学校的理想装备

电子图书·学校专集

校园网上的最佳资源

Object Pascal Language Guide

Object Pascal Language Guide

Delphi for Windows

Introduction

Copyright Agreement

This manual is about the Object Pascal language as it is used in Delphi. For an overview of the Delphi documentation, see the introduction to the Delphi User's Guide. This manual

- Presents the formal definition of the Object Pascal language
- Describes what goes on inside an Object Pascal application in regard to memory, data formats, calling conventions, input and output, and automatic optimizations
- Explains how to use assembly language in Object Pascal applications
- Introduces the command-line compiler for Delphi
- Explains the use of compiler directives
- Presents explanations of error messages

What's in this manual?

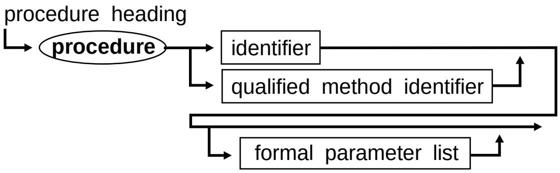
This manual contains several different kinds of information:

- The first twelve chapters define the Object Pascal language, examining each element of a program in detail.
- The next six chapters present technical information for advanced users about input and output, numerical processing, memory management, program control, and optimization.
- The last two chapters explain how to use the built-in assembler and how to link assembly-language code into your Delphi applications.

At the end of the book there are three appendixes, describing the use of the command-line compiler, compiler directives, and error messages.

Syntax Diagrams

In this book you'll encounter *syntax diagrams*. For example:



To read a syntax diagram, follow the arrows. Frequently, more than one path is possible. The above diagram indicates that a formal parameter list is optional in a procedure heading. You can follow the path from the indentifier to the end of the procedure heading, or you can follow it to the formal parameter list before reaching the end.

The names in the boxes stand for constructions. Those in circles — reserved words. operators, and punctuation — are actual terms used in the programs; they are boldfaced in the diagrams.

Chapter

1

Tokens

Tokens are the smallest meaningful units of text in an Object Pascal program. They are categorized as special symbols, identifiers, labels, numbers, and string constants.

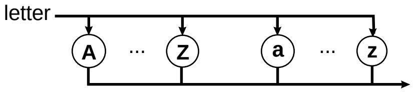
An Object Pascal program is made up of tokens and *separators*. A separator is either a blank or a comment. Two adjacent tokens must be separated by one or more separators if each token is a reserved word, an identifier, a label, or a number. Separators can't be part of tokens except in string constants.

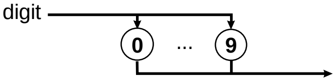
Special symbols

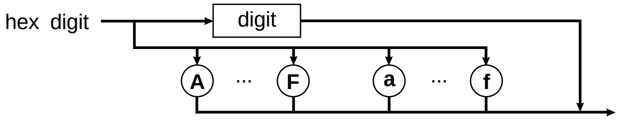
Object Pascal uses the following subsets of the ASCII character set:

- **Letters**—the English alphabet, *A* through *Z* and *a* through *z*
- **Digits**—the Arabic numerals 0 through 9
- **Hex digits**—the Arabic numerals 0 through 9, the letters *A* through *F*, and the letters *a* through *f*
- **Blanks**—the space character (ASCII 32) and all ASCII control characters (ASCII 0 through 31), including the end-of-line or return character (ASCII 13)

These are the syntax diagrams for letter, digit, and hex digit:







Special symbols and reserved words are characters that have one or more fixed meanings. The following single characters are special symbols:

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

These character pairs are also special symbols:

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

A left bracket (**[**) is equivalent to the character pair of left parenthesis and a period—**(.**, and a right bracket (**]**) is equivalent to the character pair of a period and a right parenthesis—**.)**. Likewise, a left brace (**{**) is equivalent to the character pair of left parenthesis and an asterisk—**(***, and a right brace (**}**) is equivalent to the character pair of an asterisk and a right parenthesis—***)**.

Reserved words and standard directives

Reserved words can't be redefined.

Reserved words appear in lowercase **boldface** throughout this manual. Object Pascal is *not* case sensitive, however, so you can use either uppercase or lowercase letters in your programs.

Following are Obejct Pascal's reserved words:

Table 1-1 Object Pascal reserved words

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

The following are Object Pascal's standard directives. Directives are used only in contexts where user-defined identifiers can't occur. Unlike reserved words, you can redefine standard directives, but we advise that you don't.

Table 1-2 Object Pascal directives

absolute
abstract
assembler
at
cdecl
default
dynamic
export
external
far
forward
index
interrupt
message
name
near
nodefault
override
private
protected
public
published
read
resident
stored
virtual
write

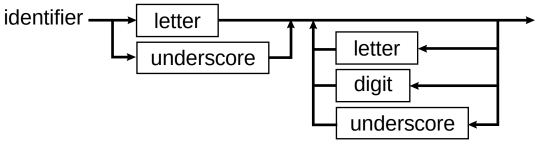
private, **protected**, **public**, and **published** act as reserved words within object type declarations, but are otherwise treated as directives.

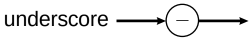
Identifiers

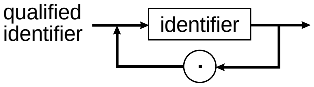
Identifiers denote constants, types, variables, procedures, functions, units, programs, and fields in records.

An identifier can be of any length, but only the first 63 characters are significant. An identifier must begin with a letter or an underscore character (_) and can't contain spaces. Letters, digits, and underscore characters (ASCII $5F) are allowed after the first character. Like reserved words, identifiers are *not* case sensitive.

When several instances of the same identifier exist, you may need to qualify the identifier by another identifier to select a specific instance. For example, to qualify the identifier *Ident* by the unit identifier *UnitName*, write *UnitName.Ident*. The combined identifier is called a *qualified identifier*. Units are described on page 206 of the Delphi *User's Guide* and Chapter 11 of this manual.







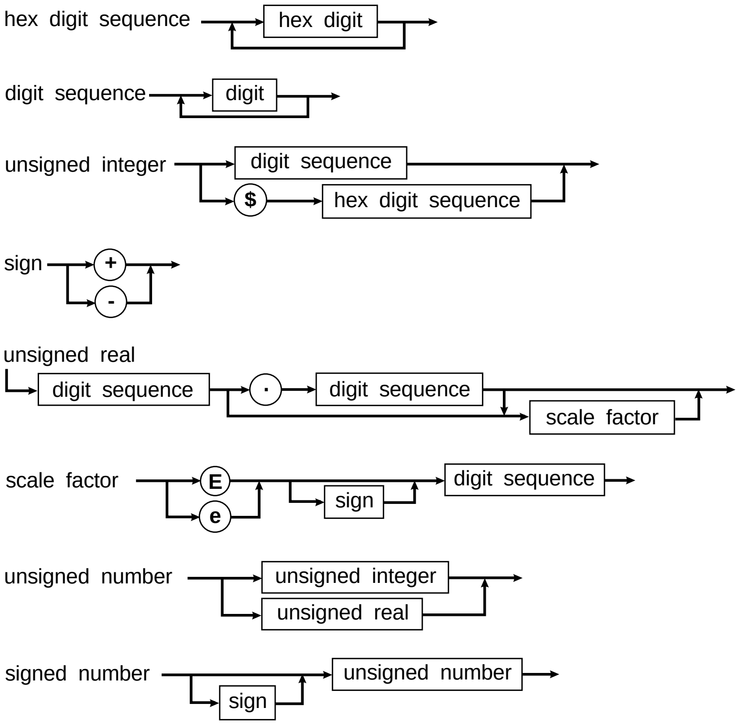
Here are some examples of identifiers and qualified identifiers:

```
Writeln
Exit
Real2String
System.MemAvail
SysUtils.StrLen
WinCrt.ReadText
```

In this manual, standard and user-defined identifiers are *italicized* when they are referred to in text.

