変数はライブラリでは常に **nil** です。

DLLProc 変数を使うと，オペレーティングシステムがライブラリのエントリポイントに対して行う呼び出しをライブラリ側で監視することができます。この機能は，通常はマルチスレッドをサポートするライブラリでのみ使用します。DLLProc は Windows と Linux 両方で使用できますが，使い方は異なります。Windows では，DLLProc はマルチスレッドアプリケーションで使用し，Linux では，ライブラリがいつアンロードされるかを調べるために使用します。終了処理には，終了手続きではなく，終了処理部を使用する必要があります（3-5 ページの「終了処理部」を参照してください）。

オペレーティングシステムからの呼び出しを監視するには，次の例のように，整数パラメータを 1 つとるコールバック手続きを作成します。

```
procedure DLLHandler(Reason: Integer);
```

次に，この手続きのアドレスを DLLProc 変数に割り当てます。この手続きを呼び出すとき，次の値のいずれかを手続きに渡します。

DLL_PROCESS_DETACH	FreeLibrary（Linux では dlclose）の呼び出し，または正常なプロセスの終了によりライブラリがアドレス空間から解放されることを示す
DLL_PROCESS_ATTACH	LoadLibrary（Linux では dlopen）の呼び出しによりライブラリがアドレス空間にアタッチされていることを示す
DLL_THREAD_ATTACH	現在のプロセスが新しいスレッドを作成中であることを示す（Windows のみ）
DLL_THREAD_DETACH	スレッドが正常に終了中であることを示す（Windows のみ）

手続きの本体では，渡されたパラメータによってアクションを変えることができます。

ライブラリ内の例外と実行時エラー

動的にロード可能なライブラリで処理されない例外が発生した場合，その例外はライブラリの呼び出し元へ伝えられます。呼び出し元アプリケーションまたはライブラリが Object Pascal で作成されている場合は，通常の **try...except** 文で処理できます。

メモ　Linux では，ライブラリとアプリケーションの両方が同じ（EH コードを含む）実行時パッケージのセットを使って構築されている場合，または両方が ShareExcept にリンクされている場合には，これが唯一可能な方法です。

呼び出し元アプリケーションまたはライブラリが別のプログラミング言語で書かれている場合，その例外は例外コード $0EEDFACE のオペレーティングシステム例外として処理できます。オペレーティングシステム例外レコードの ExceptionInformation 配列の最初のエントリには例外アドレスが入っており，2 番目のエントリには Object Pascal 例外オブジェクトへの参照が入っています。

一般的には，例外がライブラリから外に出ないようにする必要があります。Windows では，Delphi 例外は OS 例外モデルにマップされます。Linux には例外モデルはありません。

SysUtils ユニットを使っていないライブラリの場合，例外サポートは使用不可になります。その場合，ライブラリ内で実行時エラーが発生すると，そのライブラリを呼び出したアプリケーションは終了します。ライブラリは，Object Pascal プログラムから呼び出されたかどうかを知ることができないため，ライブラリからアプリケーションの終了手続きを呼び出すことはできません。アプリケーションは単純に終了し，メモリから解放されます。

共有メモリマネージャ（Windows のみ）

Windows では，パラメータまたは関数の結果として長い文字列や動的配列を渡すルーチンを DLL からエクスポートする場合，直接渡しか，レコードまたはオブジェクト内でネストしてかにかかわらず，その DLL とクライアントアプリケーション（またはクライアント DLL）はすべて ShareMem ユニットを使う必要があります。このことは，あるアプリケーションか DLL が New または GetMem で割り当てたメモリを，別のモジュールが Dispose または FreeMem の呼び出しで割り当て解除する場合にもあてはまります。ShareMem は，必ず各プログラムまたはライブラリの **uses** 節の先頭のユニットとして指定してください。

ShareMem は，BORLANDMM.DLL メモリマネージャとのインターフェースとなるユニットです。BORLANDMM.DLL によりモジュールは，動的に割り当てられたメモリを共有することができます。BORLANDMM.DLL は，ShareMem を使うアプリケーションおよび DLL と一緒に使用する必要があります。アプリケーションまたは DLL が ShareMem を使うと，そのメモリマネージャは BORLANDMM.DLL のメモリマネージャによって置き換えられます。

Linux は，glibc の malloc を使って共有メモリを管理します。

パッケージ

パッケージとは，アプリケーションまたは IDE，あるいはその両方が使う特殊なコンパイル済みライブラリのことです。パッケージを使うと，アプリケーションのソースコードに影響を与えずに，アプリケーションの内容を変えることができます。これは，アプリケーションパーティショニングと呼ばれることもあります。

実行時パッケージは，ユーザーがアプリケーションを実行する際に機能を提供します。設計時パッケージは，IDE にコンポーネントをインストールしたり，カスタムコンポーネント用に特別なプロパティエディタを作成するときに使われます。1 つのパッケージを設計時と実行時の両方で使えるようにすることもできますが，多くの場合，設計時パッケージの **requires** 節で実行時パッケージを参照します。これは，設計時に使われるコードのほとんどは，アプリケーション実行時には必要ないからです。

ほかのライブラリと区別するために，次のファイルに格納されます。

- Windows：拡張子 .bpl （Borland パッケージライブラリ）の付いたパッケージファイル
- Linux：一般にプレフィクス bpl で始まり，拡張子 .so が付いたパッケージ

通常，パッケージはアプリケーションの起動時に静的にロードされます。ただし，SysUtils ユニットの LoadPackage ルーチンと UnloadPackage ルーチンを使えば，動的にパッケージをロードできます。

メモ アプリケーションがパッケージを利用する場合，各パッケージユニットの名前はそれを参照しているソースファイルの **uses** 節にも含まれている必要があります。パッケージについての詳細は，オンラインヘルプを参照してください。

パッケージの宣言とソースファイル

各パッケージの宣言は，できる限り拡張子 .dpk を持つ独立したソースファイルの中で行い，Object Pascal のコードを含むほかのファイルとの混乱を避けるようにしてください。パッケージのソースファイルには，型，データ，手続き，関数の宣言は入れません。代わりに，以下のものを入れます。

- パッケージの名前
- 新しいパッケージが必要とするほかのパッケージのリスト。これらのパッケージは，新しいパッケージのリンク先となる
- コンパイル時にパッケージに取り込まれる（パッケージに結合される）ユニットファイルのリスト。このパッケージは，実質的にはコンパイル済みパッケージの機能を提供するこれらのソースコードユニットのラッパーとなる

パッケージの宣言は，次の形式で記述します。

```
package packageName;
  requiresClause;
  containsClause;
end.
```

packageName には有効な識別子を記述します。requiresClause と containsClause はどちらも省略できます。たとえば，次のコードは VCLDB40 パッケージを宣言します。

```
package DATAX;
requires
  baseclx,
  visualclx;
  contains Db, DBLocal, DBXpress, ... ;
end.
```

requires 節には，宣言対象のパッケージが使うほかの外部パッケージのリストを指定します。この **requires** 節は，**requires** 指令とカンマ区切りによるパッケージ名のリストから構成されます。最後にセミコロンを付けます。パッケージがほかのパッケージを参照しない場合，**requires** 節は不要です。

contains 節は，コンパイルしてパッケージに結合するユニットファイルを識別します。これは，**contains** 指令の後にカンマ区切りのユニット名のリストが続き，セミコロンで終わります。ユニット名の後には，予約語 **in** を使ってソースファイルの名前を指定できます。単引用符で囲み，ディレクトリパスは省略してもかまいません。ディレクトリパスは絶対パス，相対パスのどちらも使用できます。次に例を示します。

```
contains MyUnit in 'C:\MyProject\MyUnit.pas'; // Windows
contains MyUnit in '/home/developer/MyProject/MyUnit.pas'; // Linux
```

メモ　パッケージユニットの中で **threadvar** により宣言されているスレッドローカル変数には，パッケージを使っているクライアントからアクセスすることはできません。

パッケージの命名

コンパイルされたパッケージは，複数の生成ファイルで構成されます。たとえば，DATAX というパッケージのソースファイルは DATAX.dpk で，そこからコンパイラは実行形式ファイルとバイナリイメージを生成します。

- Windows：DATAX.bpl および DATAX.dcp
- Linux：bplDATAX.so および DATAX.dcp

DATAX は，ほかのパッケージの **requires** 節でそのパッケージを参照するときや，アプリケーション内でそのパッケージを利用するときに使われます。パッケージは，プロジェクト内に同じ名前のものがあってはなりません。

requires 節

requires 節は現在のパッケージで使われるほかの外部パッケージのリストを指定します。これは，ユニットファイル内の **uses** 節と同じように機能します。**requires** 節に指定された外部パッケージは，コンパイル時に自動的に，現在のパッケージと外部パッケージに含まれるユニットの両方を使うアプリケーションにリンクされます。

パッケージ内のユニットファイルがほかのパッケージユニットを参照する場合，ほかのパッケージは先頭のパッケージの **requires** 節に含まれている必要があります。ほかのパッケージが **requires** 節から省略された場合，コンパイラは参照先ユニット .dcu（Windows）または .dpu（Linux）ファイルからロードします。

パッケージの循環参照の回避

パッケージはその **requires** 節に循環参照を含むことはできません。これは以下のことを意味します。

- パッケージは自分の **requires** 節内で自分自身を参照することができない
- 参照のチェーンは，チェーン内のどのパッケージも再度参照することなく終了しなければならない。たとえば，パッケージ A がパッケージ B を要求する場合，パッケージ B はパッケージ A を要求することができず，パッケージ A がパッケージ B を要求し，パッケージ B がパッケージ C を要求する場合，パッケージ C はパッケージ A を要求することができない

パッケージの重複参照

コンパイラはパッケージの **requires** 節にある重複参照を無視します。しかし，プログラミング上明確かつ読みやすくするために重複参照は削除すべきです。

contains 節

contains 節はパッケージに結合するユニットファイルを識別します。**contains** 節にはファイル名拡張子を入れないでください。

冗長なソースコード利用の回避

パッケージは，別のパッケージの **contains** 節や，ユニットの **uses** 節に指定することはできません。

パッケージの **contains** 節に直接含まれるすべてのユニット，およびそれらのユニットの **uses** 節によって間接的に含まれるすべてのユニットは，コンパイル時にパッケージに結合されます。パッケージに間接または直接に含まれるユニットを，そのパッケージの **requires** 節で参照されるほかのパッケージに含めることはできません。

同一のアプリケーションが使う複数のパッケージが，同じユニットを間接または直接に含めることはできません。

パッケージのコンパイル

パッケージは通常，パッケージエディタで生成された .dpk ファイルを使って IDE からコンパイルされます。コマンドラインコンパイラを使って直接 .dpk ファイルをコンパイルすることもできます。パッケージが入ったプロジェクトを構築する場合，そのパッケージは必要に応じて暗黙に再コンパイルされます。

生成ファイル

パッケージのコンパイルに成功したときに生成されるファイルの一覧を次の表に示します。

表 9.1　コンパイル済みのパッケージファイル

ファイル拡張子	内容
dcp	パッケージヘッダーと，パッケージ内のすべての dcu（Windows）または dpu（Linux）ファイルが結合されたものが入ったバイナリイメージ。パッケージごとに 1 つの dcp ファイルが生成される。dcp の基本名は dpk ソースファイルの基本名となる
dcu（Windows） dpu（Linux）	パッケージに含まれるユニットファイルのバイナリイメージ。必要に応じて，ユニットファイルごとに 1 つの dcu または dpu が生成される
.bpl（Windows） bpl<package>.so（Linux）	実行時パッケージ。特殊な Borland 固有の機能を備えたライブラリ。パッケージのベース名は，dpk ソースファイルのベース名

パッケージのコンパイルには，数種類のコンパイラ指令とコマンドラインスイッチが利用できます。

パッケージ固有のコンパイラ指令

ソースコードに挿入できるパッケージ固有のコンパイラ指令の一覧を次の表に示します。詳細は，オンラインヘルプを参照してください。

表 9.2　パッケージ固有のコンパイラ指令

指令	用途
{$IMPLICITBUILD OFF}	パッケージが後で暗黙に再コンパイルされないようにする。低レベルの機能を提供するパッケージや，ビルド（構築）間で頻繁に変更されるパッケージ，あるいはソースコードが提供されていないパッケージをコンパイルする際に .dpk ファイルで使用する
{$G-} または {$IMPORTEDDATA OFF}	インポートデータ参照の作成を無効にする。この指令を使うと，メモリアクセスの効率は向上するが，ユニットはパッケージ化されない
{$WEAKPACKAGEUNIT ON}	ユニットを「弱く」パッケージ化する。詳細はオンラインヘルプを参照
{$DENYPACKAGEUNIT ON}	ユニットがパッケージ内に入らないようにする。

表 9.2　パッケージ固有のコンパイラ指令（つづき）

指令	用途
{$DESIGNONLY ON}	パッケージを設計時専用としてコンパイルする（.dpk ファイルに入れる）
{$RUNONLY ON}	パッケージを実行時専用としてコンパイルする（.dpk ファイルに入れる）

ソースコードに **{$DENYPACKAGEUNIT ON}** を入れると，ユニットファイルはパッケージ化されなくなります。**{$G-}** または **{$IMPORTEDDATA OFF}** を入れると，パッケージを同一のアプリケーション内でほかのパッケージと一緒に使えなくなります。

パッケージソースコードには，適宜ほかのコンパイラ指令も入れることができます。

パッケージ固有のコマンドラインコンパイラスイッチ

コマンドラインコンパイラでは，以下に示すパッケージ固有のスイッチを使用できます。詳細は，オンラインヘルプを参照してください。

表 9.3　パッケージ固有のコマンドラインコンパイラスイッチ

スイッチ	用途
-$G-	インポートデータ参照の作成を無効にする。このスイッチを使うと，メモリアクセスの効率は向上するが，ユニットはパッケージ化されない
-LE path	コンパイル済みのパッケージファイルが格納されるディレクトリを指定する
-LN path	パッケージの dcp ファイルが格納されるディレクトリを指定する
-LUpackageName [;packageName2;...]	アプリケーションで使用する追加の実行時パッケージを指定する。プロジェクトのコンパイル時に使用される
-Z	パッケージが後で暗黙に再コンパイルされないようにする。ビルド間での変更がめったにない，そのソースコードの配布予定がない，あるいは低レベル機能を提供するパッケージをコンパイルするときに使う

-$G- スイッチを使うと，パッケージを同一のアプリケーション内でほかのパッケージと一緒に使えなくなります。

パッケージのコンパイル時には，適宜ほかのコマンドラインオプションも使えます。

第10章

オブジェクトインターフェース

オブジェクトインターフェース（単にインターフェースとも呼びます）は，クラスによって実装できるメソッドを定義します。インターフェースはクラスと同じように宣言しますが，直接にインスタンス化することはできません。また，インターフェースが独自のメソッド定義を持つことはありません。インターフェースをサポートしているクラスの側で，インターフェースのメソッドを実装しなければなりません。インターフェース型の変数は，オブジェクトのクラスがそのインターフェースを実装している場合にオブジェクトを参照できますが，このような変数を使って呼び出せるのは，そのインターフェースで宣言されたメソッドだけです。

インターフェースを利用すると，意味的な難しさを回避しながら多重継承のメリットの一部を享受できます。また，分散型オブジェクトモデルを使う場合にも重要な役割を果たします。インターフェースをサポートするカスタムオブジェクトは，C++ や Java などの言語で作成されたオブジェクトと対話することができます。

インターフェース型

インターフェースはクラスの場合と同様，プログラムまたはユニットの最も外側のスコープでのみ宣言できます。手続きや関数の中では宣言できません。インターフェース型の宣言は，次の形式で記述します。

```
type interfaceName = interface (ancestorInterface)
  ['{GUID}']
  memberList
end;
```

ただし，(ancestorInterface) と ['{GUID}'] は省略可能です。インターフェースの宣言は基本的にクラスの宣言と変わりませんが，以下の制限事項が適用されます。

- memberList にメソッドとプロパティ以外を含めることはできない。フィールドをインターフェースで使うことはできない。

- インターフェースはフィールドを持たないため，プロパティの **read** 指定子と **write** 指定子はメソッドとしなければならない。
- インターフェースのメンバーはすべてパブリックとなる。可視性指定子と格納指定子は使用できない。ただし，配列プロパティは **default** として宣言できる。
- インターフェースはコンストラクタとデストラクタを持たない。原則としてインスタンス化はできないが，そのメソッドを実装しているクラスをとおすことによって可能になる。
- メソッドは **virtual**，**dynamic**，**abstract**，**override** として宣言できない。インターフェース自体はメソッドを実装しないため，これらの宣言は意味を持ち得ない。

インターフェースの宣言の例を次に示します。

```
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;
```

インターフェースの宣言によっては，予約語 **interface** の代わりに **dispinterface** を使う場合もあります。この構造は（**dispid**，**readonly**，および **writeonly** 指令とともに）プラットフォームに依存するものであり，Linux プログラミングでは使われません。

IInterface と継承

インターフェースはクラスの場合と同様，上位インターフェースのメソッドをすべて継承します。ただしメソッドを実装しない点がクラスと異なります。インターフェースが継承するのは，メソッドの実装に関する契約です。ここでいう契約とは，インターフェースをサポートしているクラスに依存するということを意味します。

インターフェースを宣言するときは上位インターフェースを指定できます。上位インターフェースの指定を省略すると，そのインターフェースは IInterface の直下の下位インターフェースとして位置付けられます。IInterface は System ユニットに定義されており，その他すべてのインターフェースの最上位に位置付けられています。IInterface では，QueryInterface，_AddRef，_Release の 3 つのメソッドが宣言されています。

メモ IInterface は IUnknown と同等です。一般的には，プラットフォームに依存しないアプリケーションには IInterface を使用し，Windows に依存する部分があるプログラムには IUnknown を使う必要があります。

QueryInterface は，オブジェクトがサポートする各インターフェース間を自由に移動するための手段を提供します。_AddRef と _Release は，インターフェース参照の寿命をメモリ管理するために使います。これらのメソッドを実装するときは，System ユニットの TInterfacedObject から実装クラスを派生させる方法がもっとも簡単です。また，いずれのメソッドも empty 関数として実装することに

よって破棄できます。ただし，COM オブジェクト（Windows のみ）は _AddRef と _Release を使って管理しなければなりません。

インターフェースの識別

インターフェースの宣言では，グローバルユニーク識別子（GUID）を指定できます。GUID は，大カッコで囲った文字列リテラルとそれに続くメンバーリストで構成されます。宣言の GUID 部は次の構文に従って記述します。

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

ここで，x は 16 進数（0 ～ 9，A ～ F）を表します。Windows では，タイプライブラリエディタによって新しいインターフェースの GUID が自動的に生成されます。コードエディタで〔Ctrl〕+〔Shift〕+〔G〕を押して GUID を生成することもできます（Linux では，コードエディタで〔Ctrl〕+〔Shift〕+〔G〕を押す方法を使わなければなりません）。

GUID は インターフェースを一意に識別する 16 ビットバイナリ値です。インターフェースに GUID がある場合は，インターフェース問い合わせを使うことによってその実装への参照を取得できます（10-10 ページの「インターフェースの問い合わせ」参照）。

System ユニットで宣言されている TGUID 型と PGUID 型は GUID の操作に使います。

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

TGUID 型の型付き定数を宣言するときは，文字列リテラルを使ってその値を指定できます。次に例を示します。

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

手続きと関数の呼び出しでは，GUID とインターフェース識別子のどちらも，TGUID 型の値または定数パラメータとして機能します。たとえば，次のような宣言があるとします。

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports は次の 2 つの方法で呼び出せます。

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

インターフェースの呼び出し規約

デフォルトの呼び出し規約は **register** ですが，インターフェースがモジュール間で共有されている場合，特にモジュールが異なる言語で記述されている場合は，すべてのメソッドを **stdcall** で宣言する必要があります。CORBA インターフェースを実装するときは，**safecall** を使用してください。Windows では，**safecall** を使って，デュアルインターフェース（10-13 ページの「デュアルインターフェース（Windows のみ）」参照）のメソッドを実装することができます。

呼び出し規約についての詳細は，6-4 ページの「呼び出し規約」を参照してください。

インターフェースのプロパティ

インターフェースで宣言されたプロパティは，インターフェース型の式をとおしてのみアクセスできます。クラス型変数からはアクセスできません。また，インターフェースのプロパティは，インターフェースがコンパイルされているプログラムの中からのみ参照できます。たとえば Windows では，COM オブジェクトの場合はプロパティは持ちません。

インターフェースではフィールドを使えないため，プロパティの **read** 指定子と **write** 指定子はメソッドとしなければなりません。

前方参照宣言

インターフェースの宣言が予約語 **interface** とセミコロンだけで終わっており，上位インターフェース，GUID，メンバーリストの指定が省略されている場合，その宣言は前方参照宣言となります。前方参照宣言は，同じ型宣言セクションの中で同じインターフェースの宣言を定義することによって解決しなければなりません。つまり，前方参照宣言とその定義宣言の間に，ほかの型宣言以外のものを含めることはできません。

forward 宣言を使用することで，相互に依存したインターフェースを使用できるようになります。次に例を示します。

```
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;
```

相互派生のインターフェースは許可されていません。たとえば，IControl から IWindow を派生させると同時に，IWindow から IControl を派生させることはできません。

インターフェースの実装

インターフェースを宣言しても，クラスの中で実装しない限り実際に使うことはできません。クラスによって実装されるインターフェースは，そのクラスの宣言内で，上位クラスの名前の後ろに指定します。宣言の形式は次のとおりです。

```
type className = class (ancestorClass, interface1, ..., interfacen)
  memberList
 end;
```