Numbers

Ordinary decimal notation is used for numbers that are constants of integer and real types. A hexadecimal integer constant uses a dollar sign ($) as a prefix. Engineering notation (E or e, followed by an exponent) is read as “times ten to the power of” in real types. For example, 7E-2 means 7 x 10^{-2}; 12.25e+6 or 12.25e6 both mean 12.25 x 10^{+6}. Syntax diagrams for writing numbers follow:

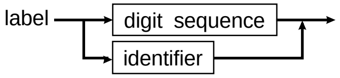


Numbers with decimals or exponents denote real-type constants. Other decimal numbers denote integer-type constants; they must be within the range -2,147,483,648 to 2,147,483,647.

Hexadecimal numbers denote integer-type constants; they must be within the range $00000000 to $FFFFFFFF. The resulting value’s sign is implied by the hexadecimal notation.

Labels

A label is a digit sequence in the range 0 to 9999. Leading zeros are not significant. Labels are used with **goto** statements.



As an extension to Standard Pascal, Object Pascal allows identifiers to function as labels.

Character strings

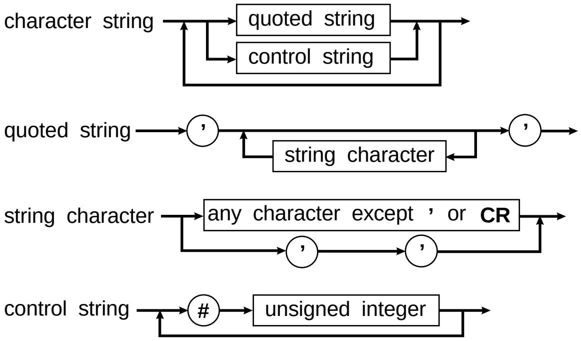
A character string is a sequence of zero or more characters from the extended ASCII character set, written on one line in the program and enclosed by apostrophes.

A character string with nothing between the apostrophes is a *null string*. Two sequential apostrophes in a character string denote a single character, an apostrophe. For example,

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

Object Pascal lets you embed control characters in character strings. The # character followed by an unsigned integer constant in the range 0 to 255 denotes a character of the corresponding ASCII value. There must be no separators between the # character and the integer constant. Likewise, if several are part of a character string, there must be no separators between them. For example,

```
#13#10
'Line 1'#13'Line2'
#7#7'Wake up!'#7#7
```



A character string’s *length* is the actual number of characters in the string. A character string of any length is compatible with any string type, and with the *PChar* type when the extended syntax is enabled {**$X+**}. Also, a character string of length one is compatible with any *Char* type, and a character string of length *N*, where *N* is greater than or equal to one, is compatible with packed arrays of *N* characters.

Comments

The following constructs are comments and are ignored by the compiler:

```
{ Any text not containing right brace }
(* Any text not containing star/right parenthesis *)
```

A comment that contains a dollar sign (**$**) immediately after the opening **{** or **(*** is a *compiler directive*. A mnemonic of the compiler command follows the **$** character.

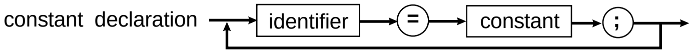
Program lines

Object Pascal program lines have a maximum length of 126 characters.

Chapter 2

Constants

A constant is an identifier that marks a value that can’t change. A *constant declaration* declares a constant within the block containing the declaration. A constant identifier can’t be included in its own declaration.

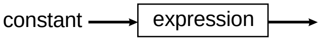


Object Pascal allows the use of *constant expressions*. A constant expression is an expression that can be evaluated by the compiler without actually executing the program. Wherever Standard Pascal allows only a simple constant, Object Pascal allows a constant expression.

Examples of constant expressions follow:

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

The simplest case of a constant expression is a simple constant, such as 100 or ‘A’.



Because the compiler has to be able to completely evaluate a constant expression at compile time, the following constructs are *not* allowed in constant expressions:

- References to variables and typed constants (except in constant address expressions as described on page 35)
- Function calls (except those noted in the following text)

- The address operator (@) (except in constant address expressions as described on page 35)

Except for these restrictions, constant expressions follow the syntactical rules as ordinary expressions. For expression syntax, see Chapter 5, “Expressions.”

The following standard functions are allowed in constant expressions:

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

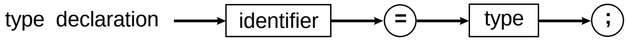
Here are some examples of the use of constant expressions in constant declarations:

```
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;
```

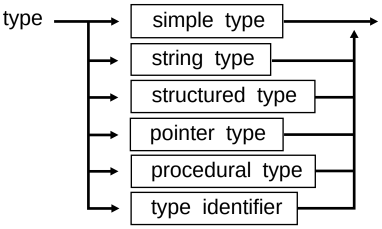
Chapter 3

Types

When you declare a variable, you must state its *type*. A variable's type circumscribes the set of values it can have and the operations that can be performed on it. A *type declaration* specifies the identifier that denotes a type.



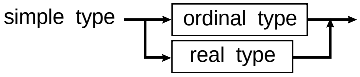
When an identifier occurs on the left side of a type declaration, it's declared as a *type identifier* for the block in which the type declaration occurs. A type identifier's scope doesn't include itself except for pointer types.

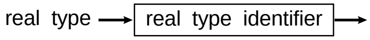


There are six major type classes. They are described in the following sections.

Simple types

Simple types define ordered sets of values.





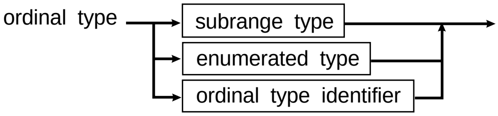
A real type identifier is one of the standard identifiers: *Real*, *Single*, *Double*, *Extended*, or *Comp*. Chapter 1 explains how to denote constant integer type and real type values.

Ordinal types

Ordinal types are a subset of simple types. All simple types other than real types are ordinal types, which are set off by six characteristics:

- All possible values of a given ordinal type are an ordered set, and each possible value is associated with an *ordinality*, which is an integral value. Except for integer type values, the first value of every ordinal type has ordinality 0, the next has ordinality 1, and so on for each value in that ordinal type. The ordinality of an integer type value is the value itself. In any ordinal type, each value other than the first has a predecessor, and each value other than the last has a successor based on the ordering of the type.
- The standard function *Ord* can be applied to any ordinal type value to return the ordinality of the value.
- The standard function *Pred* can be applied to any ordinal type value to return the predecessor of the value. If applied to the first value in the ordinal type and if range checking is enabled {**$R+**}, *Pred* produces a run-time error.
- The standard function *Succ* can be applied to any ordinal type value to return the successor of the value. If applied to the last value in the ordinal type and if range checking is enabled {**$R+**}, *Succ* produces a run-time error.
- The standard function *Low* can be applied to an ordinal type identifier and to a variable reference of an ordinal type. The result is the lowest value in the range of the given ordinal type.
- The standard function *High* can be applied to an ordinal type identifier and to a variable reference of an ordinal type. The result is the highest value in the range of the given ordinal type.

The syntax of an ordinal type follows:



Object Pascal has twelve predefined ordinal types: *Integer*, *Shortint*, *Smallint*, *Longint*, *Byte*, *Word*, *Cardinal*, *Boolean*, *ByteBool*, *WordBool*, *LongBool*, and *Char*. In addition, there are two other classes of user-defined ordinal types: enumerated types and subrange types.

Integer types

Object Pascal's predefined integer types are divided into two categories: *fundamental* types and *generic* types. The range and format of the fundamental types is independent of the underlying CPU and operating system and does not change across different implementations of Object Pascal. The range and format of the generic types, on the other hand, depends on the underlying CPU and operating system.