次に例を示します。

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ⋮
  end;
```

この例では，IMalloc インターフェースと IErrorInfo インターフェースを実装する RMemoryManager という名前のクラスを宣言しています。クラスでインターフェースを実装するときは，インターフェースで宣言されている各メソッドを実装するか，各メソッドの実装を継承しなければなりません。

TInterfacedObject は，System ユニットの中で次のように宣言されています。

```
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 は IInterface インターフェースを実装しています。したがって，TInterfacedObject は，IInterface の 3 つのメソッドのそれぞれを宣言し，実装していることになります。

インターフェースを実装するクラスは基本クラスとしても使えます。上記の最初の例では，TMemoryManager を TInterfacedObject の直下の下位インターフェースとして宣言しています。インターフェースはすべて IInterface から継承されるため，インターフェースを実装するクラスは QueryInterface，_AddRef，および _Release の各メソッドを実装しなくてはなりません。System ユニットの TInterfacedObject はこの 3 つのメソッドを実装しているので，インターフェースを実装するクラスを派生させるときは，これを基本クラスとして利用できます。

インターフェースが実装されると，その各メソッドは，結果型，呼び出し規約，パラメータの数，および対応する各パラメータの型までがすべて同じである実装クラスのメソッドにマッピングされます。デフォルトでは，各インターフェースメソッドは実装クラスの中で同じ名前を持つメソッドにマッピングされます。

メソッド解決節

デフォルトの名前に基づいたマッピングは，クラス宣言内にメソッド解決節を入れることでオーバーライドできます。同じ名前のメソッドを持つインターフェースをクラスが複数実装している場合は，メソッド解決節を使って名前の衝突を解決します。

メソッド解決節の形式は以下のとおりです。

```
procedure interface.interfaceMethod = implementingMethod;
```

または

```
function interface.interfaceMethod = implementingMethod;
```

implementingMethod は，クラスまたはその上位クラスのひとつで宣言されているメソッドです。後でクラス宣言の中で宣言されるメソッドである場合は問題ありませんが，ほかのモジュールで宣言される上位クラスのプライベートメソッドであることはできません。

たとえば，次のクラス宣言は，

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ⋮
  end;
```

IMalloc の Alloc メソッドと Free メソッドを TMemoryManager の Allocate メソッドと Deallocate メソッドにマッピングします。

メソッド解決節では，上位クラスで導入されたマッピングを変えることはできません。

継承実装の変更

下位クラスでは，実装メソッドをオーバーライドすることによって，特定のインターフェースメソッドの実装方法を変更できます。このとき，実装メソッドは仮想メソッドまたは動的メソッドである必要があります。

クラスはまた，上位クラスから継承したインターフェースの全体を再実装することもできます。再実装するには，下位クラスの宣言にインターフェースをあらためて記述します。次に例を示します。

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    ⋮
  end;
  TWindow = class(TInterfacedObject, IWindow)  // TWindow は IWindow を実装する
    procedure Draw;
    ⋮
  end;
  TFrameWindow = class(TWindow, IWindow)  // TFrameWindow は IWindow を再実装する
    procedure Draw;
    ⋮
  end;
```

インターフェースを再実装すると，同じインターフェースの継承した実装は隠蔽されます。したがってこの場合，上位クラスのメソッド解決節は，再実装されたインターフェースでは無視されます。

委任によるインターフェースの実装

implements 指令を使うと，インターフェースの実装を実装クラス内のプロパティに委任することができます。次に例を示します。

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

この例ではインターフェース IMyInterface を実装する MyInterface というプロパティを宣言しています。

implements 指令は，プロパティ宣言の最後の指定子として宣言しなければならず，コンマで区切ることによって複数のインターフェースを列挙することができます。委任プロパティには，以下の条件が適用されます。

- クラス型かインターフェース型でなければならない。
- 配列プロパティにはできない。インデックス指定子を持つことはできない。
- **read** 指定子を持たなければならない。プロパティが **read** メソッドを使用している場合，このメソッドはデフォルトの **register** 呼び出し規約を使用しなくてはなりません。また，動的メソッドにすることも（仮想メソッドは可能），**message** 指令を指定することもできません。

メモ 委任されたインターフェースを実装するために使用するクラスは，TAggregatedObjec から派生させる必要があります。

インターフェース型プロパティへの委任

委任プロパティの型がインターフェース型の場合は，プロパティが宣言されているクラスの上位クラスのリストに，そのインターフェース，またはその派生元のインターフェースを記述しなければなりません。委任プロパティは，**implements** 指令で指定されたインターフェースを（メソッド解決節は使わずに）完全に実装しているクラスのオブジェクトを返さなければなりません。次に例を示します。

```
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 := ...  // クラスが IMyInterface を実装しているオブジェクト
  MyInterface := MyClass;
  MyInterface.P1;
end;
```

クラス型プロパティへの委任

委任プロパティの型がクラス型の場合，そのクラスと上位クラスの方が，封入クラスとその上位クラスより先に，指定されたインターフェースを実装しているメソッドの検索対象となります。したがって，一部のメソッドをプロパティが指定するクラスに実装し，残りをプロパティが宣言されているクラスに実装することが可能です。メソッド解決節は，通常どおりにあいまいさの解決に使うことも，特定のメソッドの指定に使うこともできます。インターフェースを複数のクラス型プロパティで実装することはできません。次に例を示します。

```
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;       // TMyClass.MyP1 を呼び出す
  MyInterface.P2;       // TImplClass.P2 を呼び出す
end;
```

インターフェース参照

インターフェース型として宣言した変数は，そのインターフェースを実装しているクラスであればどのインスタンスでも参照できます。このような変数を使えば，インターフェースの実装場所をコンパイル時に知らなくても，インターフェースのメソッドを呼び出すことができます。ただし，これには以下の制限が適用されます。

- インターフェース型の式では，インターフェースで宣言されているメソッドとプロパティにのみアクセスでき，実装クラスのほかのメンバーにはアクセスできない。
- インターフェース型の式は，クラス（またはその継承元のクラス）が明示的に上位インターフェースも実装していない限り，下位インターフェースを実装しているクラスのオブジェクトを参照できない。

次に例を示します。

```
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;  // OK!
    A := TSomething.Create;  // エラー
    D.P1;  // OK!
    D.P2;  // エラー
  end;
```

この例では,

- A は IAncestor 型の変数として宣言されている。TSomething は，実装しているインターフェースのリストに IAncestor がないため，TSomething のインスタンスを A に代入することはできない。ただし，TSomething の宣言を，次のように変更すると

  ```
  TSomething = class(TInterfacedObject, IAncestor, IDescendant)
   ⋮
  ```

  1 番目のエラーの代入文は有効になる。

- D は IDescendant 型の変数として宣言されている。D は TSomething のインスタンスを参照しているが，これを使って TSomething の P2 メソッドにアクセスすることはできない。P2 が IDescendant のメソッドではないことがその理由である。ただし，D の宣言を以下のように変えると,

  ```
  D: TSomething;
  ```

  2 番目のエラーは有効なメソッド呼び出しとなる。

インターフェースの参照は参照カウンタによって管理されます。これは IInterface から継承された _AddRef メソッドと _Release メソッドに依存して機能します。オブジェクトの参照がインターフェース経由のみの場合，オブジェクトを手動で廃棄する必要はありません。最後の参照がスコープ外へ出たときに自動的に廃棄されます。

インターフェース型のグローバル変数は **nil** 以外に初期化できません。

インターフェース型の式がオブジェクトを参照しているかどうかを調べるには，標準関数 Assigned に渡します。

インターフェースの代入互換性

クラス型は，そのクラスによって実装されるどのインターフェース型とも代入の互換性があります。また，インターフェース型はどの上位インターフェース型とも代入の互換性があります。値 **nil** は，どのインターフェース型変数にも代入できます。

インターフェース型の式はバリアントに代入できます。インターフェースが IDispatch 型または下位の場合，バリアントは型コード varDispatch を受け取ります。それ以外の場合は，型コード varUnknown を受け取ります。

型コードが varEmpty，varUnknown，または varDispatch のバリアントは，IInterface 変数に代入できます。型コードが varEmpty または varDispatch であるバリアントは，IDispatch 変数に代入できます。

インターフェースの型キャスト

インターフェース型は，変数および値の型キャストの点でクラス型と同じ規則に従います。クラス型の式は，`IMyInterface(SomeObject)` のような形で，インターフェース型にキャストできます。ただし，クラスがインターフェースを実装していることが条件です。

インターフェース型の式はバリアント型にキャストできます。インターフェースが IDispatch 型または下位であれば，その結果のバリアントは varDispatch の型コードを持ちます。それ以外の場合は，varUnknown の型コードを持ちます。

型コードが varEmpty，varUnknown，または varDispatch のバリアントは，IInterface にキャストできます。型コードが varEmpty または varDispatch のバリアントは，IDispatch にキャストできます。

インターフェースの問い合わせ

as 演算子はチェック付きのインターフェースを型キャストをするのに使います。この作業はインターフェースの問い合わせと呼ばれ，オブジェクトの実際の（実行時の）型に基づいて，オブジェクト参照またはインターフェース参照からインターフェース型の式を生成します。インターフェースの問い合わせの形式は次のとおりです。

object `as` interface

ここで，object はインターフェースまたはバリアントの式であるか，またはインターフェースを実装するクラスのインスタンスを記述します。interface は GUID を使って宣言された任意のインターフェースです。

インターフェースの問い合わせは，オブジェクトが **nil** の場合 **nil** を返します。それ以外の場合は，インターフェースの GUID をオブジェクト中の QueryInterface メソッドに渡し，QueryInterface がゼロ以外を返すと例外を生成します。QueryInterface がゼロを返す場合（オブジェクトのクラスがインターフェースを実装している場合を含む），インターフェースの問い合わせはオブジェクトへのインターフェース参照を返します。

オートメーションオブジェクト（Windows のみ）

クラスが IDispatch インターフェースを実装しているオブジェクトは，オートメーションオブジェクトと呼びます（IDispatch インターフェースは System ユニットに宣言されています）。オートメーションオブジェクトは，Windows でのみ使用できます。

ディスパッチインターフェース型（Windows のみ）

ディスパッチインターフェース型は，オートメーションオブジェクトが IDispatch 経由で実装するメソッドとプロパティを定義するものです。ディスパッチインターフェースのメソッドの呼び出しは，

実行時に IDispatch の Invoke メソッド経由でルート指定されます。クラスはディスパッチインターフェースを実装できません。

ディスパッチインターフェースの型宣言の形式は次のとおりです。

```
type interfaceName = dispinterface
  ['{GUID}']
  memberList
end;
```

['{GUID}'] は省略でき，memberList はプロパティとメソッドの宣言で構成されます。ディスパッチインターフェースの宣言は通常のインターフェース宣言と似ていますが，上位を指定できません。次に例を示します。

```
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;
```

ディスパッチインターフェースのメソッド（Windows のみ）

ディスパッチインターフェースのメソッドは，基礎になる IDispatch 実装の Invoke メソッドの呼び出しに対するプロトタイプです。メソッドに対するオートメーションディスパッチ ID を指定するには，その宣言中に **dispid** 指令を入れ，その後に整数定数を続けてください。すでに使われている ID を指定するとエラーになります。

ディスパッチインターフェースで宣言されたメソッドが **dispid** 以外の指令を含むことはできません。パラメータと結果の型はオートメーション可能な型でなければなりません。つまり，Byte，Currency，Real，Double，Longint，Integer，Single，Smallint，AnsiString，WideString，TDateTime，Variant，OleVariant，WordBool，またはインターフェース型のいずれかでなければならないということです。

ディスパッチインターフェースのプロパティ（Windows のみ）

ディスパッチインターフェースのプロパティはアクセス指定子を含みません。このプロパティは **readonly** または **writeonly** として宣言できます。プロパティのディスパッチ ID を指定するには，その宣言中に **dispid** 指令を入れ，その後に整数定数を続けてください。すでに使われている ID を指定するとエラーになります。配列プロパティは **default** として宣言できます。ディスパッチインターフェースのプロパティの宣言に，これ以外の指令を入れることはできません。

オートメーションオブジェクトへのアクセス（Windows のみ）

オートメーションオブジェクトにアクセスするには，バリアントを使います。バリアントがオートメーションオブジェクトを参照しているときは，バリアントを通して，オブジェクトのメソッドの呼

び出しや，プロパティの参照，設定ができます。ただし，いずれかのユニット，プログラムまたはライブラリに **uses** 節で，ComObj を含めておく必要があります。

オートメーションオブジェクトのメソッド呼び出しは実行時に結合されます。事前にメソッドを宣言しておく必要はありません。これらの呼び出しが有効かどうかは，コンパイル時には確認されません。

次にオートメーションメソッド呼び出しの例を示します。CreateOleObject 関数（ComObj 内に定義されている）は，オートメーションオブジェクトへの IDispatch 参照を返します。また，バリアント Word と代入の互換性があります。

```
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;
```

インターフェース型パラメータはオートメーションメソッドに渡すことができます。

要素型が varByte のバリアント配列は，オートメーションのコントローラとサーバーの間でバイナリデータを受け渡すのによく使われます。そのような配列はデータの変換によって影響を受けず，VarArrayLock ルーチンと VarArrayUnlock ルーチンを使って効率的にアクセスできます。

オートメーションオブジェクトのメソッド呼び出しの構文

オートメーションオブジェクトのメソッド呼び出しまたはプロパティアクセスの構文は，通常のメソッド呼び出しまたはプロパティアクセスの構文に似ています。ただし，オートメーションのメソッド呼び出しでは定位置パラメータも名前付きパラメータも使用できます（一部のオートメーションサーバーは名前付きパラメータをサポートしていません）。

定位置パラメータは単なる式です。名前付きパラメータはパラメータ識別子の後に記号 **:=** を続け，その後に式を置きます。定位置パラメータはメソッド呼び出しの中で名前付きパラメータの前に置かなければなりません。名前付きパラメータはどんな順序でも指定できます。

一部のオートメーションサーバーでは，メソッド呼び出しでパラメータを省略して，デフォルト値をそのまま採用することもできます。次に例を示します。

```
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');
```

オートメーションのメソッド呼び出しパラメータは整数型，実数型，文字列型，論理型，バリアント型のいずれかの型にできます。パラメータはパラメータ式が変数参照だけで構成されている場合，その変数参照が Byte，Smallint，Integer，Single，Double，Currency，TDateTime，AnsiString，WordBool，Variant のいずれかの型であると参照渡しされます。式が上記の型のどれでもない場合，または単なる変数でない場合，パラメータは値渡しされます。値パラメータを期待するメソッドへパラメータを参照渡しすると，COM は参照パラメータから値を取得します。参照パラメータを期待するメソッドへパラメータを値渡しするとエラーになります。

デュアルインターフェース（Windows のみ）

デュアルインターフェースは，オートメーションによりコンパイル時の結合と実行時の結合の両方をサポートするインターフェースです。デュアルインターフェースは IDispatch から派生しなければなりません。

デュアルインターフェースのすべてのメソッド（IInterface および IDispatch から継承したものは除く）は **safecall** 規約を使わなければならず，すべてのメソッドパラメータおよび結果の型はオートメーション可能でなければなりません。オートメーション可能な型は，Byte，Currency，Real，Double，Real48，Integer，Single，Smallint，AnsiString，ShortString，TDateTime，Variant，OleVariant，WordBool の各型です。

第11章

メモリ管理

この章では，プログラムでのメモリの使われ方および Object Pascal で扱えるデータ型の内部形式について説明します。

メモリマネージャ（Windows のみ）

メモ Linux は，メモリ管理に malloc などの glibc 関数を使用します。詳細については，Linux システムで malloc の man ページを参照してください。

Windows システムでは，メモリマネージャはアプリケーションでのすべての動的なメモリ割り当てと割り当て解除を管理します。New，Dispose，GetMem，ReallocMem，FreeMem の各標準手続きはメモリマネージャを使い，すべてのオブジェクトと長い文字列はメモリマネージャを通して割り当てられます。

Windows では，メモリマネージャは，たとえばオブジェクト指向アプリケーションや文字列データを処理するアプリケーションなど，小さめなサイズの多数のブロックを割り当てるアプリケーション用に最適化されています。そのために，その他のメモリマネージャ，たとえば GlobalAlloc, LocalAlloc の実装や Windows でのプライベートヒープサポートなどを直接使用すると処理効率がよくないので，アプリケーションの処理速度が低下します。

最大限の処理効率を確保するため，メモリマネージャは Win32 仮想メモリ API（VirtualAlloc 関数と VirtualFree 関数）と直接やりとりします。メモリマネージャは 1 MB のアドレス領域を単位としてオペレーティングシステムからメモリを確保し，必要であれば 16 KB 単位でメモリをコミットしていきます。メモリマネージャは未使用のメモリを 16 KB と 1 MB 単位でコミット解除し解放します。もっと小さいブロックの場合，コミットされたメモリがさらに分割割り当てされます。

メモリマネージャのブロックは常に 4 バイト境界まで切り上げられ，常に 4 バイトのヘッダーを含んでおり，そのヘッダーにブロックのサイズとその他のステータスビットが格納されています。したがって，メモリマネージャのブロックは常にダブルワードアラインメントになり，これによってブロックをアドレス指定するときに最適な CPU 処理効率が保証されます。

メモリマネージャは AllocMemCount と AllocMemSize という 2 つのステータス変数を管理します。これらの変数には現在の割り当て済みメモリブロック数と，現在の割り当て済みメモリブロックを結合したサイズが入っています。これらの変数をアプリケーションで使うと，デバッグ用にステータス情報を表示できます。

System ユニットには GetMemoryManager と SetMemoryManager という 2 つの手続きがあり，低レベルのメモリマネージャ呼び出しを監視できるようになっています。GetHeapStatus という関数もあります。この関数はメモリマネージャの詳しいステータス情報が入ったレコードを返します。これらのルーチンについての詳細は，オンラインヘルプを参照してください。

変数

グローバル変数は，アプリケーションのデータセグメントに割り当てられ，プログラムの実行中は存続しつづけます。手続きや関数の中で宣言された変数はローカル変数と呼ばれ，アプリケーションのスタックに保持されます。手続きや関数の呼び出しのたびに，ローカル変数が割り当てられ，終了時に各ローカル変数は廃棄されます。コンパイラの最適化によって，変数は早い段階で廃棄されることもあります。

メモ Linux では，スタックサイズは環境によってのみ設定されます。

アプリケーションのスタックは，最小スタックサイズと最大スタックサイズの 2 つの値で定義されます。この 2 つの値は **$MINSTACKSIZE** 指令と **$MAXSTACKSIZE** 指令で制御します。それぞれのデフォルトは 16,384（16K）と 1,048,576（1M）です。アプリケーションは最小スタックサイズを利用できるよう保証されており，そしてアプリケーションのスタックが最大スタックサイズより大きくなることは決してありません。アプリケーションの最小スタックサイズ条件を満たすメモリが利用できない場合は，そのアプリケーションを起動しようとすると Windows からエラーが出されます。

Windows アプリケーションで，最小スタックサイズで指定されるよりも多いスタック領域が必要であれば，追加のメモリが 4K 単位で自動的に割り当てられます。追加メモリが利用できないか，スタックの合計サイズが最大スタックサイズを超えるため追加スタック領域の割り当てに失敗すると，EStackOverflow 例外が生成されます（スタックオーバーフローのチェックは完全に自動化されています。従来オーバーフローチェックを制御していた **$S** コンパイラ指令は，下位互換性のために残されています）。

Windows または Linux では，GetMem または New 手続きで作成した動的変数はヒープに割り当てられ，FreeMem または Dispose で割り当てが解除されるまで存続します。

長い文字列，ワイド文字列，動的配列，バリアント，およびインターフェースはヒープに割り当てられますが，そのメモリは自動的に管理されます。

内部データ形式

以下の節では，Object Pascal で扱えるデータ型の内部データ形式について説明します。

整数型

整数型の変数の形式はその変数の最小値と最大値によって決まります。

- 最小値と最大値がともに -128..127 の範囲にある場合（Shortint），変数は符号付きバイトとして格納される
- 最小値と最大値がともに 0..255 の範囲にある場合（Byte），変数は符号なしバイトとして格納される
- 最小値と最大値がともに -32768..32767 の範囲にある場合（Smallint），変数は符号付きワードとして格納される
- 最小値と最大値がともに 0..65535 の範囲にある場合（Word），変数は符号なしワードとして格納される
- 最小値と最大値がともに -2147483648..2147483647 の範囲にある場合（Longint），変数は符号付きダブルワードとして格納される
- 最小値と最大値がともに 0..4294967295 の範囲にある場合（Longword），変数は符号なしダブルワードとして格納される
- それ以外の場合は，変数は符号付きクワッドワード（Int64）として格納される

文字型

Char 型，AnsiChar 型または Char 型の部分範囲は符号なしバイトとして格納されます。WideChar 型は符号なしワードとして格納されます。

論理型

Boolean 型は Byte 型として格納され，ByteBool 型も Byte 型として格納されます。WordBool 型は Word 型として格納され，LongBool 型は Longint 型として格納されます。

Boolean 型は 0（False）または 1（True）の値をとります。ByteBool 型，WordBool 型，LongBool 型は 0（False）または 0 以外（True）の値をとります。

列挙型

列挙型は，値の個数が 256 以下で **{$Z1}** 状態（デフォルト）で宣言された場合，符号なしバイトとして格納されます。値の個数が 257 以上か，**{$Z2}** 状態で宣言された列挙型は，符号なしワードとして格納されます。列挙型が **{$Z4}** 状態で宣言された場合は，符号なしダブルワードとして格納されます。

実数型

実数型は，符号（+ か -），指数，仮数の 2 進表現を格納します。実数値は次の形式をとります。

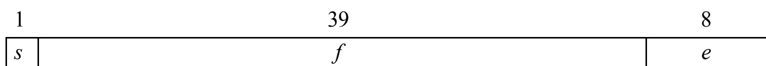
$+/- 仮数 \times 2^{指数}$

仮数では 2 進数の小数点の左に 1 ビットあります（つまり，$0 <= 仮数 < 2$）。

以降の各図では，最上位ビットを左側に，最下位ビットを右側に示します。上にある数字は，各フィールドの幅（ビット）を示しています。左端の項目が最上位アドレスに格納されます。たとえば，Real48 の値の場合，e（指数）が最初のバイトに，f（仮数）が次の 5 バイトに，s（符号）が最後のバイトの最上位ビットに格納されます。

Real48 型

6 バイト（48 ビット）の Real48 型の値は 3 つのフィールドからなります。



$0 < e <= 255$ の場合，この数の値 v は次のようにして決まります。

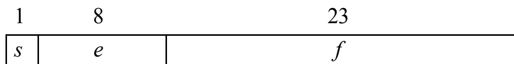
$$v = (-1)^s \times 2^{(e-129)} \times (1.f)$$

$e = 0$ ならば，$v = 0$ です。

Real48 型では，非正規化数，非数（NaN），無限大は格納できません。非正規化数を Real48 型で格納すると 0 になります。非数と無限大を Real48 型で格納しようとするとオーバーフローエラーになります。

Single 型

4 バイト（32 ビット）の Single 型の数は 3 つのフィールドからなります。



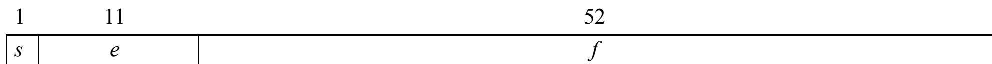
この数の値 v は次のようにして決まります。

$0 < e < 255$ ならば，$v = (-1)^s \times 2^{(e-127)} \times (1.f)$

$e = 0$ かつ $f <> 0$ ならば，$v = (-1)^s \times 2^{(-126)} \times (0.f)$
$e = 0$ かつ $f = 0$ ならば，$v = (-1)^s \times 0$
$e = 255$ かつ $f = 0$ ならば，$v = (-1)^s \times Inf$
$e = 255$ かつ $f <> 0$ ならば，v は NaN

Double 型

8 バイト（64 ビット）の Double 型の数は 3 つのフィールドからなります。



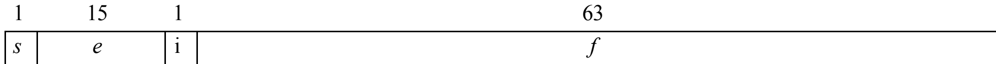
この数の値 v は次のようにして決まります。

$0 < e < 2047$ ならば，$v = (-1)^s \times 2^{(e-1023)} \times (1.f)$

$e = 0$ かつ $f <> 0$ ならば，$v = (-1)^{s} \times 2^{(-1022)} \times (0.f)$
$e = 0$ かつ $f = 0$ ならば，$v = (-1)^{s} \times 0$
$e = 2047$ かつ $f = 0$ ならば，$v = (-1)^{s} \times Inf$
$e = 2047$ かつ $f <> 0$ ならば，v は NaN

Extended 型

10 バイト（80 ビット）の Extended 型の数は 4 つのフィールドからなります。



この数の値 v は次のようにして決まります。

$0 <= e < 32767$ ならば，$v = (-1)^{s} \times 2^{(e-16383)} \times (i.f)$

$e = 32767$ かつ $f = 0$ ならば，$v = (-1)^{s} \times Inf$
$e = 32767$ かつ $f <> 0$ ならば，v は NaN

Comp 型

8 バイト（64 ビット）の Comp 型の数は符号付き 64 ビット整数として格納されます。

通貨型

8 バイト（64 ビット）の通貨型の数は，最下位 4 桁が暗黙に小数点以下 4 桁を表す位取りの符号付き 64 ビット整数として格納されます。

ポインタ型

ポインタ型は 4 バイトの 32 ビットアドレスとして格納されます。ポインタ値 **nil** はゼロとして格納されます。

短い文字列型

文字列はその文字列の最大長に 1 を加えた数のバイトを占有します。先頭バイトに文字列の現在の動的な長さが入り，残りのバイトに文字列の文字が入ります。

長さを示すバイトと各文字は符号なしの値とみなされます。文字列型の最大長は，符号なしバイトの最大値である 255 文字に，長さを示すための 1 バイトを加えた 256 バイト（**string**[255]）となります。

長い文字列型

長い文字列型変数は 4 バイトのメモリを占有し，そこには動的に割り当てられた文字列へのポインタが入っています。長い文字列型変数が空である（長さゼロの文字列が中に入っている）場合，文字列ポインタは **nil** になり，その文字列変数へ動的メモリは割り当てられません。文字列値が空でない場

合，文字列ポインタは動的メモリブロックを指し，そのメモリブロックには文字列値のほかに 32 ビットの文字列の長さと 32 ビットの参照カウンタが入っています。次の表に長い文字列メモリブロックのレイアウトを示します。

表 11.1　長い文字列メモリブロックのレイアウト

オフセット	内容
-8	32 ビットの参照カウンタ
-4	長さ（バイト数）
0..Length - 1	文字列
Length	NULL 文字

長い文字列メモリブロックの末尾にある NULL 文字はコンパイラと組み込み文字列処理ルーチンによって自動的に保守されます。これにより，長い文字列をヌルで終わる文字列へ直接，型キャストできます。

文字列定数とリテラルの場合，コンパイラは動的に割り当てられた文字列と同じレイアウトで参照カウンタが -1 のメモリブロックを生成します。長い文字列型変数へ文字列定数を代入した場合，その文字列定数用に生成されたメモリブロックのアドレスが文字列ポインタへ代入されます。組み込み文字列処理ルーチンでは参照カウンタが -1 のブロックは変更されません。

ワイド文字列型

メモ　Linux では，ワイド文字列は長い文字列とまったく同じように実装されています。

Windows では，ワイド文字列型変数は動的に割り当てられた文字列へのポインタが入っている 4 バイトのメモリを占有します。ワイド文字列型変数が空の場合（長さゼロの文字列が入っている場合），文字列ポインタは **nil** で，その文字列型変数へ動的メモリは関連付けられません。空でない文字列値の場合，文字列ポインタは動的に割り当てられたメモリブロックをポイントし，そのメモリブロックには文字列値と 32 ビットの長さインジケータが入っています。次の表に，Windows でのワイド文字列メモリブロックのレイアウトを示します。

表 11.2　ワイド文字列動的メモリのレイアウト（Windows ）

オフセット	内容
-4	長さ（バイト数）
0 ～ 長さ - 2	文字列
長さ - 1 ～ 長さ	NULL ワイド文字（2 バイト）

Linux では，ワイド文字列は長い文字列とほとんど同じレイアウトを持ちます。

表 11.3　ワイド文字列メモリブロックのレイアウト（Linux）

オフセット	内容
-8	32 ビットの参照カウンタ
-4	長さ（バイト数）
0 ～ 長さ - 2	文字列
長さ - 1 ～ 長さ	NULL ワイド文字（2 バイト）

文字列長はバイト数を表すので，ワイド文字列の長さは最大で文字数の 2 倍になります。

ワイド文字列メモリブロックの末尾にある NULL ワイド文字はコンパイラと組み込み文字列処理ルーチンによって自動的に保守されます。NULL ワイド文字の存在により，ワイド文字列をヌルで終わるワイド文字列へ直接，型キャストできます。

集合型

集合はビットの配列で，各ビットはその集合に各要素が含まれるかどうかを示します。集合の要素の最大数は 256 です。したがって，集合が 32 バイトを超えることはありません。集合のバイト数は次の式で計算します。

(Max div 8) - (Min div 8) + 1

Min と Max は集合の基本型の下限と上限です。特定の要素 E が何バイト目かは次の式でわかります。

(E div 8) - (Min div 8)

そのバイト内で何ビット目かは次の式でわかります。

E mod 8

E は要素の順序値を表します。可能であればコンパイラは CPU レジスタに集合を格納しますが，集合が汎用整数型より大きい場合，または集合のアドレスを取るコードがプログラムに含まれている場合は，集合は常にメモリに格納されます。

静的配列型

静的配列はその配列の要素型の連続した変数の並びとして格納されます。添字の最も小さい要素が最下位のメモリアドレスで格納されます。多次元配列は右端の次元が先に増加するものとして格納されます。

動的配列型

動的配列変数は動的に割り当てられた配列へのポインタが入っている 4 バイトのメモリを占有します。変数が空の場合（初期化されていない場合），または長さゼロの配列が格納されている場合，ポインタは **nil** で，変数へ動的メモリは関連付けられません。配列が空でない場合，変数は動的メモリブロックを指し，そのメモリブロックには配列のほかに 32 ビットの長さインジケータと 32 ビットの参照カウンタが入っています。次の表に動的配列メモリブロックのレイアウトを示します。

表 11.4　動的配列のメモリレイアウト

オフセット	内容
-8	32 ビットの参照カウンタ
-4	32 ビットの長さインジケータ（単位は要素数）
0 ～長さ × (要素数) - 1	配列の要素

レコード型

レコード型を **{$A+}** 状態（デフォルト）で宣言し，しかもその宣言に **packed** 修飾子が含まれていない場合はアンパックレコード型となり，レコードの各フィールドは CPU が効率的にアクセスできるようにアラインメントされます。アラインメントは，各フィールドの型によって，またフィールドがいっしょに宣言されたかどうかによって制御されます。すべてのデータ型に固有なアラインメントマスクがあり，このマスクはコンパイラによって自動的に計算されます。アラインメントマスクは 1，2，4，8 のいずれかに設定でき，その型の値をもっとも効率的なアクセスのために格納しなければならないバイト境界を表します。次の表にすべてのデータ型に関するアラインメントマスクを示します。

表 11.5　型アラインメントマスク

型	アラインメントマスク
順序型	型のサイズ（1，2，4，8 のいずれか）
実数型	Real48 型の場合は 2，Single 型の場合は 4，Double 型と Extended 型の場合は 8
短い文字列型	1
配列型	配列の要素型と同じ
レコード型	レコード内のフィールドのもっとも大きいアラインメントマスク
集合型	1，2，4 の場合は型のサイズ，それ以外の場合は 1
その他のすべての型	$A 指令によって決まる

アンパックレコード型に入っている各フィールドを正しくアラインメントするため，コンパイラは必要ならアラインメントマスクが 2 のフィールドの前に未使用の 1 バイトを挿入し，アラインメントマスクが 4 のフィールドの前に 3 バイトまでの未使用バイトを挿入します。最後に，コンパイラはそのレコードの合計サイズを，いずれかのフィールドの最大のアラインメントマスクによって指定されたバイト境界まで切り上げます。

2 つのフィールドが共通の型指定を使用している場合，宣言に **packed** 修飾子が含まれておらず，レコードが **{$A-}** 状態で宣言されていなくても，それらのフィールドはパックされます。したがって，たとえば次のように宣言されている場合，

```
type
  TMyRecord = record
    A, B: Extended;
    C: Extended;
  end;
```

A と B は，同じ型指定を使用しているので，両者はパックされ，バイト境界に配置されます。コンパイラは，レコードの未使用のバイトをパディングして，C がクォッドワード（8 バイト）境界に配置されるようにします。

レコード型を **{$A-}** 状態で宣言した場合，または宣言に **packed** 修飾子が含まれている場合，レコードの各フィールドはアラインメントされず，それらのフィールドへ連続したオフセットが代入されます。そのようなパックレコードの合計サイズは，単にすべてのフィールドのサイズとなります。データアラインメントは変更されることがあるので，レコード構造をディスクに書き込む場合や，違うバージョンのコンパイラでコンパイルされた他のモジュールにメモリ上でレコードを渡す場合には，レコードをパックすることをお勧めします。

ファイル型

ファイル型はレコードとして表現されます。型付きファイルと型なしファイルは 332 バイトを占有し，次のようなレイアウトになっています。

```
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..259] of Char; );
  end;
```

テキストファイルは 460 バイトを占有し，次のようなレイアウトになっています。

```
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..259] of Char;
    Buffer: TTextBuf;
  end;
```

Handle にはファイルのハンドル（ファイルが開いているときのハンドル）が入ります。

Mode フィールドには次の値のいずれかが入ります。

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

Closed はファイルが閉じていることを示し，fmInput と fmOutput はリセットされたテキストファイルか（fmInput），再書き込みされたテキストファイルか（fmOutput）を示し，fmInOut はファイル変数が型付きか，またはリセットされたか再書き込みされた型なしファイルであることを示します。その

他の値は，ファイル変数に対して割り当てが実行されていない（したがって初期化されていない）ことを示します。

UserData フィールドは，ユーザーが作成したルーチンでデータを格納するために利用できます。

Name にはファイル名が入ります。ファイル名は文字の並びで，最後はヌル文字（#0）で終わります。

型付きファイルと型なしファイルの場合，RecSize にはバイト単位のレコード長が入ります。Private フィールドは使われていませんが予約されています。

テキストファイルの場合，BufPtr は BufSize バイトのバッファへのポインタです。BufPos はそのバッファで次に読み書きする文字の添字です。BufEnd はバッファ内の有効な文字の数です。OpenFunc，InOutFunc，FlushFunc，CloseFunc はファイルを制御する入出力ルーチンへのポインタです。詳細は，8-5 ページの「デバイス関連関数」を参照してください。Flags は，改行の形式を次のように指定します。

ビット 0 クリア	LF で改行
ビット 0 セット	CRLF で改行

Flags の他のビットは，すべて将来の使用のために予約されています。DefaultTextLineBreakStyle および SetLineBreakStyle も参照してください。

手続き型

手続きポインタは，手続きまたは関数のエントリポイントへの 32 ビットポインタとして格納されます。メソッドポインタは，メソッドのエントリポイントへの 32 ビットポインタと，オブジェクトへの 32 ビットポインタとして格納されます。

クラス型

クラス型の値は，クラスのインスタンス（オブジェクト）への 32 ビットポインタとして格納されます。オブジェクトの内部データ形式はレコードの内部データ形式と似ています。オブジェクトのフィールドは，連続した変数の並びとして宣言の順に格納されます。フィールドは常に位置揃えされ，非パックのレコード型に相当します。上位クラスから継承されたフィールドは，下位クラスで新たに定義されたフィールドより前に格納されます。

どのオブジェクトでも，最初の 4 バイトフィールドはクラスの VMT（仮想メソッドテーブル）へのポインタです。VMT はクラスごとに 1 つだけです（オブジェクトごとに 1 つではありません）。異なるクラス型は，それがどれほど似ていても，VMT を共有することはありません。VMT はコンパイラによって自動的に作成され，プログラムで直接操作することはありません。VMT へのポインタは，作成するオブジェクトのコンストラクタメソッドによって自動的に格納され，同様にプログラムで直接操作することはありません。

VMT のレイアウトを次の表に示します。正のオフセットでは，VMT には 32 ビットメソッドポインタが入ります。このメソッドポインタはクラス型のユーザー定義の仮想メソッドと 1 対 1 に対応しており，仮想メソッドの宣言の順に並びます。各スロットには，対応する仮想メソッドのエントリポイントのアドレスが入ります。このレイアウトは，C++ の v-table および COM と互換性があります。

負のオフセットでは，VMT には Object Pascal の処理系の内部で使う多数のフィールドが入ります。Object Pascal の今後の処理系ではレイアウトの変更が見込まれるため，アプリケーションでこの情報について問い合わせる場合は，TObject で定義されているメソッドを使うようにしてください。

表 11.6　仮想メソッドテーブルのレイアウト

オフセット	型	説明
-76	Pointer	仮想メソッドテーブルへのポインタ（または **nil**）
-72	Pointer	インターフェーステーブルへのポインタ（または **nil**）
-68	Pointer	オートメーション情報テーブルへのポインタ（または **nil**）
-64	Pointer	インスタンス初期化テーブルへのポインタ（または **nil**）
-60	Pointer	型情報テーブルへのポインタ（または **nil**）
-56	Pointer	フィールド定義テーブルへのポインタ（または **nil**）
-52	Pointer	メソッド定義テーブルへのポインタ（または **nil**）
-48	Pointer	動的メソッドテーブルへのポインタ（または **nil**）
-44	Pointer	クラス名を含む短い文字列へのポインタ
-40	Cardinal	インスタンスのサイズ（単位バイト）
-36	Pointer	上位クラスへのポインタへのポインタ（または **nil**）
-32	Pointer	SafecallException メソッドのエントリポイントへのポインタ（または **nil**）
-28	Pointer	AfterConstruction メソッドのエントリポイント
-24	Pointer	BeforeDestruction メソッドのエントリポイント
-20	Pointer	Dispatch メソッドのエントリポイント
-16	Pointer	DefaultHandler メソッドのエントリポイント
-12	Pointer	NewInstance メソッドのエントリポイント
-8	Pointer	FreeInstance メソッドのエントリポイント
-4	Pointer	Destroy デストラクタのエントリポイント
0	Pointer	1 番目のユーザー定義仮想メソッドのエントリポイント
4	Pointer	2 番目のユーザー定義仮想メソッドのエントリポイント
⋮	⋮	⋮

クラス参照型

クラス参照型の値は，クラスの仮想メソッドテーブル（VMT）への 32 ビットポインタとして格納されます。

バリアント型

バリアント型は型コードと，その型コードによって与えられた型の値（または値への参照）とが入った 16 バイトのレコードとして格納されます。System ユニットおよび Variants ユニットでは次の定数と型をバリアントで定義しています。

TVarData 型は，バリアント型変数の内部構造を表します（Windows でのバリアント型変数は，COM および Win32 API で使用される Variant 型と同等です）。TVarData 型をバリアント変数の型キャストに使うと，変数の内部構造へアクセスできます。

TVarData レコードの VType フィールドには，下位 12 ビット（varTypeMask 定数によって定義されたビット）にバリアントの型コードが入っています。さらに，varArray ビットがセットされていると，そのバリアントは配列であり，varByRef ビットがセットされていると，そのバリアントには値でなく参照が入っています。

TVarData レコードの Reserved1，Reserved2，Reserved3 の各フィールドは未使用です。

TVarData レコードの残る 8 バイトの内容は，VType フィールドによって異なります。varArray ビットと varByRef ビットが両方ともセットされていなければ，バリアントには与えられた型の値が入っています。

varArray ビットがセットされている場合，バリアントには配列を定義した TVarArray 構造へのポインタが入っています。各配列要素の型は VType フィールドの varTypeMask ビットによって与えられます。

varByRef ビットがセットされている場合，バリアントには VType フィールド内の varTypeMask ビットと varArray ビットによって与えられた型の値への参照が入っています。

varString 型コードは Delphi/Kylix 専用です。varString 値が入っているバリアントを Delphi 関数以外の関数へ渡してはいけません。Windows では，Delphi のオートメーションサポートは自動的に varString バリアントを varOleStr バリアントへ変換した後，それらを外部関数へパラメータとして渡します。

Linux では，VT_decimal はサポートされていません。

第 12 章

プログラムの制御

この章では，パラメータおよび関数の結果の格納方法と受け渡し方法について説明します。最後に，終了手続きについて説明します。

パラメータと関数の結果

パラメータおよび関数の結果の扱いは，さまざまな要素（呼び出し規約，パラメータの意味，渡される値の型とサイズなど）によって左右されます。

パラメータの受け渡し

パラメータは CPU レジスタかスタックを経由してルーチンに渡されます。どちらを経由するかはルーチンの呼び出し規約によって決まります。呼び出し規約についての詳細は，6-4 ページの「呼び出し規約」を参照してください。

変数パラメータ（**var**）は常に参照渡しです。つまり，実際の格納位置をポイントする 32 ビットポインタによって渡されます。

値パラメータと定数パラメータ（**const**）は，そのパラメータの型とサイズに応じて値渡しまたは参照渡しになります。

- 順序型パラメータは，対応する型の変数と同じ形式を使って 8 ビット，16 ビット，32 ビット，64 ビットのいずれかの値として渡される。
- 実数型パラメータは常にスタックに渡される。Single パラメータは 4 バイトを占有し，Double, Comp，Currency の各パラメータは 8 バイトを占有する。Real48 は 8 バイトを占有し，Real48 値はそのうちの下位 6 バイトに格納される。Extended は 12 バイトを占有し，Extended 値はそのうちの下位 10 バイトに格納される。
- 短い文字列型パラメータは短い文字列への 32 ビットポインタとして渡される。

- 長い文字列型パラメータまたは動的配列パラメータは，長い文字列へ割り当てられた動的メモリブロックへの 32 ビットポインタとして渡されます。空の長い文字列の場合は値 **nil** が渡される。
- ポインタ型，クラス型，クラス参照型，手続きポインタ型の各パラメータは 32 ビットポインタとして渡される。
- メソッドポインタはスタックに 2 つの 32 ビットポインタとして渡される。インスタンスポインタがメソッドポインタより前にプッシュされるので，メソッドポインタが最下位アドレスを占有する。
- **register** 規約および **pascal** 規約の場合，バリアント型パラメータは Variant 値への 32 ビットポインタとして渡される。
- 1 バイト，2 バイト，または 4 バイトの集合，レコード，静的配列は，8 ビット値，16 ビット値，32 ビット値として渡される。これより大きな集合，レコード，静的配列の場合は，値への 32 ビットポインタとして渡される。この規則の例外として，**cdecl** 規約，**stdcall** 規約，**safecall** 規約の場合，レコードは常にスタックに直接渡される。このとき渡されるレコードのサイズは，もっとも近いダブルワード境界に切り上げられる。
- オープン配列パラメータは 2 つの 32 ビット値として渡される。1 番目の値は配列データへのポインタで，2 番目の値は配列の要素数から 1 を引いた値となる。