The fundamental integer types are *Shortint*, *Smallint*, *Longint*, *Byte*, and *Word*. Each fundamental integer type denotes a specific subset of the whole numbers, according to the following table:

Table 3-1 Fundamental integer types

Type	Range	Format
Shortint	-128..127	Signed 8-bit
Smallint	-32768..32767	Signed 16-bit
Longint	-2147483648..2147483647	Signed 32-bit
Byte	0..255	Unsigned 8-bit
Word	0..65535	Unsigned 16-bit

The generic integer types are *Integer* and *Cardinal*. The *Integer* type represents a generic signed integer, and the *Cardinal* type represents a generic unsigned integer. The actual ranges and storage formats of the *Integer* and *Cardinal* vary across different implementations of Object Pascal, but are generally the ones that result in the most efficient integer operations for the underlying CPU and operating system.

Table 3-2 Generic integer types for 16-bit implementations of Object Pascal

Type	Range	Format
Integer	-32768..32767	Signed 16-bit
Cardinal	0..65535	Unsigned 16-bit

Table 3-3 Generic integer types for 32-bit implementations of Object Pascal

Type	Range	Format
Integer	-2147483648..2147483647	Signed 32-bit
Cardinal	0..2147483647	Unsigned 32-bit

Applications should use the generic integer formats whenever possible, since they generally result in the best performance for the underlying CPU and operating system. The fundamental integer types should be used only when the actual range and/or storage format matters to the application.

Arithmetic operations with integer-type operands use 8-bit, 16-bit, or 32-bit precision, according to the following rules:

- The type of an integer constant is the predefined integer type with the smallest range that includes the value of the integer constant.
- For a binary operator (an operator that takes two operands), both operands are converted to their common type before the operation. The common type is the predefined integer type with the smallest range that includes all possible values of both types. For example, the common type of *Smallint* and *Byte* is *Smallint*, and the common type of *Smallint* and *Word* is *Longint*. The operation is performed using the precision of the common type, and the result type is the common type.
- The expression on the right of an assignment statement is evaluated independently from the size or type of the variable on the left.

- Any byte-sized operand is converted to an intermediate word-sized operand that is compatible with both *Smallint* and *Word* before any arithmetic operation is performed.

An integer-type value can be explicitly converted to another integer type through typecasting. Typecasting is described in Chapters 4 and 5.

Boolean types

There are four predefined boolean types: *Boolean*, *ByteBool*, *WordBool*, and *LongBool*. Boolean values are denoted by the predefined constant identifiers *False* and *True*. Because booleans are enumerated types, these relationships hold:

- *False* < *True*
- *Ord*(*False*) = 0
- *Ord*(*True*) = 1
- *Succ*(*False*) = *True*
- *Pred*(*True*) = *False*

Boolean and *ByteBool* variables occupy one byte, a *WordBool* variable occupies two bytes (one word), and a *LongBool* variable occupies four bytes (two words). *Boolean* is the preferred type and uses the least memory; *ByteBool*, *WordBool*, and *LongBool* primarily exist to provide compatibility with other languages and the Windows environment.

A *Boolean* variable can assume the ordinal values 0 and 1 only, but variables of type *ByteBool*, *WordBool*, and *LongBool* can assume other ordinal values. An expression of type *ByteBool*, *WordBool*, or *LongBool* is considered *False* when its ordinal value is zero, and *True* when its ordinal value is nonzero. Whenever a *ByteBool*, *WordBool*, or *LongBool* value is used in a context where a *Boolean* value is expected, the compiler will automatically generate code that converts any nonzero value to the value *True*.

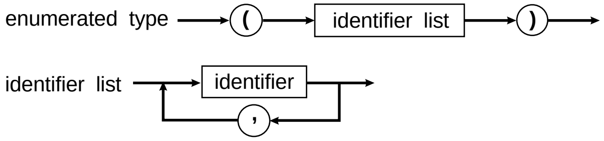
Char type

Char's set of values are characters, ordered according to the extended ASCII character set. The function call *Ord(Ch)*, where *Ch* is a *Char* value, returns *Ch*'s ordinality.

A string constant of length 1 can denote a constant character value. Any character value can be generated with the standard function *Chr*.

Enumerated types

Enumerated types define ordered sets of values by enumerating the identifiers that denote these values. Their ordering follows the sequence the identifiers are enumerated in.



When an identifier occurs within the identifier list of an enumerated type, it's declared as a constant for the block the enumerated type is declared in. This constant's type is the enumerated type being declared.

An enumerated constant's ordinality is determined by its position in the identifier list it's declared in. The enumerated type it's declared in becomes the constant's type. The first enumerated constant in a list has an ordinality of zero.

Here's an example of an enumerated type:

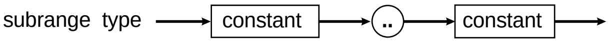
```
type
  Suit = (Club, Diamond, Heart, Spade);
```

Given these declarations, *Diamond* is a constant of type *Suit*.

When the *Ord* function is applied to an enumerated type's value, *Ord* returns an integer that shows where the value falls with respect to the other values of the enumerated type. Given the preceding declarations, *Ord(Club)* returns zero, *Ord(Diamond)* returns 1, and so on.

Subrange types

A subrange type is a range of values from an ordinal type called the *host type*. The definition of a subrange type specifies the smallest and the largest value in the subrange; its syntax follows:



Both constants must be of the same ordinal type. Subrange types of the form *A..B* require that *A* is less than or equal to *B*.

These are examples of subrange types:

```
0..99
-128..127
Club..Heart
```

A variable of a subrange type has all the properties of variables of the host type, but its run-time value must be in the specified interval.

One syntactic ambiguity arises from allowing constant expressions where Standard Pascal only allows simple constants. Consider the following declarations:

```
const
  X = 50;
  Y = 10;
type
  Color = (Red, Green, Blue);
  Scale = (X - Y) * 2..(X + Y) * 2;
```

Standard Pascal syntax dictates that, if a type definition starts with a parenthesis, it's an enumerated type, such as the *Color* type in the previous example. The intent of the declaration of *scale* is to define a subrange type, however. The solution is to reorganize the first subrange expression so that it doesn't start with a parenthesis, or to set another constant equal to the value of the expression and use that constant in the type definition:

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

Real types

A real type has a set of values that is a subset of real numbers, which can be represented in floating-point notation with a fixed number of digits. A value's floating-point notation normally comprises three values—*M*, *B*, and *E*—such that $M \times B^E = N$, where *B* is always 2, and both *M* and *E* are integral values within the real type's range. These *M* and *E* values further prescribe the real type's range and precision.

There are five kinds of real types: *Real*, *Single*, *Double*, *Extended*, and *Comp*. The real types differ in the range and precision of values they hold as shown in the following table:

Table 3-4 Real data types

Type	Range	Significant digits	Size in bytes
Real	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	3.4×10^{-4932} .. 1.1×10^{4932}	19-20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	19-20	8

The Comp type holds only integral values within $-2^{63}+1$ to $2^{63}-1$, which is approximately -9.2×10^{18} to 9.2×10^{18}.

Object Pascal supports two models of code generation for performing real-type operations: *software* floating point and *80x87* floating point. The **$N** compiler directive is used to select the appropriate model.

80x87 floating point

In the {**$N+**} state, which is selected by default, the generated code performs all real-type calculations using 80x87 instructions and can use all five real types. For more details on 80x87 floating-point code generation, refer to Chapter 14, "Using the 80x87."

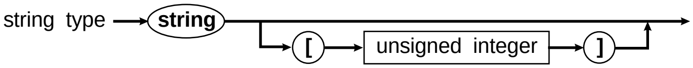
Software floating point

In the {**$N-**} state, the generated code performs all real-type calculations in software by calling run-time library routines. For reasons of speed and code size, only operations on variables of type *Real* are allowed in this state. Any attempt to compile statements that operate on the *Single*, *Double*, *Extended*, and *Comp* types generates an error.

Note The Delphi Visual Class Library requires that you compile your applications in the {**$N+**} state. Unless you are compiling an application that doesn't use VCL, you should refrain from using the {**$N–**} state.

String types

A string-type value is a sequence of characters with a dynamic length attribute (depending on the actual character count during program execution), and a constant size attribute from 1 to 255. A string type declared without a size attribute is given the default size attribute 255. The length attribute's current value is returned by the standard function *Length*. Operators for the string types are described in the sections "String operator" and "Relational operators" in Chapter 5.



The ordering between any two string values is set by the ordering relationship of the character values in corresponding positions. In two strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a higher or greater-than value; for example, 'xs' is greater than 'x'. Null strings can be equal only to other null strings, and they hold the least string values.

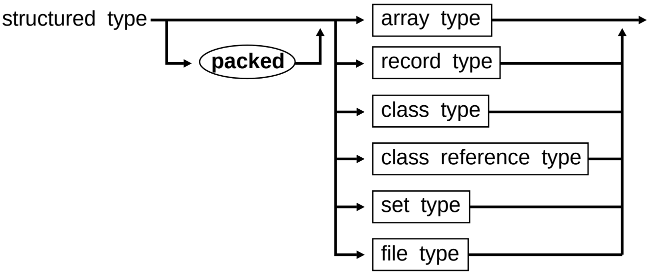
Characters in a string can be accessed as components of an array. See "Arrays, strings, and indexes" on page 32.

The *Low* and *High* standard functions can be applied to a string-type identifier and to a variable reference of a string type. In this case, *Low* returns zero, and *High* returns the size attribute (maximum length) of the given string.

A variable parameter declared using the *OpenString* identifier, or using the **string** keyword in the {**$P+**} state, is an *open string parameter*. Open string parameters allow string variables of varying sizes to be passed to the same procedure or function. Read about open string parameters on page 78.

Structured types

A structured type, characterized by its structuring method and by its component type(s), holds more than one value. If a component type is structured, the resulting structured type has more than one level of structuring. A structured type can have unlimited levels of structuring, but the maximum permitted size of any structured type in Object Pascal is 65,520 bytes.

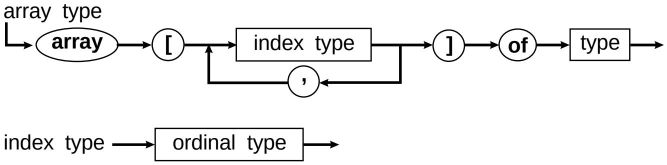


In Standard Pascal, the word **packed** in a structured type's declaration tells the compiler to compress data storage, even at the cost of diminished access to a component of a variable of this type. In Object Pascal, however, **packed** has no effect; instead packing occurs automatically whenever possible.

Class types and class reference types are the cornerstones of object oriented programming in Object Pascal. They are described in full in Chapter 9, "Classes".

Array types

Arrays have a fixed number of components of one type—the *component type*. In the following syntax diagram, the component type follows the word **of**.



The index types, one for each dimension of the array, specify the number of elements. Valid index types are all ordinal types except *Longint* and subranges of *Longint*. The array can be indexed in each dimension by all values of the corresponding index type; therefore, the number of elements is the product of the number of values in each index type.

The following is an example of an array type:

```
array[1..100] of Real
```

If an array type's component type is also an array, you can treat the result as an array of arrays or as a single multidimensional array. For example,

```
array[Boolean] of array[1..10] of array[Size] of Real
```

is interpreted the same way by the compiler as

```
array[Boolean,1..10,Size] of Real
```

You can also express

```
packed array[1..10] of packed array[1..8] of Boolean
```

as

```
packed array[1..10,1..8] of Boolean
```

You access an array's components by supplying the array's identifier with one or more indexes in brackets. See "Arrays, strings, and indexes" on page 32.

When applied to an array-type identifier or a variable reference of an array type, the *Low* and *High* standard functions return the low and high bounds of the index type of the array.

An array type of the form

```
packed array[M..N] of Char
```

where *M* is less than *N* is called a *packed string type* (the word **packed** can be omitted because it has no effect in Object Pascal). A packed string type has certain properties not shared by other array types, as explained below. See "Identical and compatible types" on page 24.

An array type of the form

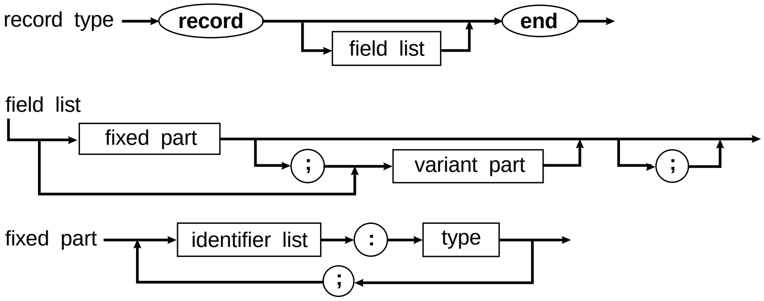
```
array[0..X] of Char
```

where *X* is a positive nonzero integer is called a *zero-based character array*. Zero-based character arrays are used to store *null-terminated strings*, and when the extended syntax is enabled (using a {**$X+**} compiler directive), a zero-based character array is compatible with a *PChar* value. For a complete discussion of this topic, read Chapter 15, "Using null-terminated strings," beginning on page 153.

A parameter declared using the **array of** *T* syntax is an *open array parameter*. Open array parameters allow arrays of varying sizes to be passed to the same procedure or function. Read about open array parameters on page 79.

Record types

A record type comprises a set number of components, or fields, that can be of different types. The record-type declaration specifies the type of each field and the identifier that names the field.

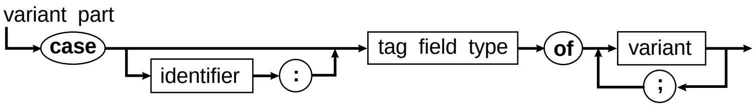


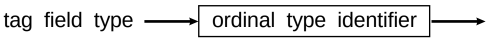
The fixed part of a record type sets out the list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

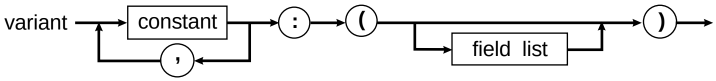
The following is an example of a record type:

```
type
  TDateRec = record
    Year: Integer;
    Month: 1..12;
    Day: 1..31;
  end;
```

The variant part shown in the syntax diagram of a record-type declaration distributes memory space for more than one list of fields, so the information can be accessed in more ways than one. Each list of fields is a *variant*. The variants overlay the same space in memory, and all fields of all variants can be accessed at all times.







You can see from the diagram that each variant is identified by at least one constant. All constants must be distinct and of an ordinal type compatible with the tag field type. Variant and fixed fields are accessed the same way.

An optional identifier, the *tag field identifier*, can be placed in the variant part. If a tag field identifier is present, it becomes the identifier of an additional fixed field—the tag field—of the record. The program can use the tag field's value to show which variant is active at a given time. Without a tag field, the program selects a variant by another criterion.

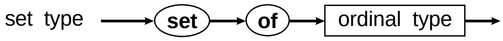
Some record types with variants follow:

```
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: TDate;
        ExitDate: TDate);
  end;

  TPolygon = record
    X, Y: Real;
    case Kind: Figure of
      TRectangle: (Height, Width: Real);
      TTriangle: (Side1, Side2, Angle: Real);
      TCircle: (Radius: Real);
  end;
```

Set types

A set type's range of values is the power set of a particular ordinal type (the base type). The power set is the set of all possible subsets of values of the base type including the empty set. Therefore, each possible value of a set type is a subset of the possible values of the base type.

A variable of a set type can hold from none to all the values of the set. Set-type operators are described in the section "Set operators" in Chapter 5. "Set constructors" in the same chapter shows how to construct set values.

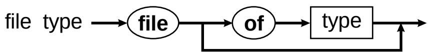


The base type must not have more than 256 possible values, and the ordinal values of the upper and lower bounds of the base type must be within the range 0 to 255.

Every set type can hold the value [ ], which is called the empty set.

File types

A file type consists of a linear sequence of components of the component type, which can be of any type except a file type, any structured type with a file-type component, or an object type. The number of components isn't set by the file-type declaration.