2 つのパラメータがスタックに渡されると，それぞれのパラメータは 4 バイトの倍数（ダブルワードの倍数）を占有します。8 ビットまたは 16 ビットのパラメータの場合，パラメータは 1 バイトか 1 ワードしか占有しませんが，ダブルワードとして渡されます。このダブルワードの未使用部分の内容は未定義です。

pascal，**cdecl**，**stdcall**，**safecall** の各規約では，すべてのパラメータがスタック上に渡されます。**pascal** 規約の場合，パラメータは宣言された順序（左から右）でプッシュされるので，第 1 パラメータは最終的に最上位アドレスに置かれ，最後のパラメータは最終的に最下位アドレスに置かれます。**cdecl** 規約，**stdcall** 規約および **safecall** 規約の場合，パラメータは宣言と逆の順序（右から左）でプッシュされるので，第 1 パラメータは最終的に最下位アドレスに置かれ，最後のパラメータは最終的に最上位アドレスに置かれます。

register 規約では最大 3 つのパラメータが CPU レジスタに渡され，残りは（もしあれば）スタックに渡されます。これらのパラメータは宣言された順序で渡され（**pascal** 規約と同じ），適格な最初の 3 つのパラメータが EAX，EDX，ECX の各レジスタにこの順序で渡されます。実数型，メソッドポインタ型，バリアント型，Int64 型，および構造化型はレジスタパラメータとして扱われません。ほかのパラメータは適格なレジスタパラメータとして扱われます。3 つを超えるパラメータがレジスタパラメータとして適格な場合は，最初の 3 つが EAX，EDX，ECX に渡され，残りのパラメータは宣言された順序でスタック上にプッシュされます。たとえば，次のような宣言があるとします。

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

Test を呼び出すと，A が 32 ビット整数として EAX に，B が Char へのポインタとして EDX に，D が長い文字列メモリブロックへのポインタとして ECX に渡され，C と E は，2 つのダブルワードと 32 ビットポインタとしてこの順序でスタックにプッシュされます。

レジスタ保存規約

手続きと関数では EBX，ESI，EDI，EBP の各レジスタの値を保持しなければなりませんが，EAX，EDX，ECX の各レジスタは自由に変更できます。コンストラクタまたはデストラクタをアセンブラで実装する場合は，必ず DL レジスタを保持するようにしてください。手続きと関数は CPU のディレクションフラグがクリアされている（CLD 命令に対応）状態で常に呼び出され，戻り時にも常にディレクションフラグがクリアされていなければなりません。

関数の結果

関数の結果値の戻りには以下の規約が使われます。

- 順序型の関数結果は，可能であれば CPU レジスタへ返される。つまり，バイトは AL へ，ワードは AX へ，ダブルワードは EAX へ返される。
- 実数型の関数結果は浮動小数点コプロセッサのスタックトップレジスタ（ST (0)）へ返される。通貨型の関数結果の場合，ST(0) 内の値に 10000 がかけられる。たとえば，通貨型の値 1.234 は 12340 として ST(0) の中に返される。
- 文字列，動的配列，メソッドポインタ，バリアント型，または Int64 型の関数結果の場合は，その関数結果が，宣言されたパラメータに続く追加 **var** パラメータとして宣言されたものとして扱われる。つまり，呼び出し側は関数結果を返す先の変数をポイントする追加の 32 ビットポインタを渡す。
- ポインタ型，クラス型，クラス参照型，手続きポインタ型の関数結果は EAX へ返される。
- 静的配列型，レコード型，集合型の関数結果の場合，値が 1 バイトを占有するときは AL に，値が 2 バイトを占有するときは AX に，値が 4 バイトを占有するときは EAX に結果が返される。その他の場合は，宣言されたパラメータの後に関数に渡される追加の **var** パラメータに結果が返される。

メソッド呼び出し

メソッドは，暗黙の追加パラメータ Self を持つ点を除いて，通常の手続きや関数と同じ呼び出し規約に従います。Self は，メソッドを呼び出したインスタンスまたはクラスへの参照です。Self パラメータは 32 ビットポインタとして渡されます。

- **register** 規約の場合，Self パラメータはほかのすべてのパラメータより前に宣言されたものとして動作する。したがって，Self パラメータは常に EAX レジスタへ渡される。
- **pascal** 規約の場合，Self パラメータは，（関数結果のために渡されることがある追加の **var** パラメータも含めて）ほかのすべてのパラメータより後に宣言されたものとして動作する。したがって，Self パラメータは常に最後にプッシュされ，最終的にはほかのすべてのパラメータより下位のアドレスに置かれる。
- **cdecl** 規約，**stdcall** 規約および **safecall** 規約の場合，Self パラメータはほかのすべてのパラメータより前に，ただし関数結果のために渡されることがある追加の **var** パラメータより後に宣言されたものとして動作する。したがって，Self パラメータは，追加の **var** パラメータを例外として，最後にプッシュされる。

コンストラクタとデストラクタ

コンストラクタとデストラクタは，その呼び出しのコンテキストを示すために追加の Boolean フラグパラメータが渡される点を除けば，ほかのメソッドと同じ呼び出し規約を使います。

コンストラクタ呼び出しのフラグパラメータの値が偽の場合，そのコンストラクタがインスタンスオブジェクトまたはキーワード **inherited** によって呼び出されたことを示します。その場合，コンストラクタは通常のメソッドのように動作します。コンストラクタ呼び出しのフラグパラメータの値が真の場合は，そのコンストラクタがクラス参照によって呼び出されたことを示します。その場合，コンストラクタは Self で与えられたクラスのインスタンスを作成し，新しく作成されたオブジェクトへの参照を EAX に返します。

デストラクタ呼び出しのフラグパラメータの値が偽の場合は，そのデストラクタがキーワード **inherited** によって呼び出されたことが示されます。その場合，デストラクタは通常のメソッドのように動作します。デストラクタ呼び出しのフラグパラメータの値が真の場合は，そのデストラクタがインスタンスオブジェクトによって呼び出されたことを示します。この場合，デストラクタは Self で与えられたインスタンスを戻りの直前に割り当て解除します。

フラグパラメータは，ほかのすべてのパラメータより前に宣言されたものとして動作します。**register** 規約の場合は，DL レジスタに渡されます。**pascal** 規約の場合は，ほかのどのパラメータよりも前にプッシュされます。**cdecl**，**stdcall**，**safecall** の各規約では，Self パラメータの直前にプッシュされます。

DL レジスタは，そのコンストラクタまたはデストラクタが呼び出しスタック内で最も外側であるかどうかを示しているので，終了する前に DL の値を復旧して，BeforeDestruction または AfterConstruction が適切に呼び出されるようにしなければなりません。

終了手続き

終了手続きを使うと，プログラムが終了する前に確実に特定の処理（ファイルの更新やクローズなど）を実行できます。ポインタ変数 ExitProc を使うと，終了手続きの「組み込み」ができます。終了手続きは，正常終了，Halt の呼び出しによる強制終了，実行時エラーによる終了のいずれかを問わず，プログラムの終了時に必ず呼び出されます。終了手続きはパラメータをとりません。

メモ 終了時の処理には，終了手続きではなく，終了処理部の使用をお勧めします（3-5 ページの「終了処理部」を参照）。終了手続きは，実行可能ファイル，共有オブジェクト（Linux）または .DLL（Windows）のターゲットでのみ使用できます。パッケージの場合は，終了処理部に終了動作を実装しなければなりません。終了手続きはすべて，終了処理部が実行される前に呼び出されます。

プログラムのほか，ユニットも終了手続きを組み込むことができます。ユニットの場合，終了手続きは初期化コードの一部として組み込むことができ，ファイルを閉じるといったクリーンアップ処理の実行を終了手続きにまかせることができます。

正しく実装された終了手続きは，一連の終了手続き連鎖の一部となります。終了手続きは組み込みと逆の順序で実行されます。したがって，あるユニットの終了コードがそれに依存しているほかのユニットの終了コードより先に実行されることはありません。終了手続きの連鎖を機能させるには，終

了手続きのアドレスをポイントする前に ExitProc の内容を保存しておく必要があります。と同時に，終了手続きの最初の文で，保存した ExitProc の値を組み込み直す必要があります。

次のコードは終了手続きの実装の概略を示しています。

```
var
  ExitSave: Pointer;
procedure MyExit;
begin
  ExitProc := ExitSave;  // 必ず古いベクターから先に復元する
  ⋮
end;
begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  ⋮
end.
```

エントリコードによって ExitProc の内容が ExitSave に保存され，MyExit 手続きが組み込まれます。終了プロセスの一部として呼び出された後で，MyExit は最初に前の終了手続きを再度組み込みます。

ランタイムライブラリの終了ルーチンは ExitProc が **nil** になるまで終了手続きを呼び出し続けます。無限ループを避けるために，呼び出しの前に毎回 ExitProc は **nil** に設定されます。したがって，現在の終了手続きで ExitProc にアドレスが代入されたときに限り次の終了手続きが呼び出されます。終了手続きでエラーが発生した場合，その終了手続きが再び呼び出されることはありません。

終了手続きでは，整数型変数の ExitCode とポインタ変数の ErrorAddr を調べることにより，終了の原因がわかります。正常終了の場合，ExitCode はゼロになり，ErrorAddr は **nil** になります。Halt の呼び出しによる終了の場合，ExitCode は Halt へ渡された値になり，ErrorAddr は **nil** になります。実行時エラーによる終了の場合は，ExitCode にエラーコードが格納され，ErrorAddr にはエラー文のアドレスが格納されます。

最後の終了手続き（ランタイムライブラリによって組み込まれる終了手続き）は Input ファイルと Output ファイルを閉じます。ErrorAddr が **nil** 以外の場合は，実行時エラーメッセージが出力されます。独自の実行時エラーメッセージを出力したい場合は，終了手続きで ErrorAddr の値を調べ，これが **nil** 以外の場合にメッセージを出力します。戻る前には，ErrorAddr を **nil** に設定し，ほかの終了手続きでエラーが再度報告されないようにしてください。

すべての終了手続きを呼び出すと，ランタイムライブラリはオペレーティングシステムに制御を戻し，ExitCode の値を戻りコードとして渡します。

第 13 章

インラインアセンブラコード

組み込みアセンブラを使うと，アセンブラコードを Object Pascal プログラム内に直接記述できます。以下の特徴を持っています。

- インラインアセンブルが可能
- Intel Pentium III，Intel MMX エクステンション，SSE（Streaming SIMD Extensions），および AMD Athlon（3D Now! を含む）のすべての命令をサポート
- マクロサポートはないが，純粋なアセンブラ関数プロシージャをサポート
- アセンブラ文で Object Pascal の識別子（定数，型，変数など）が使用可能

組み込みアセンブラを使用するかわりに，外部の手続きや関数が含まれるオブジェクトファイルをリンクするという方法もあります。詳細については，6-6 ページの「オブジェクトファイルのリンク」を参照してください。

メモ アプリケーションで使用したい外部アセンブラコードがある場合は，それを Object Pascal で書き直すか，少なくともインラインアセンブラを使って再実装することを検討する必要があります。

asm 文

組み込みアセンブラには **asm** 文を使ってアクセスします。**asm** 文の構造を次に示します。

`asm` statementList `end`

statementList の部分にアセンブラ文を記述します。複数のアセンブラ文を記述する場合は，セミコロン，改行，または Object Pascal のコメントで区切ります。

コメントは **asm** 文中でも Object Pascal スタイルで記述しなければなりません。セミコロンには，その行の残りがコメントであるという意味はありません。

予約語 **inline** と **assembler** 指令は，下位互換性を保つ目的でのみ残されています。コンパイラはこれらを無視します。

レジスタの使用

全般的には **asm** 文で使用されるレジスタの規則は **external** 手続きまたは関数の規則と同じです。**asm** 文は EDI，ESI，ESP，および EBX レジスタを保持しなければなりませんが，EAX，ECX，および EDX レジスタは自由に変更できます。**asm** 文のエントリで，BP は現在のスタックフレームを指し，SP はスタックの最上位を指し，SS はスタックセグメントのセグメントアドレスを含み，DS はデータセグメントのセグメントアドレスを含みます。ESP と EBP を除いて，**asm** 文は文のエントリでのレジスタの内容については関知しません。

アセンブラ文の構文

アセンブラ文の構文は次のとおりです。

Label：Prefix Opcode $Operand_1$，$Operand_2$

Label にはラベル識別子を，Prefix にはアセンブラのプレフィクスオペコードを，Opcode にはアセンブラ命令のオペコードまたは疑似命令を，Operand にはアセンブラ式を指定します。Label と Prefix はオプションです。オペコードの中には，オペランドを 1 つしかとらないものや，まったくとらないものもあります。

コメントはアセンブラ文とアセンブラ部の間には記述できますが，アセンブラ文の内部には記述できません。次に例を示します。

```
MOV AX,1 { 初期値 }        { OK }
MOV CX,100 { カウント }    { OK }

MOV { 初期値 } AX,1;       { エラー！ }
MOV CX, { カウント } 100   { エラー！ }
```

ラベル

ラベルは，組み込みアセンブラでも Object Pascal の場合と同じように，文の前にラベルとコロンを付加する形で付けます。ラベルの長さには制限はありません。Object Pascal の場合と同じように，ラベルは **asm** 文が含まれるブロック内の **label** 宣言部で宣言しなければなりません。ただし，ローカルラベルにはこの規則はあてはまりません。

ローカルラベルは記号 **@** で始まります。ローカルラベルは，**@** の後に英字，数字，下線，**@** が 1 つまたは複数続いた形になります。ローカルラベルは **asm** 文でしか使用できません。また，ローカルラベルのスコープは，予約語 **asm** から，その **asm** 文の終わりまでになります。ローカルラベルは宣言の必要はありません。

命令コード

組み込みアセンブラは，Intel が一般アプリケーションでの使用のために公開しているすべてのオペコードをサポートしています。オペレーティングシステムの特権命令はサポートされない場合があることに注意してください。以下のプロセッサファミリーの命令はサポートされています。

- Pentium ファミリー
- Pentium Pro，Pentium II
- Pentium III
- Pentium 4

メモ　Pentium 4 の命令は，Windows でのみサポートされます。

また，以下の命令セットもサポートされています。

- AMD 3DNow!（AMD K6 以降）
- AMD Enhanced 3DNow（AMD Athlon 以降）

各命令についての詳細は，使用しているマイクロプロセッサのマニュアルを参照してください。

RET 命令のサイズ

RET オペコードは常に near のリターンコードを生成します。

自動ジャンプサイズ

特に指定しない限り，組み込みアセンブラは最も効率的なジャンプ命令の形式を自動的に選択することによって，ジャンプ命令を最適化します。ターゲットがラベルの場合（手続きや関数でない場合），この自動ジャンプサイズは無条件ジャンプ命令（JMP）とすべての条件ジャンプ命令に適用されます。

無条件ジャンプ命令（JMP）では，ターゲットラベルまでの距離が -128 ～ 127 バイトまでの場合，組み込みアセンブラは short ジャンプ（1 バイトの変位が後に続く 1 バイトのオペコード）を生成し，それ以外の場合は near ジャンプ（2 バイトの変位が後に続く 1 バイトのオペコード）を生成します。

条件ジャンプ命令では，ターゲットラベルまでの距離が -128 ～ 127 バイトまでの場合 short ジャンプ（1 バイトの変位が後に続く 1 バイトのオペコード）を生成します。それ以外の場合，組み込みアセンブラは short inverse ジャンプを生成し，near ジャンプを越えてターゲットラベルまでジャンプします（合計で 5 バイト）。次のアセンブラ文を見てください。

```
JC      Stop
```

ここで short ジャンプの範囲外の Stop は，次の文に対応する機械語シーケンスに変換されます。

```
JNC      Skip
JMP      Stop
Skip:
```

手続きおよび関数のエントリポイントへのジャンプは常に near です。

アセンブラ疑似命令

組み込みアセンブラは DB（バイト定義），DW（ワード定義），DD（ダブルワード定義）の 3 つのデータ定義疑似命令をサポートしています。各疑似命令は，疑似命令の後にカンマ区切りで指定されたオペランドに対応するデータを生成します。

DB 疑似命令はバイトシーケンスを生成します。各オペランドには，-128 ～ 255 の範囲の値となる定数式，または任意の長さの文字列を指定できます。定数式は 1 バイトのコードを生成し，文字列は各文字の ASCII コードに対応するバイトシーケンスを生成します。

DW 疑似命令はワードシーケンスを生成します。各オペランドには，-32,768 〜 65,535 の範囲の値となる定数式，またはアドレス式を指定できます。アドレス式を指定した場合，組み込みアセンブラは **near** ポインタ（アドレスのオフセット部を含むワード）を生成します。

DD 疑似命令はダブルワードシーケンスを生成します。各オペランドには，-2,147,483,648 〜 4,294,967,295 の範囲の値となる定数式，またはアドレス式を指定できます。アドレス式を指定した場合，組み込みアセンブラは，後ろにアドレスのセグメント部を含むワードが続く **far** ポインタ（アドレスのオフセット部を含むワード）を生成します。

DQ 疑似命令は，Int64 値用のクオッドワード（8 バイト）を定義します。

DB，DW，DD の各疑似命令によって生成されるデータは，ほかの組み込みアセンブラ文で生成されたコードと同様，常にコードセグメントに格納されます。未初期化または初期化済みのデータをデータセグメントに生成するには，通常の Object Pascal の **var** 宣言または **const** 宣言を使います。

以下に DB，DW，および DD 疑似命令の例を示します。

```
asm
  DB      0FFH                          { 1 バイト }
  DB      0, 99                         { 2 バイト }
  DB      'A'                           { Ord ('A')}
  DB      'Hello world...', 0DH, 0AH    { 文字列 + CR/LF }
  DB      12,"string"                   { Object Pascal スタイルの文字列 }
  DW      0FFFFH                        { 1 ワード }
  DW      0, 9999                       { 2 ワード }
  DW      'A'                           { DB 'A', 0 と同じ }
  DW      'BA'                          { DB 'A', 'B' と同じ }
  DW      MyVar                         { MyVar のオフセット }
  DW      MyProc                        { MyProc のオフセット }
  DD      0FFFFFFFFH                    { 1 ダブルワード }
  DD      0, 999999999                  { 2 ダブルワード }
  DD      'A'                           { DB 'A', 0, 0, 0 と同じ }
  DD      'DCBA'                        { DB 'A', 'B', 'C', 'D' と同じ }
  DD      MyVar                         { MyVar へのポインタ }
  DD      MyProc                        { MyProc へのポインタ }
end;
```

Turbo Assember では，識別子の後ろに DB，DW，または DD 疑似命令がある場合，疑似命令の位置でバイト，ワード，またはダブルワードサイズの変数を宣言します。たとえば Turbo Assembler では次のような宣言が可能です。

```
ByteVar   DB    ?
WordVar   DW    ?
IntVar    DD    ?
  ⋮
            MOV   AL,ByteVar
            MOV   BX,WordVar
            MOV   ECX,IntVar
```

組み込みアセンブラは，このような変数の宣言をサポートしません。インラインアセンブラ文で定義可能なシンボルの種類はラベルだけです。変数はすべて Object Pascal 構文を使って宣言しなければなりません。したがって，上記のコードは次のように記述します。

```
var
  ByteVar: Byte;
  WordVar: Word;
```

```
    IntVar: Integer;
    ⋮
  asm
    MOV   AL,ByteVar
    MOV   BX,WordVar
    MOV   ECX,IntVar
  end;
```

SMALL および LARGE は，変位のサイズを指定するために使用できます。たとえば,

```
MOV EAX, [LARGE $1234]
```

この命令は，通常の 32 ビット変位（$00001234）の移動を生成しますが,

```
MOV EAX, [SMALL $1234]
```

この命令は，アドレスサイズオーバーライドプレフィクスと 16 ビット変位（$1234）を持つ移動を生成します。

SMALL は，領域を節約するために使用できます。次の例は，アドレスサイズオーバーライドと 2 バイトのアドレス（合計 3 バイト）を生成します。

```
MOV EAX, [SMALL 123]
```

これに対して次の例は,

```
MOV EAX, [123]
```

アドレスサイズオーバーライドなしで 4 バイトのアドレス（合計 4 バイト）を生成します。

VMTOFFSET と DMTINDEX の 2 つの疑似命令によって，アセンブラコードから動的メソッドと仮想メソッドにアクセスすることができます。

VMTOFFSET は，仮想メソッド引数の仮想メソッドポインタテーブル（VMT）エントリの，テーブルの先頭からのオフセットを示すバイト数を取得します。この疑似命令では，パラメータとして完全に修飾されたクラス名付きのメソッド名（たとえば TExample.VirtualMethod）が必要です。

DMTINDEX は，渡された動的メソッドの動的メソッドテーブルインデックスを取得します。この疑似命令でも，パラメータとして完全に修飾されたクラス名付きのメソッド名（たとえば TExample.DynamicMethod）が必要です。動的メソッドを呼び出すには，DMTINDEX で取得した値を (E)SI レジスタに入れて System.@CallDynaInst を呼び出します。

メモ **message** 指令付きのメソッドは，動的メソッドとして実装され，DMTINEDX を使って呼び出すことができます。

```
TMyClass = class
  procedure x; message MYMESSAGE;
end;
```