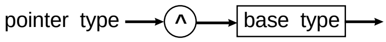


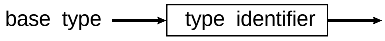
If the word **of** and the component type are omitted, the type denotes an untyped file. Untyped files are low-level I/O (input/output) channels primarily used for direct access to any disk file regardless of its internal format.

The standard file type *Text* signifies a file containing characters organized into lines. Text files use special I/O procedures, which are discussed in Chapter 13, "Input and output."

Pointer types

A pointer type defines a set of values that point to dynamic variables of a specified type called the *base type*. A pointer-type variable contains the memory address of a dynamic variable.





If the base type is an undeclared identifier, it must be declared in the same type declaration part as the pointer type.

You can assign a value to a pointer variable with the *New* procedure, the @ operator, the *Ptr* function, or the *GetMem* procedure. *New* allocates a new memory area in the application heap for a dynamic variable and stores the address of that area in the pointer variable. The @ operator directs the pointer variable to the memory area containing any existing variable or procedure or function entry point, including variables that already have identifiers. *Ptr* points the pointer variable to a specific memory address. *GetMem* creates a new dynamic variable of a specified size, and puts the address of the block in the pointer variable.

The reserved word denotes a pointer-valued constant that doesn't point to anything.

Type Pointer

The predefined type *Pointer* denotes an untyped pointer; that is, a pointer that doesn't point to any specific type. Variables of type *Pointer* can't be dereferenced; writing the pointer symbol ^ after such a variable is an error. Generic pointers, however, can be typecast to allow dereferencing. Like the value denoted by the word **nil**, values of type *Pointer* are compatible with all other pointer types. For the syntax of referencing the dynamic variable pointed to by a pointer variable, see Chapter 4's section entitled "Pointers and dynamic variables" on page 33.

Type PChar

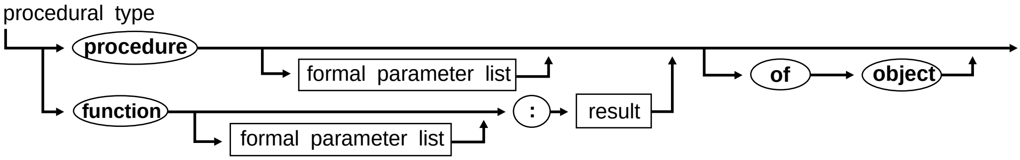
Object Pascal has a predefined type, *PChar*, to represent a pointer to a null-terminated string. The *System* unit declares *PChar* as

```
type PChar = ^Char;
```

Object Pascal supports a set of *extended syntax* rules to facilitate handling of null-terminated strings using the *PChar* type. For a complete discussion of this topic, see Chapter 15, "Using null-terminated strings."

Procedural types

Object Pascal allows procedures and functions to be treated as entities that can be assigned to variables and passed as parameters. Such actions are made possible through *procedural types.*



The syntax for a procedural-type declaration is the same as a that of a procedure or function header, except that the identifier after the **procedure** or **function** keyword is omitted.

There are two categories of procedural types: *Global procedure pointers* and *method pointers*.

Global procedure pointers

A procedural type declared without the **of object** clause is called a global procedure pointer. A global procedure pointer can reference a global procedure or function, and is encoded as a pointer that stores the address of a global procedure or function. Some examples of global procedure pointer types follow:

```
type
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
```

```
    TMathFunc = function(X: Double): Double;
```

Method pointers

A procedural type declared with the **of object** clause is called a method pointer. A method pointer can reference a procedure or function method of an object, and is encoded as two pointers. The first pointer stores the address of a method, and the second pointer stores a reference to the object that the method belongs to. Some examples of method pointer types follow:

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

Procedural values

A variable of a procedural type can be assigned a *procedural value*. Procedural values can be one of the following:

- The value **nil**
- A variable reference of a procedural type
- A global procedure or function identifier
- A method designator

In the context of procedural values, a global procedure or function identifier denotes a global procedure pointer value, and a method designator denotes a method pointer value. For example, given the following declarations:

```
  type
    TMainForm = class(TForm)
      procedure ButtonClick(Sender: TObject);
      ...
    end;

  var
    MainForm: TMainForm;
    MathFunc: TMathFunc;
    OnClick: TNotifyEvent;

  function Tan(Angle: Double): Double; far;
  begin
    Result := Sin(Angle) / Cos(Angle);
  end;
```

The variables *MathFunc* and *OnClick* can be assigned values as follows:

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

and calls can be made using *MathFunc* and *OnClick* as follows:

```
  X := MathFunc(X);    { Equivalent to X := Tan(X) }
  OnClick(Self);       { Equivalent to MainForm.ButtonClick(Self) }
```

Using a procedural variable that contains the value **nil** in a procedure statement or function call results in an error. The value **nil** is intended to indicate that a procedural variable is unassigned, and whenever there is a possibility that a procedural value is **nil**, procedure statements or function calls involving that procedural variable should be guarded by a test:

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

The *Assigned* standard function returns *True* if the given procedural variable has been assigned a procedural value, or *False* if the procedural variable contains **nil**.

Procedural type compatibility

To be considered compatible, procedural types must have the same number of parameters, and parameters in corresponding positions must be of identical types. Finally, the result types of functions must be identical. Parameter names have no significance when determining procedural-type compatibility.

The value **nil** is compatible with any procedural type.

Global procedure pointer types and method pointer types are always mutually incompatible. In other words, a global procedure or function cannot be assigned to a method pointer variable, and a method cannot be assigned to a global procedure pointer variable.

To be used as procedural values, global procedures and functions must be declared with a **far** directive or compiled in the {**$F+**} state. Also, standard procedures and functions, nested procedures and functions, and **inline** procedures and functions can't be used as procedural values.

Standard procedures and functions are the ones declared by the *System* unit, such as *WriteLn*, *ReadLn*, *Chr*, and *Ord*. To use a standard procedure or function as a procedural value, write a "shell" around it. For example, the following function *FSin* is assignment-compatible with the *TMathFunc* type declared above.

```
function FSin(X: Real): Real; far;
begin
  FSin := Sin(X);
end;
```

A procedure or function is *nested* when it's declared within another procedure or function. Such nested procedures and functions can't be used as procedural values.

Identical and compatible types

Two types can be the same, and this sameness (identity) is mandatory in some contexts. At other times, the two types need only be compatible or merely assignment-compatible. They are identical when they are declared with, or their definitions stem from, the same type identifier.

Type identity

Type identity is required only between actual and formal variable parameters in procedure and function calls.

Two types—say, *T1* and *T2*—are identical if one of the following is true: *T1* and *T2* are the same type identifier; *T1* is declared to be equivalent to a type identical to *T2*.

The second condition connotes that *T1* doesn't have to be declared directly to be equivalent to *T2*. The type declarations

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

result in *T1*, *T2*, *T3*, *T4*, and *Integer* as identical types. The type declarations

```
T5 = set of Char;
T6 = set of Char;
```

don't make *T5* and *T6* identical because **set of** Char isn't a type identifier. Two variables declared in the same declaration, for example,

```
V1, V2: set of Char;
```

are of identical types—unless the declarations are separate. The declarations

```
V1: set of Char;
V2: set of Char;
V3: Integer;
V4: Integer;
```

mean *V3* and *V4* are of identical type, but not *V1* and *V2*.

Type compatibility

Compatibility between two types is sometimes required, such as in expressions or in relational operations. Type compatibility is important, however, as a precondition of assignment compatibility.

Type compatibility exists when at least one of the following conditions is true:

- Both types are the same.
- Both types are real types.
- Both types are integer types.
- One type is a subrange of the other.
- Both types are subranges of the same host type.
- Both types are set types with compatible base types.
- Both types are packed string types with an identical number of components.

- One type is a string type and the other is either a string type, packed string type, or *Char* type.
- One type is *Pointer* and the other is any pointer type.
- Both types are class types or class reference types, and one type is derived from the other.
- One type is *PChar* and the other is a zero-based character array of the form **array**[0..X] **of** *Char*. (This applies only when extended syntax is enabled with the {**$X+**} directive.)
- Both types are pointers to identical types. (This applies only when type-checked pointers are enabled with the {**$T+**} directive.)
- Both types are procedural types with identical result types, an identical number of parameters, and a one-to-one identity between parameter types.