次の例は DMTINDEX と VMTOFFSET を使って動的メソッドと仮想メソッドにアクセスします。

```
program Project2;

type
  TExample = class
    procedure DynamicMethod; dynamic;
    procedure VirtualMethod; virtual;
  end;

procedure TExample.DynamicMethod;
```

```
begin
end;

procedure TExample.VirtualMethod;
begin
end;

procedure CallDynamicMethod(e: TExample);
asm
  // ESI レジスタを保存
  PUSH    ESI
  // インスタンスポインタは EAX に入れる必要がある
  MOV     EAX, e
  // DMT エントリインデックスは (E)SI に入れる必要がある
  MOV     ESI, DMTINDEX TExample.DynamicMethod
  // ここでメソッドを呼び出す
  CALL    System.@CallDynaInst
  // ESI レジスタをリストア
  POP ESI
end;

procedure CallVirtualMethod(e: TExample);
asm
  // インスタンスポインタは EAX に入れる必要がある
  MOV     EAX, e
  // VMT テーブルエントリを取得
  MOV     EDX, [EAX]
  // ここでオフセット VMTOFFSET のメソッドを呼び出す
  CALL    DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]
end;

var
  e: TExample;
begin
  e := TExample.Create;
  try
    CallDynamicMethod(e);
    CallVirtualMethod(e);
  finally
    e.Free;
  end;
end.
```

オペランド

組み込みアセンブラのオペランドは，定数，レジスタ，シンボル，および演算子で構成される式です。

オペランド内で，以下の予約語はすでに意味が定義されています。

表 13.1　組み込みアセンブラの予約語

AH	BX	DI	EBX	ESP	OFFSET	SP
AL	BYTE	DL	ECX	FS	OR	SS
AND	CH	DS	EDI	GS	PTR	ST

表 13.1　組み込みアセンブラの予約語

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

予約語はユーザー定義の識別子よりも常に優先されます。次に例を示します。

```
var
  Ch: Char;
  ⋮
asm
  MOV     CH, 1
end;
```

このコードでは，Ch 変数ではなく CH レジスタに 1 がロードされます。組み込みアセンブラの予約語と同じ名前のユーザー定義シンボルにアクセスするには，次のようにオーバーライド演算子（**&**）を使います。

```
MOV     &Ch, 1
```

できる限り，ユーザー定義の識別子名には組み込みアセンブラの予約語を使わないようにしてください。

式

組み込みアセンブラは，すべての式を 32 ビットの整数値として評価します。浮動小数点数や，文字列定数以外の文字列値はサポートしていません。

式は式要素と演算子から構成され，それぞれの式は関連する式クラスと式型を持ちます。

Object Pascal とアセンブラの式の相違点

Object Pascal の式とアセンブラの式の大きな相違点は，アセンブラの式は定数値に解決され，その値はコンパイル時に計算できなくてはならないことです。たとえば，次のように宣言されているとします。

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

次の文は有効な文です。

```
asm
  MOV     Z,X+Y
end;
```

X と Y は両方とも定数であるため，式 `X + Y` は定数 30 を記述するための都合のよい方法に過ぎず，この命令は，単に値 30 を変数 Z に代入するだけです。しかし，X と Y が変数の場合，

```
var
```

```
X, Y: Integer;
```

組み込みアセンブラは X + Y の値をコンパイル時に計算できなくなります。このような場合に X と Y の合計値を Z に代入するには，次のようにします。

```
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     Z,EAX
end;
```

Object Pascal の式では，変数を参照するということは，その変数の内容を参照するということです。しかし，アセンブラの式では，変数を参照するということは，その変数のアドレスを参照するということです。Object Pascal では，式 X + 4 で X が変数の場合，X の内容に 4 を加えることを意味しますが，組み込みアセンブラでは，X のアドレスから 4 バイト上位のアドレスにあるワードの内容を意味します。このため，次のようなコードを記述することは可能ですが，

```
asm
  MOV     EAX,X+4
end;
```

このコードでは X の値に 4 を加えたものが EAX に代入されるのではなく，X から 4 バイト上位のアドレスに格納されているダブルワードの値が代入されてしまいます。4 を X の内容に加算する正しい方法を次に示します。

```
asm
  MOV     EAX,X
  ADD     EAX,4
end;
```

式の要素

式の構成要素は，定数，レジスタ，シンボル，演算子です。

定数

組み込みアセンブラは，数値定数と文字列定数の 2 種類の定数をサポートしています。

数値定数

組み込みアセンブラの数値定数は -2,147,483,648 〜 4,294,967,295 の範囲の整数でなくてはなりません。

デフォルトでは，数値定数には 10 進表記を使いますが，組み込みアセンブラでは 2 進，8 進，16 進の各表記もサポートしています。2 進では数値の後に文字 B を指定し，8 進では数値の後に文字 O を指定し，16 進では数値の後に文字 H を指定するか，数値の前に **$** を指定します。

組み込みアセンブラの数値定数は，0 〜 9 までの数字または **$** で始まらなければなりません。そのため，サフィックス H を使って 16 進定数を記述するとき，最初の桁が A 〜 F までの 16 進数字の場合は，数値の前にゼロ（0）を付けなければなりません。たとえば，0BAD4H と $BAD4 は 16 進定数ですが，BAD4H は数字でなく文字で始まるので識別子です。

文字列定数

文字列定数は，単引用符または二重引用符で囲まなければなりません。文字列定数を囲む引用符と同じ種類の 2 つの連続する引用符は 1 文字として数えられます。次に文字列定数の例を示します。

```
'Z'
'Delphi'
'Linux'
"That's all folks"
'"That''s all folks," he said.'
'100'
'"'
"'"
```

DB 疑似命令では任意の長さの文字列定数が使えます。文字列定数を指定すると，文字列を構成する各文字の ASCII コードが含まれるバイトのシーケンスが割り当てられます。DB 疑似命令以外で使う場合，文字列定数は 4 文字以内でなければならず，式の一部として使用できる数値を示します。特定の文字列定数に対応する数値は，次のように計算されます。

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

ここで，Ch1 は右端（最後）の文字を示し，Ch4 は左端（最初）の文字を示します。文字列が 4 文字未満の場合，左端の文字は 0（ゼロ）とみなされます。文字列定数と，それに対応する数値の例を次に示します。

表 13.2　文字列の例とその値

文字列	値
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

レジスタ

以下の予約済みシンボルを使って CPU レジスタを指定します。

表 13.3　CPU レジスタ

32 ビット汎用レジスタ	EAX，EBX，ECX，EDX	32 ビットポインタまたはインデックス	ESP，EBP，ESI，EDI
16 ビット汎用レジスタ	AX，BX，CX，DX	16 ビットポインタまたはインデックス	SP，BP，SI，DI
8 ビット下位レジスタ	AL，BL，CL，DL	16 ビットセグメントレジスタ	CS，DS，SS，ES
		32 ビットセグメントレジスタ	FS GS
8 ビット上位レジスタ	AH，BH，CH，DH	コプロセッサレジスタスタック	ST

オペランドがレジスタ名だけの場合，これをレジスタオペランドと呼びます。すべてのレジスタがレジスタオペランドとして使えます。さらに，いくつかのレジスタはその他の目的にも使えます。

ベースレジスタ（BX と BP）とインデックスレジスタ（SI と DI）を [ ] で囲んで記述すると，インデックス指定ができます。ベースレジスタとインデックスレジスタの正しい組み合わせは [BX], [BP]，[SI]，[DI]，[BX+SI]，[BX+DI]，[BP+SI]，および [BP+DI] です。すべての 32 ビットレジスタについても，[EAX+ECX]，[ESP]，[ESP+EAX+5] のようにインデックス指定ができます。

セグメントレジスタ（ES，CS，SS，DS，FS，GS）はサポートされていますが，通常の 32 ビットアプリケーションではセグメントは使用されません。

シンボル ST は 80x87 浮動小数点レジスタスタック上の最上位のレジスタを表します。8 個の浮動小数点レジスタは ST(X) を使ってそれぞれ参照できます。X は 0 ～ 7 までの定数で，レジスタスタックの最上位からの距離を示します。

シンボル

組み込みアセンブラでは，アセンブラの式の中で定数，型，変数，手続き，関数など Object Pascal の大部分の識別子を使用できます。これに加えて，組み込みアセンブラには，関数の本体で使用される Result 変数に対応する特殊シンボル @Result が用意されています。たとえば，次のような関数があるとします。

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X + Y;
end;
```

この関数は，アセンブラでは次のように記述できます。

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

以下のシンボルは **asm** 文では使えません。

- 標準の手続きと関数（Writeln，Chr など）
- 文字列定数，浮動小数点定数，集合定数（レジスタのロード時は除く）
- 現在のブロックで宣言されていないラベル
- 関数外部の @Result シンボル

次の表に **asm** 文で使用できるシンボルを示します。

表 13.4　組み込みアセンブラが認識するシンボル

シンボル	値	クラス	型
ラベル	ラベルのアドレス	メモリ参照	SHORT
定数	定数の値	即値	0
型	0	メモリ参照	型のサイズ
フィールド	フィールドのオフセット	メモリ	型のサイズ

表 13.4　組み込みアセンブラが認識するシンボル（つづき）

シンボル	値	クラス	型
変数	変数のアドレス	メモリ参照	型のサイズ
手続き	手続きのアドレス	メモリ参照	NEAR
関数	関数のアドレス	メモリ参照	NEAR
ユニット	0	即値	0
@Code	コードセグメントのアドレス	メモリ参照	0FFF0H
@Data	データセグメントのアドレス	メモリ参照	0FFF0H
@Result	Result 変数のオフセット	メモリ参照	型のサイズ

最適化を無効にした場合，ローカル変数（手続きや関数内で宣言された変数）は常にスタック上に割り当てられ，EBP に基づきアクセスされます。ローカル変数シンボルの値は EBP からの符号付きオフセットです。組み込みアセンブラはローカル変数への参照に自動的に [EBP] を追加します。たとえば，次のような宣言があるとします。

```
var Count: Integer;
```

関数または手続き内に次の命令があると，

```
MOV     EAX,Count
```

これは MOV EAX,[EBP-4] となります。

組み込みアセンブラは **var** パラメータを 32 ビットのポインタとして扱い，**var** パラメータのサイズは常に 4 になります。**var** パラメータにアクセスするための構文と値パラメータにアクセスするための構文は異なります。**var** パラメータの内容にアクセスするには，まず 32 ビットポインタをロードしてから，そのパラメータが指している位置にアクセスします。次に例を示します。

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    MOV     EAX,[EAX]
    MOV     EDX,Y
    ADD     EAX,[EDX]
    MOV     @Result,EAX
  end;
end;
```

asm 文の中では，限定子を使って識別子を限定できます。たとえば，次のように宣言されているとします。

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

asm 文の中で，次のような構文を使ってフィールドにアクセスできます。

```
MOV     EAX,P.X
```

```
MOV     EDX,P.Y
MOV     ECX,R.A.X
MOV     EBX,R.B.Y
```

プログラムを記述するとき，型識別子を使って変数を作成できます。以下の各命令は同一の機械語を生成します。生成されるコードは [EDX] の内容を EAX にロードします。

```
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
```

式クラス

組み込みアセンブラは，式をレジスタ，メモリ参照，即値の 3 つのクラスに分類します。

レジスタ式はレジスタ名だけで構成される組み込みアセンブラ式です。レジスタ式には，たとえば EAX，ECX，EDI，ESI などがあります。レジスタ式をオペランドとして使う場合，アセンブラは CPU レジスタを操作する命令を生成します。

メモリ参照はメモリ位置を示す式です。Object Pascal のラベル，変数，型付き定数，手続き，関数はこのカテゴリに属します。

即値はレジスタではなくメモリ位置にも関連付けられない式です。このグループには Object Pascal の型なし定数と型識別子が含まれます。

即値とメモリ参照では，オペランドとして使われた場合に生成されるコードが異なります。次に例を示します。

```
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 は即値であるため，最初の MOV は即値データ転送命令になります。しかし 2 番目の MOV は，Count がメモリ参照であるためメモリ転送命令と解釈されます。3 番目の MOV では，Start をメモリ参照（この場合はデータセグメントのオフセット 10 にあるワード）に変換するのに大カッコが使われています。4 番目の MOV では OFFSET 演算子が Count を即値（データセグメントでの Count のオフセット）に変換するのに使われています。

大カッコと OFFSET 演算子は互いに補完しあっています。次の **asm** 文からは，前の **asm** 文の最初の 2 行と完全に同一の機械語が生成されます。

```
asm
  MOV     EAX,OFFSET [Start]
  MOV     EBX,[OFFSET Count]
end;
```

メモリ参照と即値は，さらに再配置可能な式と絶対式のどちらかに分類されます。再配置はリンカがシンボルに絶対アドレスを割り当てるプロセスです。再配置可能な式とはリンク時に再配置を必要とする値を指し，絶対式とは再配置を必要としない値を指します。通常，ラベル，変数，手続き，関数のいずれかを参照する式は再配置可能な式です。これは，これらのシンボルのアドレスはコンパイル時にはわからないからです。一方，定数だけを操作する式は絶対式です。

組み込みアセンブラでは，絶対値に対してはどのような演算も行えますが，再配置可能な値に対する演算は定数の加算と減算に制限されます。

式の型

組み込みアセンブラのすべての式には型があります。もっと正確に言えば，アセンブラは式の型をそのメモリ位置のサイズとみなすため，この型はサイズです。たとえば，Integer 変数は 4 バイトを占有するため，型は 4 です。組み込みアセンブラは可能ならば必ず型をチェックします。このため次の命令では，

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  ⋮
asm
  MOV    AL,QuitFlag
  MOV    BX,OutBufPtr
end;
```

組み込みアセンブラは，QuitFlag のサイズが 1（1 バイト）で OutBufPtr のサイズが 2（1 ワード）であることをチェックします。たとえば次の命令はエラーになります。

```
MOV    DL,OutBufPtr
```

問題なのは DL が 1 バイトサイズのレジスタで OutBufPtr が 1 ワードサイズである点です。メモリ参照の型を変更したいときは型キャストが使えます。たとえば，次に示すメモリ参照はすべて OutBufPtr 変数の最初の（最下位）バイトを参照します。

```
MOV    DL,BYTE PTR OutBufPtr
MOV    DL,Byte(OutBufPtr)
MOV    DL,OutBufPtr.Byte
```

ここでの MOV 命令は，すべて OutBufPtr 変数の最初（最下位）のバイトを参照します。

メモリ参照は型なしの（関連付けられた型がない）場合があります。たとえば，次のように [ ] で囲んだ即値（Buffer）はその例です。

```
procedure Example(var Buffer);
asm
  MOV  AL, [Buffer]
  MOV  CX, [Buffer]
  MOV  EDX, [Buffer]
```

式 [Buffer] には型がないので，組み込みアセンブラは上の両方の命令を受け付けます。この場合は，「Buffer によって示される位置の内容」を意味し，型は最初のオペランド（AL は BYTE，CX は WORD，EDX は DWORD）から判断できます。ほかのオペランドから型を判断できない場合，次の例に示すような明示的な型キャストが必要です。

```
INC     BYTE PTR [ECX]
IMUL    WORD PTR [EDX]
```

現在宣言されている Object Pascal 型以外に，組み込みアセンブラには次の定義済みの型シンボルもあります。

表 13.5　定義済みの型シンボル

シンボル	型
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

式演算子

組み込みアセンブラには，さまざまな演算子が用意されています。組み込みアセンブラの演算子の優先順位は Object Pascal とは異なります。たとえば，**asm** 文では AND は加算演算子や減算演算子より優先順位が低くなります。次の表に組み込みアセンブラの式演算子を優先順位の高い順に示します。

表 13.6　組み込みアセンブラの式演算子の優先順位

演算子	説明	優先順位
&		最も高い
()，[]，.，HIGH，LOW		
+，-	単項プラスおよび単項マイナス	
:		
OFFSET，TYPE，PTR，*，/，MOD，SHL，SHR，+，-	二項プラスおよび二項マイナス	
NOT，AND，OR，XOR		最も低い

次の表に組み込みアセンブラの式演算子とその説明を示します。

表 13.7　組み込みアセンブラの式演算子

演算子	説明
&	演算子オーバーライド。アンド記号の直後の識別子は，組み込みアセンブラの予約語と同じスペリングであっても，ユーザー定義シンボルとして扱われる
(...)	部分式。カッコで囲まれた式は 1 つの式要素として扱う前に完全に評価される。部分式の前にもう 1 つ別の式を置くことができる。この場合，結果は 2 つの式の値の合計になり，型は最初の式と同じになる
[...]	メモリ参照。大カッコ [ ] で囲まれた式は 1 つの式要素として扱う前に完全に評価される。メモリ参照式の前にもう 1 つ別の式を置くことができる。この場合，結果は 2 つの式の値の合計になり，型は最初の式と同じになる。結果は常にメモリ参照になる
.	構造体メンバー選択子。結果は，ピリオドの前の式と後ろの式の合計であり，型はピリオドの後ろの式と同じになる。ピリオドの前の式で示されるスコープに属すシンボルに対し，ピリオドの後ろの式でアクセスできる

表 13.7　組み込みアセンブラの式演算子（つづき）

演算子	説明
HIGH	HIGH 演算子に続くワードサイズの式から上位 8 ビットを返す。式は絶対即値でなければならない
LOW	LOW 演算子に続くワードサイズの式から下位 8 ビットを返す。式は絶対即値でなければならない
+	単項プラス。単項プラス演算子に続く式を変更なしにそのまま返す。式は絶対即値でなければならない
-	単項マイナス。単項マイナス演算子に続く式の負の値を返す。式は絶対即値でなければならない
+	加算。式は即値またはメモリ参照を指定できる。一方の式だけを再配置可能な値にできる。一方の式が再配置可能な値の場合，結果は再配置可能な値になる。どちらかの式がメモリ参照の場合は結果はメモリ参照になる
-	減算。最初の式はどのクラスでもかまわないが，2 番目の式は絶対即値でなければならない。結果は最初の式と同じクラスになる
:	セグメントオーバーライド。この演算子はコロンの前のセグメントレジスタ名（CS，DS，SS，ES のいずれか）で指定されたセグメントにコロンの後の式が属するよう指定する。結果は，コロンの後の式の値によるメモリ参照になる。セグメントオーバーライドを命令オペランドで使う場合，命令の先頭には該当するセグメントオーバーライドプレフィクス命令が付く。これにより指定したセグメントが確実に選択される
OFFSET	OFFSET 演算子に続く式のオフセット部（ダブルワード）を返す。結果は即値になる
TYPE	TYPE 演算子に続く式の型（バイト数で示したサイズ）を返す。即値の型は 0 になる
PTR	型キャスト演算子。結果は，2 番目の式の値と最初の式の型を持つメモリ参照になる
*****	乗算。どちらの式も絶対即値でなければならない。結果は絶対即値になる
/	整数除算。どちらの式も絶対即値でなければならない。結果は絶対即値になる
MOD	整数除算後の剰余。どちらの式も絶対即値でなければならない。結果は絶対即値になる
SHL	論理左シフト。どちらの式も絶対即値でなければならない。結果は絶対即値になる
SHR	論理右シフト。どちらの式も絶対即値でなければならない。結果は絶対即値になる
NOT	ビット否定。式は絶対即値でなければならない。結果は絶対即値になる
AND	ビット **AND**。どちらの式も絶対即値でなければならない。結果は絶対即値になる
OR	ビット **OR**。どちらの式も絶対即値でなければならない。結果は絶対即値になる
XOR	ビット排他的 **OR**。どちらの式も絶対即値でなければならない。結果は絶対即値になる

アセンブラで記述した手続きと関数

インラインアセンブラコードでは，**begin...end** 文を使用しなくても完全な手続きと関数を記述できます。次に例を示します。

```
function LongMul(X, Y: Integer): Longint;
asm
  MOV     EAX,X
  IMUL    Y
end;
```

このようなルーチンに対して，以下のような最適化が行われます。

- 値パラメータをローカル変数にコピーするためのコードは生成されません。これは，すべての文字列型の値パラメータと，文字列型以外の，サイズが 1，2，または 4 バイトでない値パラメータ

で有効です。ルーチン内部では，これらのパラメータは変数パラメータのように扱わなければなりません。

- 文字列，バリアント，インターフェース参照のいずれも返さない関数の場合，関数の Result 変数は割り当てられません。したがって，@Result シンボルへの参照はエラーになります。文字列，バリアント，およびインターフェースに対しては，呼び出し側は常に @Result ポインタを割り当てます。
- コンパイラは，ネストしたルーチン，ローカルパラメータを持つルーチン，およびスタック上にパラメータを置くルーチンに対してのみ，スタックフレームを生成します。
- ルーチンに対して自動生成されるエントリコードと終了コードは次のようになります。

```
PUSH   EBP            ; Locals <> 0 または Params <> 0 ならば生成
MOV    EBP,ESP        ; Locals <> 0 または Params <> 0 ならば生成
SUB    ESP, Locals    ; Locals <> 0 ならば生成
 ⋮
MOV    ESP, EBP       ; Locals <> 0 ならば生成
POP    EBP            ; Locals <> 0 または Params <> 0 ならば生成
RET    Params         ; 常に生成
```

Locals にバリアント，長い文字列，またはインターフェースが含まれている場合，それらはゼロに初期化されますが，終了処理は行われません。

- Locals はローカル変数のサイズを示し，Params はパラメータのサイズを示します。Locals と Params がともに 0 の場合，エントリコードは生成されず，終了コードは RET 命令だけになります。

アセンブラで記述された関数では，結果は以下のような形で返されます。

- 順序値は，AL（8 ビット値），AX（16 ビット値），または EAX（32 ビット値）に返されます。
- 実数値は，コプロセッサのレジスタスタックの ST(0) に返されます。Currency 値は 10000 分の 1 になります。
- 長い文字列を含むポインタは，EAX に返されます。
- 短い文字列とバリアントは，@Result が指す一時的な場所に返されます。

付録 A

Object Pascal の文法

```
ゴール -> ( プログラム | パッケージ | ライブラリ | ユニット )
プログラム -> [PROGRAM 識別子 ['(' 識別子リスト ')'] ';']
            プログラムブロック '.'
ユニット -> UNIT 識別子 [ 移植性指令 ] ';'
            インターフェース部
            実現部
            初期化部 '.'
パッケージ -> PACKAGE 識別子 ';'
              [requires 節 ]
              [contains 節 ]
              END '.'
ライブラリ -> LIBRARY 識別子 ';'
              プログラムブロック '.'
プログラムブロック -> [uses 節 ]
                      ブロック
移植性指令 -> platform
           -> deprecated
           -> library
uses 節 -> USES 識別子リスト ';'
インターフェース部 -> INTERFACE
                      [uses 節 ]
                      [ インターフェース宣言 ]...
インターフェース宣言 -> 定数宣言部
                     -> 型宣言部
                     -> 変数宣言部
                     -> エクスポートヘッダー
エクスポートヘッダー -> 手続きヘッダー ';' [ 指令 ]
                     -> 関数ヘッダー ';' [ 指令 ]
実現部 -> IMPLEMENTATION
          [uses 節 ]
          [ 宣言部 ]
          [Exports 文 ]...
ブロック -> [ 宣言部 ]
            [Exports 文 ]...
             複合文
            [Exports 文 ]...
```

```
Exports 文 -> EXPORTS Exports 項目 [, Exports 項目 ]...
Exports 項目 -> 識別子 [NAME | INDEX "'" 定数式 "'"]
               [NAME | INDEX "'" 定数式 "'"]
宣言部 -> ラベル宣言部
       -> 定数宣言部
       -> 型宣言部
       -> 変数宣言部
       -> 手続き宣言部
ラベル宣言部 -> LABEL ラベル識別子
定数宣言部 -> CONST ( 定数宣言 ';')...
定数宣言 -> 識別子 '=' 定数式 [ 移植性指令 ]
         -> 識別子 ':' 型識別子 '=' 型付き定数 [ 移植性指令 ]
型宣言部 -> TYPE ( 型宣言 ';')...
型宣言 -> 識別子 [TYPE] '=' 型 [ 移植性指令 ]
       -> 識別子 [TYPE] '=' 限定型 [ 移植性指令 ]
型付き定数  -> ( 定数式 | 配列定数 | レコード定数 )
配列定数 -> '(' 型付き定数 ','... ')'
レコード定数 -> '(' レコードフィールド定数 ';'... ')'
レコードフィールド定数 -> 識別子 ':' 型付き定数
型 -> 型識別子
   -> 単純型
   -> 構造化型
   -> ポインタ型
   -> 文字列型
   -> 手続き型
   -> バリアント型
   -> クラス参照型
限定型 -> オブジェクト型
       -> クラス型
       -> インターフェース型
クラス参照型  -> CLASS OF 型識別子
単純型 -> ( 順序型 | 実数型 )
実数型 -> REAL48
       -> REAL
       -> SINGLE
       -> DOUBLE
       -> EXTENDED
       -> CURRENCY
       -> COMP
順序型 -> ( 部分範囲型 | 列挙型 | 順序型識別子 )
順序型識別子 -> SHORTINT
             -> SMALLINT
             -> INTEGER
             -> BYTE
             -> LONGINT
             -> INT64
             -> WORD
             -> BOOLEAN
             -> CHAR
             -> WIDECHAR
             -> LONGWORD
             -> PCHAR
```

```
バリアント型 -> VARIANT
            -> OLEVARIANT
部分範囲型 -> 定数式 '..' 定数式
列挙型 -> '(' 列挙型要素 ','... ')'
列挙型要素 -> 識別子 [ '=' 定数式 ]
文字列型 -> STRING
         -> ANSISTRING
         -> WIDESTRING
         -> STRING '[' 定数式 ']'
構造化型 -> [PACKED] （配列型 [PACKED] ｜ 集合型 ｜ ファイル型 ｜ レコード型 [PACKED])
配列型 -> ARRAY ['[' 順序型 ','...']'] OF 型 ［移植性指令］
レコード型 -> RECORD ［フィールドリスト］ END ［移植性指令］
フィールドリスト ->  フィールド宣言 ';'...［レコード可変部］ [';']
フィールド宣言 -> 識別子リスト ':' 型 ［移植性指令］
レコード可変部 -> CASE ［識別子 ':'] 型識別子 OF 可変部 ';'...
可変部 -> 定数式 ','...':' '(' ［フィールドリスト］ ')'
集合型 -> SET OF 順序型 ［移植性指令］
ファイル型 -> FILE OF 型識別子 ［移植性指令］
ポインタ型 -> '^' 型識別子 ［移植性指令］
手続き型 -> （手続きヘッダー ｜ 関数ヘッダー） [OF OBJECT]
変数宣言部 -> VAR （変数宣言 ';')...
変数宣言
    (Windows の場合）
     -> 識別子リスト ':' 型 [(ABSOLUTE （識別子 ｜ 定数式）) ｜ '=' 定数式）［移植性指令］
    (Linux の場合）
     -> 識別子リスト ':' 型 [ABSOLUTE （識別子） ｜ '=' 定数式 ［移植性指令］
式 -> 単純式 ［関係演算子 単純式］...
単純式 -> ['+' ｜ '-'] 項 ［加減演算子 項］...
項 -> 要素 ［乗除演算子 要素］...
要素 -> 指定子 ['(' 式リスト ')']
     -> '@' 指定子
     -> 数
     -> 文字列
     -> NIL
     -> '(' 式 ')'
     -> NOT 要素
     -> 集合構成子
     -> 型識別子 '(' 式 ')'
関係演算子 -> '>'
           -> '<'
           -> '<='
           -> '>='
           -> '<>'
           -> IN
           -> IS
           -> AS
加減演算子 -> '+'
           -> '-'
           -> OR
           -> XOR
```

```
乗除演算子 -> '*'
          -> '/'
          -> DIV
          -> MOD
          -> AND
          -> SHL
          -> SHR
指定子 -> 限定識別子 ['.' 識別子 | '[' 式リスト ']' | '^']...
集合構成子 -> '[' [ 集合要素 ','...] ']'
集合要素 -> 式 ['..' 式 ]
式リスト -> 式 ','...
文 -> [ ラベル識別子 ':'] [ 単純文 | 構造化文 ]
文リスト -> 文 ';'...
単純文 -> 指定子 ['(' [ 式リスト ] ')']
       -> 指定子 ':=' 式
       -> INHERITED
       -> GOTO ラベル識別子
構造化文 -> 複合文
         -> 条件文
         -> ループ文
         -> with 文
         -> Try..Except 文
         -> Try..Finally 文
         -> Raise 文
         -> アセンブラ文
複合文 -> BEGIN 文リスト END
条件文 -> if 文
       -> case 文
if 文 -> IF 式 THEN 文 [ELSE 文 ]
case 文 -> CASE 式 OF ケースセレクタ ';'...[ELSE 文リスト ] [';'] END
ケースセレクタ -> ケースラベル ','...':' 文
ケースラベル -> 定数式 ['..' 定数式 ]
ループ文 -> repeat 文
         -> while 文
         -> for 文
repeat 文 -> REPEAT 文 UNTIL 式
while 文 -> WHILE 式 DO 文
for 文 -> FOR 限定識別子 ':=' 式 (TO | DOWNTO) 式 DO 文
with 文 -> WITH 識別子リスト DO 文
Try..Except 文 -> TRY
                   文 ...
                 EXCEPT
                   例外ブロック
                 END
例外ブロック -> [ON [ 識別子 ':'] 型識別子 DO 文 ]...
                  [ELSE 文 ...]
Try..Finally 文 -> TRY
                    文
                  FINALLY
                    文
                  END
```

```
Raise 文 -> RAISE [ オブジェクト ] [AT アドレス ]
アセンブラ文 -> ASM
             -> < アセンブリ言語 >
             -> END
手続き宣言部 -> 手続き宣言
             -> 関数宣言
手続き宣言 -> 手続きヘッダー ';' [ 指令 ] [ 移植性指令 ]
              ブロック ';'
関数宣言 -> 関数ヘッダー ';' [ 指令 ] [ 移植性指令 ]
            ブロック ';'
関数ヘッダー -> FUNCTION 識別子 [ 仮パラメータリスト ] ':' ( 単純型 | STRING)
手続きヘッダー -> PROCEDURE 識別子 [ 仮パラメータリスト ]
仮パラメータリスト -> '(' 仮パラメータ ';'... ')'
仮パラメータ -> [VAR | CONST | OUT] パラメータ
パラメータ -> 識別子リスト  [':' ([ARRAY OF] 単純型 | STRING | FILE)]
           -> 識別子 ':' 単純型 '=' 定数式
指令 -> CDECL
     -> REGISTER
     -> DYNAMIC
     -> VIRTUAL
     -> EXPORT
     -> EXTERNAL
     -> NEAR
     -> FAR
     -> FORWARD
     -> MESSAGE 定数式
     -> OVERRIDE
     -> OVERLOAD
     -> PASCAL
     -> REINTRODUCE
     -> SAFECALL
     -> STDCALL
     -> VARARGS
     -> LOCAL
     -> ABSTRACT
オブジェクト型 -> OBJECT [ オブジェクト継承 ] [ オブジェクトフィールドリスト ] [ メソッドリスト ] END
オブジェクト継承 -> '(' 限定識別子 ')'
メソッドリスト -> ( メソッドヘッダー [';' VIRTUAL]) ';'...
メソッドヘッダー -> 手続きヘッダー
                 -> 関数ヘッダー
                 -> コンストラクタヘッダー
                 -> デストラクタヘッダー
コンストラクタヘッダー -> CONSTRUCTOR 識別子 [ 仮パラメータリスト ]
デストラクタヘッダー -> DESTRUCTOR 識別子 [ 仮パラメータリスト ]
オブジェクトフィールドリスト -> ( 識別子リスト ':' 型 ) ';'...
初期化部 -> INITIALIZATION 文リスト [FINALIZATION 文リスト ] END
         -> BEGIN 文リスト END
         -> END
クラス型 -> CLASS [ クラス継承 ]
            [ クラス可視性 ]
            [ クラスフィールドリスト ]
```

［クラスメソッドリスト］
［クラスプロパティリスト］
END

クラス継承 -> '(' 識別子リスト ')'

クラス可視性 -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]

クラスフィールドリスト -> （クラス可視性 オブジェクトフィールドリスト） ';'...

クラスメソッドリスト -> （クラス可視性 メソッドリスト） ';'...

クラスプロパティリスト -> （クラス可視性 プロパティリスト ';')...

プロパティリスト -> PROPERTY 識別子 ［プロパティインターフェース］［プロパティ指定子］
［移植性指令］

プロパティインターフェース -> ［プロパティパラメータリスト］ ':' 識別子

プロパティパラメータリストt -> '[' （識別子リスト ':' 型識別子） ';'... ']'

プロパティ指定子 -> [INDEX 定数式］
[READ 識別子］
[WRITE 識別子］
[STORED （識別子 | 定数）]
[（DEFAULT 定数式） | NODEFAULT]
[IMPLEMENTS 型識別子］

インターフェース型 -> INTERFACE ［インターフェース継承］
［クラスメソッドリスト］
［クラスプロパティリスト］
...
END

インターフェース継承 -> '(' 識別子リスト ')'

requires 節 -> REQUIRES 識別子リスト ... ';'

contains 節 -> CONTAINS 識別子リスト ... ';'

識別子リスト -> 識別子 ','...

限定識別子 -> ［ユニット識別子 '.'］識別子

型識別子 -> ［ユニット識別子 '.'］ <型識別子>

識別子 -> <識別子>

定数式 -> <定数式>

ユニット識別子 -> <ユニット識別子>

ラベル識別子 -> <ラベル識別子>

数 -> <数>

文字列 -> <文字列>

索　引

記号

- 4-4, 4-6, 4-9, 4-10
" 13-9
4-4
$ 4-4, 4-5
(*, *) 4-5
(,) 4-2, 4-12, 4-14, 5-6, 5-41, 6-2, 6-3, 6-11, 7-2, 10-1
* 4-2, 4-6, 4-10
+ 4-4, 4-6, 4-9, 4-10
, 3-6, 4-23, 5-6, 5-21, 5-22, 6-11, 7-17, 9-5, 9-9, 10-7, 13-2
. 3-2, 4-2, 4-13, 5-26, 9-9, 10-5
/ 4-2, 4-6
// 4-5
: 4-2, 4-17, 4-23, 5-21, 5-22, 5-36, 5-41, 6-3, 6-11, 7-17, 7-19, 7-29, 13-2
:= 4-17, 4-25
　名前付きパラメータ 10-12
; 3-2, 3-6, 4-16, 4-17, 4-19, 4-24, 5-21, 5-22, 5-36, 6-2, 6-3, 6-11, 7-2, 7-7, 9-5, 9-9, 10-1, 10-4, 10-11
　else の前 4-22
= 4-2, 4-9, 4-10, 4-11, 4-17, 5-35, 5-37, 5-39, 5-41, 6-11, 6-17, 10-5
< 4-2, 4-9, 4-11
<= 4-9, 4-10, 4-11
<> 4-9, 4-10, 4-11
> 4-2, 4-9, 4-11
>= 4-9, 4-10, 4-11
@ 4-6, 4-12, 5-25, 5-29, 5-42, 7-18
@@ 5-29
@Result 13-10, 13-16
[,] 4-13, 4-14, 5-11, 5-17, 5-30, 6-15, 6-19, 7-17, 7-19, 7-20, 10-1, 10-11
^ 4-6, 4-9, 5-19, 5-26
　バリアント 5-32
　ポインタの概要 5-25
_ 4-2
{, } 4-5, 10-1, 10-11

数字

16 進数値 4-4
16 ビットアプリケーション（下位互換性）6-5

A

$A 指令 11-8
absolute（指令）5-37
Add メソッド（TCollection）7-9
Addr 関数 5-26
_AddRef メソッド 10-2, 10-5, 10-9
AllocMemCount 変数 11-2
AllocMemSize 変数 11-2
and 4-7, 4-8
ANSI 文字 5-4, 5-12, 5-13
AnsiChar 型 5-4, 5-11, 5-13, 5-27, 11-3
AnsiString 型 5-10, 5-12, 5-13, 5-15, 5-27
　→「長い文字列」も参照
　バリアント配列と～ 5-33
　メモリ管理 11-6
Append 手続き 8-2, 8-4, 8-5, 8-6
Application 変数 2-6, 3-3
as 4-11, 7-25, 10-10
ASCII 4-1, 4-4, 5-12
asm 文 13-1, 13-15
assembler（指令）6-6, 13-1
assembly language
　built-in assembler 13-1～??
Assert 手続き 7-27
Assign 手続き
　カスタム～ 8-4
Assigned 関数 5-30, 10-9
AssignFile 手続き 8-2, 8-4, 8-6
at（予約語）7-28
automated クラスメンバー 7-4, 7-6

B

$B 指令 4-8
begin（予約語）3-3, 4-19, 6-2, 6-3
BlockRead 手続き 8-4
BlockWrite 手続き 8-4
Boolean 型 5-5, 11-3
BORLANDMM.DLL 9-8
.bpl ファイル 9-8, 9-11
Break 手続き 4-24
　try...finally ブロック内の～ 7-32
　例外ハンドラ 7-30
BSTR 型（COM）5-12
built-in assembler 13-1～??
Byte 型 5-4, 11-3
　アセンブラ 13-14
ByteArray 型 5-27
ByteBool 型 5-5, 11-3

C

C++ 10-1, 11-11
Cardinal 型 5-3
case 文 4-23
case（予約語）4-23, 5-22
-cc コンパイラスイッチ 8-3
cdecl（呼び出し規約）6-4, 12-2

Self 12-3
コンストラクタとデストラクタ 12-4
Char 型 5-5, 5-13, 5-27, 11-3
Chr 関数 5-5
Classes ユニット 7-9, 7-24
ClassParent メソッド 7-25
ClassType メソッド 7-25
Close 関数 8-4, 8-6
CloseFile 関数 8-5
CloseFile 手続き 8-6
CLX 1-1
CmdLine 変数 9-7
COM 10-4
→「オートメーション」も参照
OleVariant 5-33
safecall 6-5
インターフェース 10-3
エラー処理 6-5
バリアントと～ 5-32, 11-11
ComObj ユニット 7-6, 10-12
Comp 型 5-9, 5-10, 11-5
const（予約語）5-39, 5-41, 6-11, 6-13, 6-16, 12-1
contains 節 9-9, 9-10
Continue 手続き 4-24
try...finally ブロック内の～ 7-32
例外ハンドラ 7-30
Copy 関数 5-19
CORBA
インターフェース 10-3
CPU→「レジスタ」を参照
Create メソッド 7-13
Currency 型 5-9, 5-10, 5-27, 11-5

D

.dcp ファイル 9-11
.dcu ファイル 2-3, 3-7, 9-10, 9-11
Dec 手続き 5-3, 5-4
default 指定子 7-6, 7-17, 7-21
default（指令）7-20, 10-11
DefaultHandler メソッド 7-16, 7-17
DefWindowProc 関数 7-17
Delphi 1-1
バリアントと～ 11-12
$DENYPACKAGEUNIT 指令 9-11
$DESIGNONLY 指令 9-12
.desk ファイル 2-3
Destroy メソッド 7-14, 7-15, 7-30
.dfm ファイル 2-2, 2-7
Dispatch メソッド 7-17
dispid（指令）10-2, 10-11
dispinterface 10-11
dispinterface（予約語）10-2
Dispose 手続き 5-19, 5-38, 7-4, 9-8, 11-1, 11-2
div 4-6
dlclose 9-2
DLL 9-1～9-8
グローバル変数 9-6
静的ロード 9-2
～でのルーチンの呼び出し 6-7, 9-1
動的にロードする 9-2
～内の動的配列 9-8
～内の動的変数 9-8
～内の長い文字列 9-8
～の作成 9-3
変数 9-1
マルチスレッド 9-7
例外 9-7
.DLL ファイル 9-1
DLL_PROCESS_ATTACH 9-7
DLL_PROCESS_DETACH 9-7
DLL_THREAD_ATTACH 9-7
DLL_THREAD_DETACH 9-7
DLLProc 変数 9-7
dlopen 9-2
dlsym 9-2
do（予約語）4-20, 4-25, 7-29
.dof ファイル 2-3
Double 型 5-9, 11-4
downto（予約語）4-25
.dpk ファイル 2-2, 9-9, 9-11
.dpr ファイル 2-2, 3-1, 3-6
.dpu ファイル 2-3, 3-7, 9-10, 9-11
.drc ファイル 2-3
.dsk ファイル 2-3
DWORD 型（アセンブラ）13-14

E

E（数値表記内の～）4-4
EAssertionFailed 7-27
else（予約語）4-22, 4-23, 7-29
end（予約語）3-3, 4-19, 4-23, 5-21, 5-22, 6-2, 6-3, 7-2, 7-29, 7-32, 9-9, 10-1, 10-11, 13-1
Eof 関数 8-5
Eoln 関数 8-5
ErrorAddr 変数 12-5
EStackOverflow 例外 11-2
EVariantError 例外 5-32
except（予約語）7-29
ExceptAddr 関数 7-32
Exception クラス 7-27, 7-32
ExceptionInformation 変数 9-7
ExceptObject 関数 7-32
Exit 手続き 6-1
try...finally ブロック内の～ 7-32
例外ハンドラ 7-30
ExitCode 変数 9-6, 12-5

ExitProc 変数 9-6, 12-4, 12-5
export（指令）6-5
exports 節 4-27, 9-5
　オーバーロードルーチン 9-5
Extended 型 4-7, 5-9, 5-27, 11-5
external（指令）6-6, 9-2

F

False 5-5, 11-3
far（指令）6-5
file（予約語）5-24
FilePos 関数 8-2
FileSize 関数 8-2
Finalize 手続き 5-19
finally（予約語）7-32
Flush 関数 8-4, 8-6
for 文 4-19, 4-24, 4-25
forward 宣言
　インターフェース 10-4
　オーバーロードと〜 6-9
　クラス 7-7
　デフォルトパラメータ 6-18
　ルーチン 3-4, 6-5
Free メソッド 7-15
FreeLibrary 関数 9-2
FreeMem 手続き 5-38, 9-8, 11-1, 11-2

G

-$G- コンパイラスイッチ 9-12
$G 指令 9-11
GetHeapStatus 関数 11-2
GetMem 手続き 5-26, 5-38, 9-8, 11-1, 11-2
GetMemoryManager 手続き 11-2
GetProcAddress 関数 9-2
GlobalAlloc 11-1
goto 文 4-18
GUID 10-1, 10-3, 10-10

H

$H 指令 5-10, 6-15
Halt 手続き 12-4, 12-5
Hello world! 2-3
HelpContext プロパティ 7-32
High 関数 5-3, 5-4, 5-12, 5-18, 5-19, 6-16
HInstance 変数 9-7