Assignment compatibility

Assignment compatibility is necessary when a value is assigned to something, such as in an assignment statement or in passing value parameters.

A value of type T_2 is assignment-compatible with a type T_1 (that is, $T_1 := T_2$ is allowed) if any of the following are true:

- T_1 and T_2 are identical types and neither is a file type or a structured type that contains a file-type component at any level of structuring.
- T_1 and T_2 are compatible ordinal types, and the values of type T_2 falls within the range of possible values of T_1 .
- T_1 and T_2 are real types, and the value of type T_2 falls within the range of possible values of T_1.
- T_1 is a real type, and T_2 is an integer type.
- T_1 and T_2 are string types.
- T_1 is a string type, and T_2 is a *Char* type.
- T_1 is a string type, and T_2 is a packed string type.
- T_1 and T_2 are compatible, packed string types.
- T_1 and T_2 are compatible set types, and all the members of the value of type T_2 fall within the range of possible values of T_1 .
- T_1 and T_2 are compatible pointer types.
- T_1 is a class type and T_2 is a class type derived from T_1.
- T_1 is a class reference type and T_2 is a class reference type derived from T_1.
- T_1 is a *PChar* and T_2 is a string constant. (This applies only when extended syntax is enabled {**$X+**}.)

- T_1 is a *PChar* and T_2 is a zero-based character array of the form **array**[0..X] **of** *Char*. (This applies only when extended syntax is enabled {**$X+**}.)
- T_1 and T_2 are compatible procedural types.
- T_1 is a procedural type, and T_2 is a procedure or function with an identical result type, an identical number of parameters, and a one-to-one identity between parameter types.

A compile-time error occurs when assignment compatibility is necessary and none of the items in the preceding list are true.

The type declaration part

Programs, procedures, functions, and methods that declare types have a *type declaration part*. This is an example of a type declaration part:

```
type
  TRange = Integer;
  TNumber = Integer;
  TColor = (Red, Green, Blue);
  TCharVal = Ord('A')..Ord('Z');
  TTestIndex = 1..100;
  TTestValue = -99..99;
  TTestList = array[TTestIndex] of TTestValue;
  PTestList = ^TTestList;
  TDate = class
    Year: Integer;
    Month: 1..12;
    Day: 1..31;
    procedure SetDate(D, M, Y: Integer);
    function ShowDate: String;
  end;
  TMeasureData = record
    When: TDate;
    Count: TTestIndex;
    Data: PTestList;
  end;
  TMeasureList = array[1..50] of TMeasureData;
  TName = string[80];
  TSex = (Male, Female);
  PPersonData = ^TPersonData;
  TPersonData = record
    Name, FirstName: TName;
    Age: Integer;
    Married: Boolean;
    TFather, TChild, TSibling: PPersonData;
    case S: TSex of
      Male: (Bearded: Boolean);
      Female: (Pregnant: Boolean);
  end;
```

```
    TPersonBuf = array[0..SizeOf(TPersonData)-1] of Byte;
    TPeople = file of TPersonData;
```

In the example, *TRange*, *TNumber*, and *Integer* are identical types. *TTestIndex* is compatible and assignment-compatible with, but not identical to, the types *TNumber*, *TRange*, and *Integer*. Notice the use of constant expressions in the declarations of *TCharVal* and *TPersonBuf*.

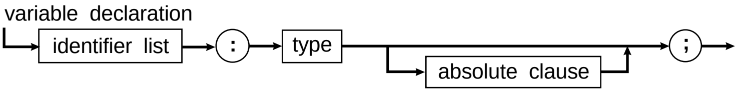
Chapter

4

Variables and typed constants

Variable declarations

A variable is an identifier that marks a value that can change. A *variable declaration* embodies a list of identifiers that designate new variables and their type.



The type given for the variable(s) can be a type identifier previously declared in a **type** declaration part in the same block, in an enclosing block, or in a unit; it can also be a new type definition.

When an identifier is specified within the identifier list of a variable declaration, that identifier is a variable identifier for the block in which the declaration occurs. The variable can then be referred to throughout the block, unless the identifier is redeclared in an enclosed block. Redeclaration creates a new variable using the same identifier, without affecting the value of the original variable.

An example of a variable declaration part follows:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  C: Color;
  Done, Error: Boolean;
  Operator: (Plus, Minus, Times);
  Hue1, Hue2: set of Color;
  Today: Date;
  Results: MeasureList;
  P1, P2: Person;
```

```
Matrix: array[1..10, 1..10] of Double;
```

Variables declared outside procedures and functions are called *global variables* and they reside in the *data segment*. Variables declared within procedures and functions are called *local variables* and they reside in the *stack segment*.

The data segment

The maximum size of the data segment is 65,520 bytes. When a program is linked (this happens automatically at the end of the compilation of a program), the global variables of all units used by the program, as well as the program's own global variables, are placed in the data segment.

If you need more than 65,520 bytes of global data, you should allocate the larger structures as dynamic variables. For more details on this subject, see "Pointers and dynamic variables" on page 33.

The stack segment

The size of the stack segment is set through a **$M** compiler directive—it can be anywhere from 1,024 to 65,520 bytes. The default stack-segment size is 16,384 bytes for a Windows application.

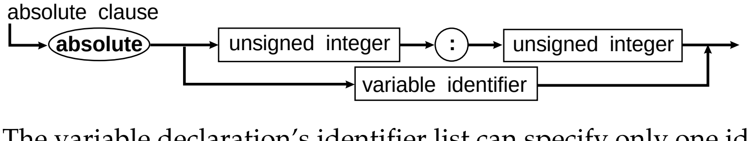
Windows places special demands on the data and stack segments of your program, so the working maximum stack and the data-segment space can be less than the maximum data and stack segment space mentioned here.

Each time a procedure or function is activated (called), it allocates a set of local variables on the stack. On exit, the local variables are disposed of. At any time during the execution of a program, the total size of the local variables allocated by the active procedures and functions can't exceed the size of the stack segment.

The **$S** compiler directive is used to include stack-overflow checks in the code. In the default {**$S+**} state, code is generated to check for stack overflow at the beginning of each procedure and function. In the {**$S-**} state, no such checks are performed. A stack overflow can cause a system crash, so don't turn off stack checks unless you're absolutely sure that an overflow will never occur.

Absolute variables

Variables can be declared to reside at specific memory addresses, and are then called *absolute variables*. The declaration of such variables must include an **absolute** clause following the type:



Note The variable declaration's identifier list can specify only one identifier when an **absolute** clause is present.

The first form of the **absolute** clause specifies the segment and offset at which the variable is to reside:

```
CrtMode : Byte absolute $0040:$0049;
```

The first constant specifies the segment base, and the second specifies the offset within that segment. Both constants must be within the range $0000 to $FFFF (0 to 65,535).

Note Use the first form of the **absolute** clause very carefully, if at all. While a program is running in protected mode, it might not have access rights to memory areas outside your program. Attempting to access these memory areas will likely crash your program.

The second form of the **absolute** clause is used to declare a variable "on top" of another variable, meaning it declares a variable that resides at the same memory address as another variable:

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

This declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and because the first byte of a string variable contains the dynamic length of the string, *StrLen* contains the length of *Str*.

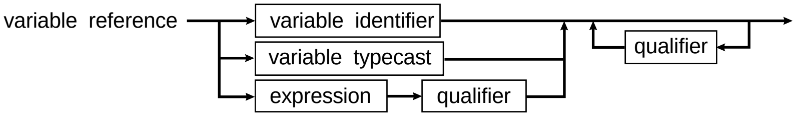
This second form of the **absolute** clause is safe to use in Windows programming. Memory you are accessing is within your program's domain.

Variable references

A variable reference signifies one of the following:

- A variable
- A component of a structured- or string-type variable
- A dynamic variable pointed to by a pointer-type variable

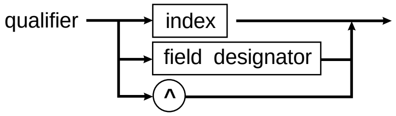
This is the syntax of a variable reference:



Note The syntax for a variable reference allows an expression that computes a pointer type value. The expression must be followed by a qualifier that dereferences the pointer value (or indexes the pointer value if the extended syntax is enabled with the {**$X+**} directive) to produce an actual variable reference.

Qualifiers

A variable reference can contain zero or more qualifiers that modify the meaning of the variable reference.



An array identifier with no qualifier, for example, references the entire array:

```
Results
```

An array identifier followed by an index denotes a specific component of the array—in this case, a structured variable:

```
Results[Current + 1]
```

With a component that is a record or object, the index can be followed by a field designator. Here the variable access signifies a specific field within a specific array component:

```
Results[Current + 1].Data
```

The field designator in a pointer field can be followed by the pointer symbol (^) to differentiate between the pointer field and the dynamic variable it points to:

```
Results[Current + 1].Data^
```

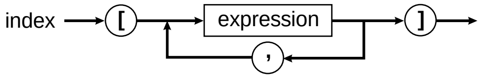
If the variable being pointed to is an array, indexes can be added to denote components of this array:

```
Results[Current + 1].Data^[J]
```

Arrays, strings, and indexes

A specific component of an array variable is denoted by a variable reference that refers to the array variable, followed by an index that specifies the component.

A specific character within a string variable is denoted by a variable reference that refers to the string variable, followed by an index that specifies the character position.



The index expressions select components in each corresponding dimension of the array. The number of expressions can't exceed the number of index types in the array declaration. Also, each expression's type must be assignment-compatible with the corresponding index type.

When indexing a multidimensional array, multiple indexes or multiple expressions within an index can be used interchangeably. For example,

```
Matrix[I][J]
```

is the same as

```
Matrix[I, J]
```

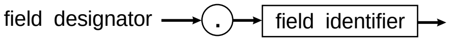
You can index a string variable with a single index expression, whose value must be in the range 0..*N*, where *N* is the declared size of the string. This accesses one character of the string value, with the type *Char* given to that character value.

The first character of a string variable (at index 0) contains the dynamic length of the string; that is, *Length(S)* is the same as *Ord(S[0]*). If a value is assigned to the length attribute, the compiler doesn't check whether this value is less than the declared size of the string. It's possible to index a string beyond its current dynamic length. The characters read are random and assignments beyond the current length don't affect the actual value of the string variable.

When the extended syntax is enabled (using the {**$X+**} compiler directive), a value of type *PChar* can be indexed with a single index expression of type *Word*. The index expression specifies an *offset* to add to the character pointer before it's dereferenced to produce a *Char* type variable reference.

Records and field designators

A specific field of a record variable is denoted by a variable reference that refers to the record variable, followed by a field designator specifying the field.



These are examples of a field designator:

```
Today.Year
Results[1].Count
Results[1].When.Month
```

In a statement within a **with** statement, a field designator doesn't have to be preceded by a variable reference to its containing record.

Object component designators

The format of an object component designator is the same as that of a record field designator; that is, it consists of an instance (a variable reference), followed by a period and a component identifier. A component designator that designates a method is called a *method designator*. A **with** statement can be applied to an instance of a class type. In that case, the instance and the period can be omitted in referencing components of the class type.

The instance and the period can also be omitted within any method block, and when they are, the effect is the same as if *Self* and a period were written before the component reference.

Pointers and dynamic variables

The value of a pointer variable is either **nil** or the address of a dynamic variable.

The dynamic variable pointed to by a pointer variable is referenced by writing the pointer symbol (^) after the pointer variable.

You create dynamic variables and their pointer values with the procedures *New* and *GetMem*. You can use the @ (address-of) operator and the function *Ptr* to create pointer values that are treated as pointers to dynamic variables.

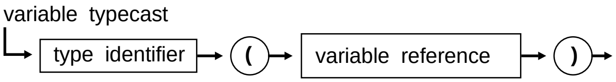
nil doesn’t point to any variable. The results are undefined if you access a dynamic variable when the pointer’s value is **nil** or undefined.

These are examples of references to dynamic variables:

```
P1^
P1^.Sibling^
Results[1].Data^
```

Variable typecasts

Variable typecasting changes the variable reference of one type into a variable reference of another type. The programmer is responsible for determining the validity of a typecast.



When a variable typecast is applied to a variable reference, the variable reference is treated as an instance of the type specified by the type identifier. The size of the variable must be the same as the size of the type denoted by the type identifier.

A variable typecast can be followed by one or more qualifiers, as allowed by the specific type.

Some examples of variable typecasts follow:

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
  TWordRec = record
    Low, High: Word;
  end;
  TPtrRec  = record
    Ofs, Seg: 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)^;
  P := Ptr($40,$49);
  W := TPtrRec(P).Seg;
  Inc(TPtrRec(P).Ofs, 4);
end.
```

Notice the use of the *TByteRec* type to access the low- and high-order bytes of a word. This corresponds to the built-in functions *Lo* and *Hi*, except that a variable typecast can also be used on the left side of an assignment. Also, observe the use of the *TWordRec* and *TPtrRec* types to access the low- and high-order words of a long integer and the offset and segment parts of a pointer.

Object Pascal fully supports variable typecasts involving procedural types. For example, given the declarations

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

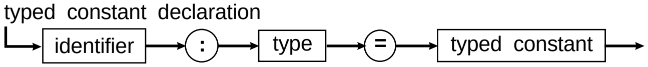
you can construct the following assignments:

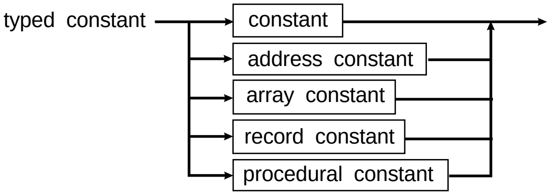
```
F := Func(P);       { Assign procedural value in P to F }
Func(P) := F;       { Assign procedural value in F to P }
@F := P;    { Assign pointer value in P to F }
P := @F;    { Assign pointer value in F to P }
N := F(N);  { Call function via F }
N := Func(P)(N);    { Call function via P }
```

In particular, notice that the address operator (@), when applied to a procedural variable, can be used on the left side of an assignment. Also, notice the typecast on the last line to call a function via a pointer variable.

Typed constants

Typed constants can be compared to initialized variables—variables whose values are defined on entry to their block. Unlike an untyped constant, the declaration of a typed constant specifies both the type and the value of the constant.





Typed constants can be used exactly like variables of the same type and they can appear on the left-hand side in an assignment statement. Note that typed constants are initialized *only once*—at the beginning of a program. Therefore, for each entry to a procedure or function, the locally declared typed constants aren't reinitialized.

In addition to a normal constant expression, the value of a typed constant can be specified using a *constant-address expression.* A constant-address expression is an expression that involves taking the address, offset, or segment of a global variable, a typed constant, a procedure, or a function. Constant-address expressions can't reference local variables (stack-based) or dynamic (heap-based) variables, because their addresses can't be computed at compile time.

Simple-type constants

Declaring a typed constant as a simple type specifies the value of the constant:

```
const
  Maximum: Integer = 9999;
  Factor: Real = -0.1;
  Breakchar: Char = #3;
```

As mentioned earlier, the value of a typed constant can be specified using a constant-address expression, that is, an expression that takes the address, offset, or segment of a global variable, a typed constant, a procedure, or a function. For example,

```
var
  Buffer: array[0..1023] of Byte;
const
  BufferOfs: Word = Ofs(Buffer);
  BufferSeg: Word = Seg(Buffer);
```

Because a typed constant is actually a variable with a constant value, it can't be interchanged with ordinary constants. For example, it can't be used in the declaration of other constants or types:

```
const
  Min: Integer = 0;
  Max: Integer = 99;
type
  Vector = array[Min..Max] of Integer;
```

The *Vector* declaration is invalid because *Min* and *Max* are typed constants.