I

$I 指令 8-3
IDE 1-1
　→「Delphi」も参照
　プロジェクトファイル 3-1
　プロジェクトマネージャ 2-1
IDispatch 10-9, 10-10, 10-11
　デュアルインターフェース 10-13
if...then 文 4-21
　ネストした〜 4-22
IInterface 10-2, 10-5
implements（指令）7-22, 10-6
$IMPLICITBUILD 指令 9-11
$IMPORTEDDATA 指令 9-11
in（予約語）3-6, 4-10, 5-17, 5-32, 9-9
Inc 手続き 5-3, 5-4
index 指定子（Windows のみ）9-5
index（指令）6-7
inherited（予約語）7-9, 7-14
　メッセージハンドラ 7-16
　呼び出し規約 12-4
InheritsFrom メソッド 7-25
Initialize 手続き 5-38
inline assembler code 13-1〜??
inline（予約語）13-1
InOut 関数 8-4, 8-5
input（プログラムパラメータ）3-2
Input 変数 8-3
Int64 型 4-7, 5-3, 5-4, 5-10, 11-3
　バリアントと〜 5-30
　標準関数と標準手続き 5-4
Integer 型 4-7, 5-3, 5-4
IntToHex 関数 5-4
IntToStr 関数 5-4
Invoke メソッド 10-11
IOResult 関数 8-3, 8-4
is 4-11, 5-32, 7-25
IsLibrary 変数 9-7
IUnknown 10-2, 10-9, 10-13

J

$J 指令 5-41
Java 10-1

K

Kylix 1-1

L

-$LE- コンパイラスイッチ 9-12
Length 関数 5-10, 5-18, 5-19
library（予約語）9-3
-$LN- コンパイラスイッチ 9-12
LoadLibrary 関数 9-2
local（指令）9-4
　Linux 9-4
LocalAlloc 11-1
LongBool 型 5-5, 11-3
Longint 型 5-4, 11-3
Longword 型 5-4, 11-3

Low 関数 5-3, 5-4, 5-12, 5-18, 5-19, 6-16
-$LU- コンパイラスイッチ 9-12

M

$M 指令 7-4, 7-6
$MAXSTACKSIZE 指令 11-2
message（指令）7-16
  インターフェース 10-7
Message プロパティ 7-32
Messages ユニット 7-16
$MINSTACKSIZE 指令 11-2
mod 4-6

N

name（指令）6-7, 9-5
near（指令）6-5
New 手続き 5-19, 5-26, 5-38, 7-4, 9-8, 11-1, 11-2
nil 5-26, 5-30, 11-5
nodefault 指定子 7-6, 7-17, 7-21
not 4-6, 4-7
Null（バリアント）5-30, 5-32

O

.OBJ ファイル
  ～でのルーチンの呼び出し 6-6
of object（メソッドポインタ）5-28
of（予約語）4-23, 5-16, 5-18, 5-24, 5-28, 6-15, 6-16, 7-23
OLE オートメーション 5-33
OleAuto ユニット 7-6
OleVariant 5-33
OleVariant 型 5-27, 5-33
on（予約語）7-29
Open 関数 8-4, 8-5
OpenString 6-15
or 4-7, 4-8
Ord 関数 5-3, 5-4
out（予約語）6-11, 6-13
OutlineError 7-32
output（プログラムパラメータ）3-2
Output 変数 8-3

P

$P 指令 6-15
packed 配列 4-5, 4-9, 5-18
packed 文字列 5-18
  比較 4-11
packed（予約語）5-16, 11-8
PAnsiChar 型 5-13, 5-27
PAnsiString 型 5-27
.PAS ファイル 3-3
.pas ファイル 2-1, 3-1, 3-7
pascal（呼び出し規約）6-4, 12-2
  Self 12-3
  コンストラクタとデストラクタ 12-4
PByteArray 型 5-27
PChar 型 4-5, 4-9, 5-13, 5-14, 5-15, 5-27, 5-43
  比較 4-11
PCurrency 型 5-27
PExtended 型 5-27
PGUID 10-3
Pointer 型 5-25, 5-26, 11-5
POleVariant 型 5-27
Pred 関数 5-3, 5-4
private クラスメンバー 7-4, 7-5
program（予約語）3-2
protected クラスメンバー 7-4, 7-5
PShortString 型 5-27
PString 型 5-27
PTextBuf 型 5-27
Ptr 関数 5-26
public クラスメンバー 7-4, 7-5
published クラスメンバー 7-4, 7-5
  $M 指令 7-6
  制限 7-5
PVariant 型 5-27
PVarRec 型 5-27
PWideChar 型 5-13, 5-14, 5-27
PWideString 型 5-27
PWordArray 型 5-27

Q

QueryInterface メソッド 10-2, 10-5, 10-10
QWORD 型（アセンブラ）13-14

R

raise（予約語）4-19, 7-28, 7-29, 7-31
read 指定子 7-6, 7-17, 7-18
  インデックス指定子と～ 7-21
  オブジェクトインターフェース 10-2, 10-4, 10-7
  ～のオーバーロード 7-13, 7-18
  配列プロパティ 7-20
Read 手続き 8-2, 8-3, 8-4, 8-5, 8-6
Readln 手続き 8-5, 8-6
readonly（指令）10-2, 10-11
Real 型 5-9
Real48 型 5-9, 7-5, 11-4
$REALCOMPATIBILITY 指令 5-9
ReallocMem 手続き 5-38, 11-1
register（呼び出し規約）6-4, 7-6, 7-13, 7-15, 12-2
  DLL 9-4
  Self 12-3
  インターフェース 10-3, 10-7
  コンストラクタとデストラクタ 12-4
reintroduce（指令）7-12
_Release メソッド 10-2, 10-5, 10-9

Rename 手続き 8-6
repeat 文 4-19, 4-24
requires 節 9-8, 9-9, 9-10
.RES ファイル 2-2, 3-2
Reset 手続き 8-2, 8-4, 8-5, 8-6
resident（指令）9-5
resourcestring（予約語）5-40
Result 変数 6-3, 6-4
RET 命令 13-3
Rewrite 手続き 8-2, 8-4, 8-5, 8-6
Round 関数 5-4
RTTI 7-5, 7-13, 7-21
$RUNONLY 指令 9-12

S

$S 指令 11-2
safecall（呼び出し規約）6-4, 12-2
  Self 12-3
  コンストラクタとデストラクタ 12-4
  デュアルインターフェース 10-13
Seek 手続き 8-2
SeekEof 関数 8-5
SeekEoln 関数 8-5
Self 7-9
  クラスメソッド 7-26
  呼び出し規約 12-3
SetLength 手続き 5-10, 5-16, 5-18, 5-20, 6-16
SetMemoryManager 手続き 11-2
SetString 手続き 5-16
ShareMem ユニット 9-8
shl 4-8
Shortint 型 5-4, 11-3
ShortString 型 5-10, 5-11, 5-27, 11-5
  パラメータ 6-15
  バリアント配列と～ 5-33
ShowException 手続き 7-32
shr 4-8
Single 型 5-9, 11-4
SizeOf 関数 5-2, 5-5, 6-16
Smallint 型 5-4, 11-3
.so ファイル 9-1
stdcall（呼び出し規約）6-4, 12-2
  DLL 9-4
  Self 12-3
  インターフェース 10-3
  コンストラクタとデストラクタ 12-4
stored 指定子 7-6, 7-17, 7-21
Str 手続き 8-6
StrAlloc 関数 5-38
StrDispose 手続き 5-38
string（予約語）5-10
StringToWideChar 関数 8-7
StrToInt64 関数 5-4
StrToInt64Def 関数 5-4
StrUpper 関数 5-14
Succ 関数 5-3, 5-4
System ユニット 3-1, 3-5, 5-27, 5-31, 5-32, 6-17, 7-3, 7-28, 8-1, 8-7, 10-2, 10-3, 10-5, 10-10, 11-11
  DLL 9-6, 9-7
  uses 節と～ 8-1
  スコープ 4-28
  ～の変更 8-1
  メモリ管理 11-2
SysUtils ユニット 3-5, 5-27, 6-11, 6-17, 7-26, 7-27, 7-28, 7-32
  DLL 9-8
  uses 節と～ 8-1

T

$T 指令 4-12
TAggregatedObject 10-7
TAutoObject 7-6
TBYTE 型（アセンブラ）13-14
TClass 7-3, 7-24, 7-25
TCollection 7-24
  Add メソッド 7-9
TCollectionItem 7-24
TDateTime 5-32
Text 型 5-24, 8-3
TextBuf 型 5-27
TextFile 型 5-24
TGUID 10-3
then（予約語）4-21
threadvar 5-38
TInterfacedObject 10-2, 10-5
to（予約語）4-25
TObject 7-3, 7-16, 7-17, 7-25
TPersistent 7-6
True 5-5, 11-3
Trunc 関数 5-4
try...except 文 4-19, 7-28, 7-29
try...finally 文 4-19, 7-32
TTextRec 型 5-27
Turbo Assembler 13-4
TVarData 5-31, 11-11
TVarRec 型 5-27, 6-17
TWordArray 5-27

U

Unassigned（バリアント）5-30, 5-32
Unicode 5-4, 5-12, 5-13
UniqueString 手続き 5-16
until（予約語）4-24
UpCase 関数 5-11
uses 節 2-1, 3-1, 3-2, 3-4, 3-5～3-8
  ShareMem 9-8
  System ユニットと～ 8-1

SysUtils ユニットと～ 8-1
インターフェース部 3-7
構文 3-6

V

Val 手続き 8-6
var（予約語）5-36, 6-11, 6-12, 12-1
VarArrayCreate 関数 5-32
VarArrayDimCount 関数 5-33
VarArrayHighBound 関数 5-33
VarArrayLock 関数 5-33, 10-12
VarArrayLowBound 関数 5-33
VarArrayOf 関数 5-32
VarArrayRedim 関数 5-33
VarArrayRef 関数 5-33
VarArrayUnlock 手続き 5-33, 10-12
VarAsType 関数 5-31
VarCast 手続き 5-31
varOleString 定数 5-33
varString 定数 5-33
VarType 関数 5-30
varTypeMask 定数 5-30
VCL 1-1
VirtualAlloc 関数 11-1
VirtualFree 関数 11-1
VMT 11-10

W

$WEAKPACKAGEUNIT 指令 9-11
while 文 4-19, 4-24, 4-25
WideChar 型 4-9, 5-4, 5-11, 5-13, 5-27, 11-3
WideCharLenToString 関数 8-7
WideCharToString 関数 8-7
WideString 型 5-10, 5-12, 5-13, 5-27
メモリ管理 11-6
Windows 2-2, 5-13, 5-32, 6-5, 7-17, 8-4
DLL 9-1, 9-6
バリアントと～ 11-11
メッセージ 7-15
メモリ管理 11-1, 11-2
Windows ユニット 9-2
with 文 4-19, 4-20, 5-21
Word 型 5-4, 11-3
アセンブラ 13-14
WordBool 型 5-5, 11-3
write 指定子 7-6, 7-17, 7-18
インデックス指定子と～ 7-21
オブジェクトインターフェース 10-2, 10-4
～のオーバーロード 7-13, 7-18
配列プロパティ 7-20
Write 手続き 8-2, 8-3, 8-4, 8-5, 8-6
Writeln 手続き 2-4, 8-5, 8-6
writeonly（指令）10-2, 10-11

X

$X 指令 4-5
.xfm ファイル 2-2
xor 4-7, 4-8

Z

-$Z- コンパイラスイッチ 9-12
$Z 指令 11-3

あ

アクセス指定子 7-1, 7-17, 7-18
インデックス指定子と～ 7-21
オートメーション 7-6
～のオーバーライド 7-22
～のオーバーロード 7-13, 7-18
配列プロパティ 7-20
呼び出し規約 6-5, 7-18
アスタリスク（*）→「記号」セクション
アセンブリ言語
Object Pascal と～ 13-4, 13-7, 13-8, 13-10, 13-12, 13-14
アセンブラルーチン 13-15
外部ルーチン 6-7
組み込みアセンブラ ??～13-16
値型キャスト 4-14
値の巡回（順序型）5-4, 5-5
値パラメータ 6-11, 6-12, 6-19, 12-1
オープン配列コンストラクタ 6-20
値渡し（パラメータ）6-12, 10-12, 12-1
後の順序値 5-3
アドレス演算子 4-12, 5-25, 5-29, 5-42
プロパティと～ 7-18
アプリケーションパーティショニング 9-8
アラインメント
データの～ 5-16, 11-8
アラインメント（データの～）
→「内部データ形式」も参照
アンド記号 →「記号」セクション

い

依存
ユニット 3-7～3-8
委任（インターフェースの実装）10-6
委任されたインターフェース 10-7
イベント 2-7, 7-5
イベントハンドラ 2-7, 7-5
インクリメント
順序型変数 5-3, 5-4, 5-5
インターフェース 7-2, 10-1～10-13
GUID 10-1, 10-3, 10-10
アクセス 10-8～10-10
委任 10-6
オートメーション 10-10

〜型 10-1〜10-4
型キャスト 10-10
互換性 10-9
〜参照 10-8〜10-10
ディスパッチ〜型 10-11
デュアル〜 10-13
問い合わせ 10-10
〜の実装 10-4〜10-7
バリアントと〜 5-30
プロパティ 10-2, 10-4, 10-7
メソッド解決節 10-5, 10-6
メモリ管理 11-2
呼び出し規約 10-3
レコードと〜 5-23
インターフェース宣言 3-4
デフォルトパラメータ 6-18
インターフェース部 3-3, 3-4, 3-7
forward 宣言と〜 6-6
uses 節 3-7
スコープ 4-28
メソッド 7-9
インデックス 4-14
配列プロパティ 7-20
インデックス指定子 7-6, 7-17, 7-21
隠蔽
インターフェースの実装 10-6
クラスメンバー 7-8, 7-12, 7-23
reintroduce 7-12
隠蔽（クラスメンバーの〜）
→「オーバーロードメソッド」も参照
インポート（DLL からのルーチンの〜）9-1
インポート（ライブラリからのルーチンの〜）9-1
引用符（Õ Õ）→「記号」セクション
引用符付き文字列 4-4, 5-43
アセンブラ 13-9
インラインアセンブラコード ??〜13-16

え

英数字 4-1, 4-2
エラー処理 →「例外」
演算子 4-6〜4-13
アセンブラ 13-14
クラス 7-25
優先順位 4-12, 7-25

お

オートメーション 7-6, 10-10〜10-13
→「COM」も参照
デュアルインターフェース 10-13
バリアントと〜 11-12
メソッド呼び出し 10-12
オートメーション可能な型 7-6, 10-11
オーバーライド
インターフェースの実装 10-6
プロパティの〜 7-22
アクセス指定子と〜 7-22
隠蔽と〜 7-23
オートメーション 7-6
メソッドの〜 7-11, 10-6
隠蔽と〜 7-12
オーバーロード手続き/関数 6-6, 6-8
DLL 9-5
forward 宣言 6-9
デフォルトパラメータ 6-10, 6-18
オーバーロードメソッド 7-12
アクセス指定子 7-13, 7-18
〜のパブリッシュ 7-6
オープン配列コンストラクタ 6-17, 6-19
オープン配列パラメータ 6-15, 6-19
動的配列と〜 6-15
大文字小文字の区別 4-1, 4-2, 6-8
ユニット名とファイル 4-2
オブジェクト 4-20, 7-1
→「クラス」も参照
of object 5-28
比較 4-11
ファイルと〜 5-24
メモリ 11-10
オブジェクトインスペクタ 7-5
オブジェクトインターフェース →「インターフェース」，「COM」，「CORBA」を参照
オブジェクト型 7-4
オペコード（アセンブラ）13-2
オペランド 4-6

か

改行 4-5
下位クラス 7-3, 7-5
外部ブロック 4-28
書き込み時コピー 5-12
書き込み専用プロパティ 7-19
拡張構文 4-5
格納指定子 7-21
配列プロパティと〜 7-22
加算 4-6
ポインタ 4-9
可視性（クラスメンバー）7-4
インターフェース 10-2
下線記号 4-2
仮想メソッド 7-10, 7-11
オートメーション 7-6
コンストラクタ 7-14
〜のオーバーロード 7-12
仮想メソッドテーブル 11-10
型 5-1〜5-36
アセンブラ 13-13

インターフェース 10-1～10-4
オートメーション可能な～ 7-6, 10-11
オブジェクト～ 7-4
基本～ 5-2
組み込みの～ 5-1
クラス～ 7-1, 7-2, 7-4, 7-7, 7-8, 7-17, 11-10
クラス参照～ 7-23, 11-11
構造化～ 5-16
互換性 5-16, 5-28, 5-34, 5-35
実数～ 5-9, 11-3
集合～ 5-16, 11-7
順序～ 5-3～5-9
スコープ 5-36
整数～ 5-3, 11-3
宣言済みの～ 5-1
代入互換性 5-35
単純～ 5-2
定義済みの～ 5-1
定数 5-39
ディスパッチインターフェース 10-11
手続き～ 5-27～5-30, 11-10
内部データ形式 11-2～11-12
～の宣言 5-35
～の同一性 5-34
配列～ 5-17～5-20, 11-7
バリアント～ 5-30～5-33
汎用～ 5-2
ファイル～ 5-24, 11-9
部分範囲～ 5-8
分類 5-1
ポインタ～ 5-26～5-27
文字～ 5-4, 11-3
文字列～ 5-10～5-16, 11-5, 11-6
ユーザー定義の～ 5-1
例外 7-27
レコード～ 5-21～5-24, 11-8
列挙～ 5-6～5-8, 11-3
論理～ 5-5, 11-3
型可変オープン配列パラメータ 6-16, 6-19
型キャスト 4-14～4-16, 7-8
インターフェース 10-10
型なしパラメータ 6-14
チェック付き～ 7-25, 10-10
定数宣言での～ 5-39
バリアント 5-31
列挙型 5-7
型なしパラメータ 6-14
型識別子 5-2
型チェック（オブジェクト）7-25
型なしファイル 5-24, 8-2, 8-4
カッコ ( ) →「記号」セクション
可変部（レコード）5-22～5-24
仮パラメータ 6-19
関係演算子 4-11
関数 3-4, 6-1～6-20
アセンブラ 13-15
外部～の呼び出し 6-6
ネストした～ 5-28, 6-10
～のオーバーロード 6-6, 6-8
～の宣言 6-3, 6-5
～ポインタ 4-12, 5-28
戻り型 6-3, 6-4
戻り値 6-3, 6-4
～呼び出し 4-13, 4-18, 6-1, 6-19～6-20
レジスタ上の戻り値 12-3, 13-16
間接ユニット参照 3-6～3-7
完全評価 4-7
カンマ（,）→「記号」セクション

き

記号→「記号」セクションを参照
疑似命令（アセンブラ）13-3
基本型 5-2, 5-8, 5-16, 5-17, 5-18
逆参照演算子 4-9, 5-19
バリアントと～ 5-32
ポインタの概要 5-25
キャレット（^）→「記号」セクション
行終端文字 4-1, 8-3
共有オブジェクト 9-1
共有オブジェクトファイル 2-3, 9-1
関数のインポート 6-7
動的ロード可能な～ 9-2

く

空集合 5-17
空白 4-1
組み合わせシンボル 4-2
組み込みアセンブラ ??～13-16
組み込みの型 5-1
クライアント 3-4
クラス 7-1～7-32
～演算子 4-11, 7-25
～型 7-1, 7-2
～型の宣言 7-2, 7-4, 7-7, 7-8, 7-17, 10-5
互換性 7-3, 10-9
～参照 7-23
スコープ 4-28
バリアントと～ 5-30
比較 4-11
ファイルと～ 5-24
～メソッド 7-1, 7-26
メタクラス 7-23
メモリ 11-10
クラス参照型 7-23
コンストラクタと～ 7-24
バリアントと～ 5-30

比較 4-11
メモリ 11-11
グローバル識別子 4-27
グローバル変数 5-37
DLL 9-6
インターフェース 10-9
動的ロード可能なライブラリ 9-6
メモリ管理 11-2
グローバルユニーク識別子 →「GUID」を参照

け

継承 7-2, 7-3, 7-5
インターフェース 10-2
結合
フィールド 7-7
メソッド 7-10
文字列の〜 4-9
減算 4-6
ポインタの〜 4-9
限定識別子 3-6, 4-2, 4-28, 5-21
Self 7-10
型キャストでの〜 4-14, 4-15
ポインタ 5-26
厳密な型チェック 5-1

こ

構造化型 5-16
バリアントと〜 5-30
ファイルと〜 5-24
レコードと〜 5-23
構造化文 4-19
構造体 5-21
構文
正式な〜 A-1〜A-6
説明 1-2
コメント 4-1, 4-5
コロン（:）→「記号」セクション
コンストラクタ 7-1, 7-9, 7-13
クラス参照と〜 7-24
呼び出し規約 12-4
例外 7-28, 7-32
コンソールアプリケーション 2-3, 8-3
コンパイラ 2-2, 2-3, 2-5, 3-1
検索パス 3-6
コマンドライン〜 2-3〜2-5
〜指令 3-2, 4-5
パッケージ 9-11
コンパイル時バインディング →「静的メソッド」
コンポーネント（クラスの〜）→「メンバー（クラスの〜）」

さ

再帰的呼び出し（手続きと関数）6-1, 6-4
再配置可能な式（アセンブラ）13-13
差（集合の〜）4-10
算術演算子 4-6, 5-4
参照カウント 5-12, 10-9, 11-6, 11-7
参照渡し（パラメータ）6-12, 6-13, 10-12, 12-1

し

式 4-1, 4-5
アセンブラ 13-7〜13-15
識別子 4-1, 4-2, 4-3
グローバル〜とローカル〜 4-27
限定〜 3-6
スコープ 4-27〜4-28
例外ハンドラ内の〜 7-30
実現部 3-3, 3-4, 3-7
forward 宣言と〜 6-6
uses 節 3-7
スコープ 4-28
メソッド 7-9
実行時型情報 →「RTTI」
実行時バインディング →「動的メソッド」,「仮想メソッド」
実行時パッケージ 9-8
実数型 5-9, 11-3
のパブリッシュ 7-5
比較 4-11
変換 4-15
実数（浮動小数点）演算子 4-6
実パラメータ 6-19
シャープ（#）→「記号」セクション
ジャンプ命令（アセンブラ）13-3
集合
演算子 4-10
〜型 5-16
空〜 5-17
〜構成子 4-13
〜のパブリッシュ 7-5
バリアントと〜 5-30
メモリ 11-7
終了処理部 3-3, 3-5, 12-4
終了手続き 9-6, 12-4〜12-5
パッケージと〜 12-4
出力 →「ファイル入出力」
出力（out）パラメータ 6-11, 6-13, 6-19
取得関数 →「read 指定子」
循環参照
パッケージ 9-10
ユニット 3-7〜3-8
順序型 5-3〜5-9
順序値 5-3
列挙型 5-7
上位クラス 7-3
状況感知型ヘルプ（エラー処理）7-32

条件文 4-19
乗算 4-6
ショートサーキット評価 4-7
初期化
DLL 9-6
オブジェクト 7-13
バリアント 5-30, 5-37
ファイル 5-37
変数 5-37
ユニット 3-4
初期化部 3-3, 3-4
例外 7-28
除算 4-6
指令 4-1, 4-3
→「予約語」も参照
一覧 4-3
コンパイラ〜 3-2, 4-5
真の定数 5-39
シンボル 4-1
→「特殊シンボル」，I-1ページの「記号」セクションも参照
アセンブラ 13-10

す

数字 4-1, 4-4
ラベルとしての〜 4-4, 4-18
数値
アセンブラ 13-8
〜型 5-39, 6-8
スコープ 4-27〜4-28
型識別子 5-36
クラス 7-3
レコード 5-22
スタックサイズ 11-2
ストリーミング（データ）7-6
ストリーム処理（データ）5-2
スペース 4-1
スラッシュ（/）→「記号」セクション
スレッド変数 5-38
パッケージ内の〜 9-10

せ

制御（プログラムの〜）6-19, 12-1〜12-5
制御文字 4-1, 4-4
制御文字列 4-4
制御ループ 4-19, 4-24
整数演算子 4-6
整数型 5-3
定数 5-39
データ形式 11-3
比較 4-11
変換 4-15
静的にロードされるライブラリ 9-2
静的配列 5-17, 11-7
バリアントと〜 5-30
静的メソッド 7-10
積（集合）4-10
設計時パッケージ 9-8
絶対アドレス 5-37
絶対式（アセンブラ）13-13
設定関数 →「write 指定子」
セパレータ 4-1, 4-5
セミコロン（;）→「記号」セクション
宣言 4-1, 4-16, 4-27
forward 〜 3-4, 6-5, 7-7, 10-4
インターフェース 3-4
型 5-35
関数 6-1, 6-3
クラス 7-2, 7-7, 7-8, 7-17, 10-5
実装 7-8
定義〜 6-6, 7-7, 7-8, 10-4
定数 5-39, 5-41
手続き 6-1, 6-2
パッケージ 9-9
フィールド 7-7
プロパティ 7-17, 7-19
変数 5-36
メソッド 7-8
ローカル〜 6-10
宣言済みの型 5-1

そ

相互依存クラス 7-7
相互依存ユニット 3-7
添字
var パラメータでの〜 5-33, 6-13
配列の〜 5-18, 5-19, 5-20
バリアント配列 5-33
文字列 5-11
文字列バリアント 5-30
ソースコードファイル 2-1
即値（アセンブラ）13-12

た

大カッコ（[ ]）→「記号」セクション
ダイナミックリンクライブラリ →「DLL」を参照
代入互換性 5-35, 7-3, 10-9
代入文 4-17
型キャスト 4-14, 4-15
タグ（レコード）5-22
多次元配列 5-18, 5-20, 5-41
多態性 7-9, 7-11, 7-14
単価記号（@）→「@」，「アドレス演算子」
単項演算子 4-6
単純型 5-2
単純文 4-17

ち

チェック付き型キャスト
- インターフェース 10-10
- オブジェクト 7-25

中カッコ（{ }）→「記号」セクション
抽象メソッド 7-12

て

定位置パラメータ 10-12
定義済みの型 5-1
定義宣言 6-6, 7-7, 7-8, 10-4
定数 4-6, 5-38
- アセンブラ 13-8
- 型付き〜 5-41
- 型の互換性 5-39
- 真の〜 5-39
- 数値〜 →「数値」を参照
- 宣言された〜 5-38〜5-43
- 手続き型〜 5-42
- 配列 5-41
- ポインタ 5-42
- レコード 5-42

定数式
- case 文 4-23
- 型 5-39, 6-8
- 定数宣言 5-39, 5-41, 5-42
- デフォルトパラメータ 6-18
- 〜とは 5-40
- 配列定数 5-41
- 部分範囲型 5-8, 5-9
- 変数の初期化 5-37
- 列挙型 5-7

定数パラメータ 6-11, 6-13, 6-19, 12-1
- オープン配列コンストラクタ 6-20

ディスパッチ
- メソッド呼び出し 7-11
- メッセージの〜 7-17

ディスパッチインターフェース型 10-11
ディレクトリパス
- uses 節内の〜 3-6

データアラインメント 5-16, 11-8
データ型 →「型」
データ形式（内部〜） 11-2〜11-12
テキストファイル 8-2, 8-3
テキストファイルデバイスドライバ 8-4
デクリメント（順序型変数の〜） 5-3, 5-4, 5-5
デスクトップ設定ファイル 2-3
デストラクタ 7-1, 7-14
- 呼び出し規約 12-4

手続き 3-4, 6-1〜6-20
- アセンブラ 13-15
- 外部〜の呼び出し 6-6
- ネストした〜 5-28, 6-10
- 〜のオーバーロード 6-6, 6-8
- 〜の宣言 6-2, 6-5
- 〜ポインタ 4-12, 5-28
- 〜呼び出し 4-18, 6-1, 6-2, 6-19〜6-20

手続き型 4-15, 5-27〜5-30
- 互換性 5-28
- 代入での〜 5-29
- 〜で DLL を呼び出す 9-2
- デフォルトパラメータ 6-18
- 〜でルーチンを呼び出す 5-29
- メモリ 11-10

手続き型定数 5-42
手続きポインタ 4-12, 5-28
デバイス関数 8-4, 8-5
デバイスドライバ（テキストファイル） 8-4
デフォルトパラメータ 6-11, 6-17〜6-18, 6-19
- forward 宣言とインターフェース宣言での〜 6-18
- オートメーションオブジェクト 10-12
- オーバーロードと〜 6-10, 6-18
- 手続き型 6-18

デフォルトプロパティ 7-20
- COM オブジェクト 5-32
- インターフェース 10-2

デュアルインターフェース 10-3, 10-13
- メソッド 6-5

と

問い合わせ（インターフェース） 10-10
等価演算子 4-11
統合開発環境 →「IDE」
動的配列 5-18, 11-7
- DLL 内の〜 9-8
- オープン配列パラメータと〜 6-15
- 多次元〜 5-20
- 動的ロード可能なライブラリ 9-8
- 〜の切り捨て 5-19
- バリアントと〜 5-30
- 比較 5-19
- ファイルと〜 5-24
- 〜への代入 5-19
- メモリ管理 11-2
- レコードと〜 5-23

動的変数 5-38
- DLL 内の〜 9-8
- 動的ロード可能なライブラリ 9-8
- ポインタ定数と〜 5-43

動的メソッド 7-10, 7-11
動的ロード可能なライブラリ 6-7, 9-1〜9-12
- グローバル変数 9-6
- 静的ロード 9-2
- 動的配列 9-8
- 動的変数 9-8

長い文字列 9-8
〜の作成 9-3
変数 9-1
例外 9-7
トークン 4-1
特殊シンボル 4-1
ドル記号（$）→「記号」セクション

な

内部データ形式 11-2〜11-12
内部ブロック 4-28
長い文字列 4-9, 5-10, 5-12
DLL 内の〜 9-8
動的ロード可能なライブラリ 9-8
ファイルと〜 5-24
メモリ管理 11-2, 11-6
レコードと〜 5-23
名前
→「識別子」も参照 ii
エクスポートされるルーチン（DLL）9-5
関数 6-3, 6-4
識別子 4-16
〜の競合 3-6, 4-28
パッケージ 9-10
プログラム 3-1, 3-2
ユニット 3-3, 3-6
名前付きパラメータ 10-12

に

二項演算子 4-6
入力 →「ファイル入出力」

ぬ

ヌルで終わる文字列 5-13〜5-16, 5-27, 11-6, 11-7
Pascal 文字列との混在 5-15
標準ルーチン 8-6, 8-7
ヌル文字 5-12, 5-13, 11-6, 11-7, 11-10
ヌル文字列 4-4

ね

ネストした条件文 4-22
ネストしたルーチン 5-28, 6-10
ネストした例外 7-31

は

パーティショニング
アプリケーション〜 9-8
配列 5-3, 5-17〜5-20
array of const 6-16
PByteArray でのアクセス 5-27
PWordArray でのアクセス 5-27
インデックス 4-14
オープン配列コンストラクタ 6-17, 6-19
静的〜 5-17, 11-7
代入と〜 5-20
多次元 5-18, 5-20
定数〜 5-41
動的〜 5-18, 6-15, 11-7
パラメータ 6-12, 6-15
バリアントと〜 5-30, 5-32
文字〜 4-5, 5-13, 5-14, 5-16, 5-18
文字配列と文字列定数 5-41
配列プロパティ 7-5, 7-19
格納指定子と〜 7-22
ディスパッチインターフェース内の〜 10-11
デフォルト 7-20
パックレコード 11-8
パッケージ 9-8〜??, 9-8, ??〜9-12
uses 節と〜 9-9
コンパイラ指令 9-11
コンパイラスイッチ 9-12
スレッド変数 9-10
静的にロードする 9-9
動的にロードする 9-9
〜のコンパイル 9-11
〜の宣言 9-9
パッケージファイル 2-2, 2-3, 9-9
パブリックな識別子（インターフェース部）3-4
パラメータ 5-28, 6-2, 6-3, 6-11〜6-18, 6-19
→「オーバーロード手続き/関数」も参照
値〜 6-12, 12-1
オートメーションのメソッド呼び出し 10-12
オーバーロードと〜 6-6, 6-8, 6-9
オープン配列〜 6-15
型可変オープン配列〜 6-16
型付き〜 6-11
型なし〜 6-14, 6-19
仮〜 6-19
実〜 6-19
出力（out）6-13
定位置〜 10-12
定数〜 6-13, 12-1
デフォルト〜 6-17〜6-18, 6-19, 10-12
〜としてのプロパティ 7-18
名前 10-12
〜の受け渡し 12-1
配列 6-12, 6-15
配列プロパティのインデックス 7-20
ファイル 6-12
プログラム制御 12-1
変数（var）〜 6-12, 12-1
短い文字列 6-15
呼び出し規約 6-4
〜リスト 6-11
レジスタ 6-4, 6-5, 12-2

バリアント 5-30～5-33
OleVariant 5-33
Variant 型 5-27, 5-30
インターフェースと～ 10-9, 10-10
演算子 4-6, 5-32
オートメーションと～ 11-12
～型 5-30～5-33
型キャスト 5-31
～の初期化 5-30, 5-37
～配列 5-32
～配列と文字列 5-33
非 Delphi 関数 11-12
ファイルと～ 5-24
変換 5-30, 5-31～5-32
メモリ管理 11-2, 11-11
レコードと～ 5-23
バリアント配列 5-30
範囲チェック 5-4, 5-5, 5-8
汎用型 5-2

ひ

ヒープメモリ 5-38, 11-2
比較
packed 文字列 4-11
PChar 型 4-11
オブジェクト 4-11
関係演算子 4-11
クラス 4-11
クラス参照型 4-11
実数型 4-11
整数型 4-11
動的配列 5-19
文字列 4-11, 5-10
ビジュアルコンポーネントライブラリ → 「VCL」
左シフト（ビット演算子）4-8
ビット演算子 not 4-8
否定 4-7
ビットごとの～ 4-8
表記規則 1-2
標準ルーチン 8-1～8-10
ヌルで終わる文字列 8-6, 8-7
ワイド文字列 8-7

ふ

ファイル
as パラメータ 6-12
～型 5-24, 8-2
型付き～ 5-24, 8-2
型なし～ 5-24, 8-2, 8-4
生成される～ 2-2, 2-3, 9-10, 9-11
ソースコード～ 2-1
テキスト～ 8-2, 8-3
～の初期化 5-37
バリアントと～ 5-30
メモリ 11-9
ファイル入出力 8-1～8-6
例外 8-3
ファイル変数 8-2
フィールド 5-21～5-24, 7-1, 7-7
→「レコード」,「クラス」も参照
～のパブリッシュ 7-6
フォーム 2-2
フォームファイル 2-2, 2-6～2-8, 3-1, 7-5, 7-21
複合文 4-19
複数ユニットの参照 3-6～3-7
符号
型キャストでの～ 4-14
数値 4-4
復帰 4-1, 4-5
不等価演算子 4-11
不等号（<，>）→「記号」セクション
浮動小数点型→「実数型」
太字 1-2
部分集合演算子（>）4-10
部分集合演算子（<）4-10
部分的評価 4-7
部分範囲型 4-7, 4-23, 5-8
プラス（+）→「記号」セクション
プログラム 2-1～2-8, 3-1～3-8
構文 3-1～3-3
例題～ 2-3～2-8
プログラム制御 6-19, 12-1～12-5
プロジェクト 2-6～2-8, 3-6
プロジェクトオプションファイル 2-3
プロジェクトファイル 2-2, 3-1, 3-2, 3-6
ブロック 4-27
try...except 7-28, 7-31
try...finally 7-32
外部～と内部～ 4-28
関数 3-4, 6-1, 6-3
スコープ 4-27～4-28
手続き 3-4, 6-1, 6-2
プログラム 3-1, 3-3
ライブラリ 9-6
プロトタイプ 6-1
プロパティ 7-1, 7-17～7-23
アクセス指定子 7-18
インターフェース 10-4
書き込み専用～ 7-19
デフォルト 7-20, 10-2
～のオーバーライド 7-6, 7-22
～の宣言 7-17, 7-19
配列～ 7-5, 7-19
パラメータとしての～ 7-18
読み出し専用～ 7-19
レコード型～ 7-5

文 4-1, 4-17～4-26, 4-27, 6-1
文法（正式な～）A-1～A-6

へ

べき乗記号（^）→「記号」セクション
ヘッダー
　プログラム 2-1, 3-1, 3-2
　ユニット 3-3
　ルーチン 6-1
変換
　→「型キャスト」も参照
　バリアント 5-30, 5-31～5-32
変数 4-6, 5-36～5-38
　グローバル 5-37, 10-9
　スレッド～ 5-38
　絶対アドレス 5-37
　動的～ 5-38
　動的ロード可能ライブラリからの～ 9-1
　～の初期化 5-37
　～の宣言 5-36
　ヒープに割り当てられる～ 5-38
　ファイル～ 8-2
　メモリ管理 11-2
　ローカル～ 5-37, 6-10
変数（var）パラメータ 6-11, 6-12, 6-19, 12-1
変数型キャスト 4-14, 4-15
変数パラメータ 6-19

ほ

ポインタ 5-25～5-27
　nil ～ 5-26, 11-5
　Pointer 型 4-12, 11-5
　var パラメータでの～ 6-13
　～演算 4-9
　演算子 4-9
　概要 5-25
　型可変オープン配列パラメータでの～ 6-17
　関数～ 4-12, 5-28
　定数～ 5-42
　手続き型 4-12, 5-27～5-30
　長い文字列 5-16
　ヌルで終わる文字列 5-14, 5-16
　バリアントと～ 5-30
　標準の型 5-27
　ファイルと～ 5-24
　ポインタ型 4-12, 5-26, 5-26～5-27, 11-5
　メソッド～ 5-28
　メモリ 11-5
　文字～ 5-27
　レコードと～ 5-23
本体（ルーチン）6-1

ま

マイナス（-）→「記号」セクション
前の順序値 5-3
マルチスレッドアプリケーション 5-38
　DLL 9-7
マルチバイト文字セット 5-12
　文字列処理ルーチン 8-7

み

右シフト（ビット演算子）4-8
短い文字列 5-3, 5-10, 5-11

む

無名値（列挙型）5-7

め

命令コード（アセンブラ）13-2
メインフォーム 2-6～2-8
メソッド 7-1, 7-2, 7-8～7-17
　オートメーション 7-6, 10-12
　仮想～ 7-6, 7-10, 7-11
　クラス～ 7-1, 7-26
　結合 7-10
　コンストラクタ～ 7-13, 12-4
　静的～ 7-10
　抽象～ 7-12
　ディスパッチインターフェースの～ 10-11
　デストラクタ～ 7-14, 12-4
　デュアルインターフェース 6-5
　動的～ 7-10, 7-11
　～のオーバーライド 7-11, 7-12, 10-6
　～のオーバーロード 7-12
　～の実装 7-8
　～のパブリッシュ 7-6
　～ポインタ 4-12, 5-28
　呼び出し規約 12-3
　呼び出しのディスパッチ 7-11
メソッド解決節 10-5, 10-6
メソッドポインタ 4-12, 5-28
メタクラス 7-23
メッセージディスパッチ 7-17
メッセージハンドラ 7-15
　継承した～ 7-16
　～のオーバーライド 7-16
メモリ 4-1, 5-2, 5-25, 5-26, 5-30, 5-37, 7-14
　DLL 9-6
　～管理 11-1～11-12
　共有メモリマネージャ 9-8
　ヒープ 5-38
　レコード内での～の共有 5-23
メモリ参照（アセンブラ）13-12
メンバー

→「集合」も参照
クラス～ 7-1
インターフェース 10-2
可視性 7-4

も

文字
～型 5-4, 11-3
～ポインタ 5-27
文字列リテラルとして 4-5, 5-5
ワイド～ 5-13, 11-3
文字セット
1 バイト～（SBCS）5-12
ANSI ～ 5-4, 5-12, 5-13
Pascal 4-1, 4-2, 4-4
拡張～ 5-12
マルチバイト～（MBCS）5-12
文字の組み合わせ 4-2
モジュール →「ユニット」
文字列 4-1, 4-4, 5-43
→「文字セット」も参照
演算子 4-9, 5-15
～型 5-10～5-16
型可変オープン配列パラメータ 6-17
処理 →「標準ルーチン/ヌルで終わる文字列」
添字 4-14
定数 4-4, 5-43, 13-9
ヌルで終わる～ 5-13～5-16, 5-27
パラメータ 6-15
バリアント 5-30
バリアント配列 5-33
比較 4-11, 5-10
メモリ管理 11-5, 11-6
～リテラル 4-4, 5-43
ワイド～ 5-13, 8-7, 11-2
文字列演算子 4-9
戻り型（関数）6-3, 6-4
戻り値（関数）6-3, 6-4
コンストラクタ 7-13

ゆ

優先順位（演算子）4-12, 7-25
ユニット 2-1, 3-1～3-8
構文 3-3～3-8
スコープ 4-28
ユニットファイル 2-1, 3-1, 3-3
大文字小文字の区別 4-2

よ

呼び出し規約 5-28, 6-4, 12-1
アクセス指定子 6-5, 7-18
インターフェース 10-3, 10-7
動的ロード可能なライブラリ 9-4
メソッド 12-3
呼び出し（ルーチンの～）9-1
読み出し専用プロパティ 7-19
予約語 4-1, 4-2, 4-3
→「指令」も参照
アセンブラ 13-6
一覧 4-3

ら

ライブラリ →「DLL」,「動的ロード可能なライブラリ」,「パッケージ」を参照
ライブラリ検索パス 3-6
ラベル 4-1, 4-4, 4-18
アセンブラ 13-2

り

リソースファイル 2-2, 2-3, 3-2
リソース文字列 5-40

る

ルーチン 6-1～6-20
→「関数」,「手続き」も参照
～のエクスポート 9-5
標準～ 8-1～8-10
ループ文 4-19, 4-24

れ

例外 4-19, 7-14, 7-15, 7-26～7-32
DLL 9-6, 9-7
コンストラクタ 7-28, 7-32
初期化部での～ 7-28
～処理 7-28, 7-29, 7-30, 7-31, 7-32
動的ロード可能なライブラリ 9-7
ネストした～ 7-31
～の再生成 7-31
～の生成 7-28
～の宣言 7-27
～の伝播 7-29, 7-31, 7-32
～の破棄 7-28, 7-30
標準ルーチン 7-32
標準例外 7-32
ファイル入出力 8-3
例外ハンドラ 7-26, 7-29
～内の識別子 7-30
例題プログラム 2-3～2-8
レコード 4-20, 5-21～5-24
～型 5-21
可変部 5-22～5-24
スコープ 4-28, 5-22
定数 5-42
バリアントと～ 5-30
プロパティ内の～ 7-5

メモリ 11-8
レジスタ 6-4, 6-5, 12-2, 12-3
アセンブラ 13-2, 13-9, 13-12, 13-16
集合の格納 11-7
列挙型 5-6～5-8, 11-3
パブリッシュの設定 7-5
無名値 5-7

ろ

ローカル識別子 4-27
ローカル変数 5-37, 6-10
メモリ管理 11-2
論理演算子 4-7
完全評価と部分的評価 4-7
論理型 5-5, 11-3
論理積 4-7
論理（ビット）演算子 4-8
論理和 4-7
ビットごとの～ 4-8

わ

ワイド文字/文字列 5-13
標準ルーチン 8-7
メモリ管理 11-2
和（集合）4-10