String-type constants

The declaration of a typed constant of a string type specifies the maximum length of the string and its initial value:

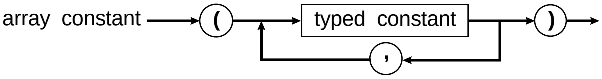
```
const
  Heading: string[7] = 'Section';
  NewLine: string[2] = #13#10;
  TrueStr: string[5] = 'Yes';
  FalseStr: string[5] = 'No';
```

Structured-type constants

The declaration of a structured-type constant specifies the value of each of the structure's components. Object Pascal supports the declaration of array, record, and set type constants. File type constants and constants of array, and record types that contain file type components aren't allowed.

Array-type constants

The declaration of an array-type constant, enclosed in parentheses and separated by commas, specifies the values of the components.



This is an example of an array-type constant:

```
type
  TStatus = (Active, Passive, Waiting);
  TStatusMap = array[TStatus] of string[7];
const
  StatStr: TStatusMap = ('Active', 'Passive', 'Waiting');
```

This example defines the array constant *StatStr*, which can be used to convert values of type *TStatus* into their corresponding string representations. These are the components of *StatStr*:

```
StatStr[Active] = 'Active'
StatStr[Passive] = 'Passive'
StatStr[Waiting] = 'Waiting'
```

The component type of an array constant can be any type except a file type. Packed string-type constants (character arrays) can be specified both as single characters and as strings. The definition

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

can be expressed more conveniently as

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

When the extended syntax is enabled (using a {**$X+**} compiler directive), a zero-based character array can be initialized with a string that is shorter than the declared length of the array. For example,

```
  const
    FileName = array[0..79] of Char = 'TEST.PAS';
```

In such cases, the remaining characters are set to NULL (#0) and the array effectively contains a null-terminated string. For more about null-terminated strings, see Chapter15.

Multidimensional-array constants are defined by enclosing the constants of each dimension in separate sets of parentheses, separated by commas. The innermost constants correspond to the rightmost dimensions. The declaration

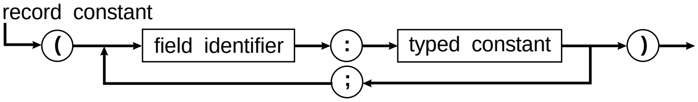
```
type
  TCube = array[0..1, 0..1, 0..1] of Integer;
const
  Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));
```

provides an initialized array *Maze* with the following values:

```
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
```

Record-type constants

The declaration of a record-type constant specifies the identifier and value of each field, enclosed in parentheses and separated by semicolons.



Some examples of record constants follow:

```
type
  TPoint = record
    X, Y: Real;
  end;
  TVector = array[0..1] of Point;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
    Nov, Dec);
  TDate = record
    D: 1..31;
    M: Month;
    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);
```

The fields must be specified in the same order as they appear in the definition of the record type. If a record contains fields of file types, the constants of that record type can't be declared. If a record contains a variant, only fields of the selected variant can be specified. If the variant contains a tag field, then its value must be specified.

Set-type constants

Just like a simple-type constant, the declaration of a set-type constant specifies the value of the set using a constant expression. Here are some examples:

```
type
  TDigits = set of 0..9;
  TLetters = set of 'A'..'Z';
const
  EvenDigits: TDigits = [0, 2, 4, 6, 8];
  Vowels: TLetters = ['A', 'E', 'I', 'O', 'U', 'Y'];
  HexDigits: set of '0'..'z' = ['0'..'9', 'A'..'F', 'a'...f'];
```

Pointer-type constants

The declaration of a pointer-type constant uses a constant-address expression to specify the pointer value. Some examples follow:

```
type
  TDirection = (Left, Right, Up, Down);
  TStringPtr = ^String;
  TNodePtr = ^Node;
  TNode = record
    Next: TNodePtr;
    Symbol: TStringPtr;
    Value: TDirection;
  end;

const
  S1: string[4] = 'DOWN';
  S2: string[2] = 'UP';
  S3: string[5] = 'RIGHT';
  S4: string[4] = 'LEFT';
  N1: TNode = (Next: nil; Symbol: @S1; Value: Down);
  N2: TNode = (Next: @N1; Symbol: @S2; Value: Up);
  N3: TNode = (Next: @N2; Symbol: @S3; Value: Right);
  N4: TNode = (Next: @N3; Symbol: @S4; Value: Left);
  DirectionTable: TNodePtr = @N4;
```

When the extended syntax is enabled (using a {**$X+**} compiler directive), a typed constant of type *PChar* can be initialized with a string constant. For example,

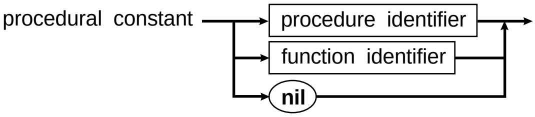
```
  const
```

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

The result is that the pointer now points to an area of memory that contains a zero-terminated copy of the string literal. See Chapter15, "Using null-terminated strings," for more information.

Procedural-type constants

A procedural-type constant must specify the identifier of a procedure or function that is assignment-compatible with the type of the constant, or it must specify the value **nil**.



Here's an example:

```
type
  TErrorProc = procedure(ErrorCode: Integer);

procedure DefaultError(ErrorCode: Integer); far;
begin
  Writeln('Error ', ErrorCode, '.');
end;

const
  ErrorHandler: TErrorProc = DefaultError;
```

Chapter

5

Expressions

Expressions are made up of *operators* and *operands*. Most Object Pascal operators are *binary*; they take two operands. The rest are *unary* and take only one operand. Binary operators use the usual algebraic form (for example, *A* + *B*). A unary operator always precedes its operand (for example, -*B*).

In more complex expressions, rules of precedence clarify the order in which operations are performed.

Table 5-1 Precedence of operators

Operators	Precedence	Categories
@, not	first (high)	unary operators
*, /, **div, mod, and, shl, shr, as**	second	multiplying operators
+,-, or, xor	third	adding operators
=, <>, <, >, <=, >=, in, is	fourth (low)	relational operators

There are three basic rules of precedence:

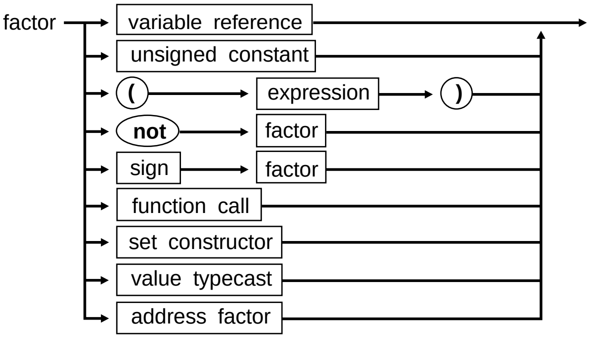
- An operand between two operators of different precedence is bound to the operator with higher precedence.
- An operand between two equal operators is bound to the one on its left.
- Expressions within parentheses are evaluated prior to being treated as a single operand

Operations with equal precedence are normally performed from left to right, although the compiler may rearrange the operands to generate optimum code.

Expression syntax

The precedence rules follow from the syntax of expressions, which are built from factors, terms, and simple expressions.

A factor’s syntax follows:



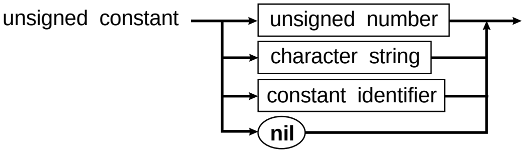
A function call activates a function and denotes the value returned by the function. See “Function calls” on page 50.

A set constructor denotes a value of a set type. See “Set constructors” on page 50.

A value typecast changes the type of a value. See “Value typecasts” on page 51.

An address factor computes the address of a variable, procedure, function, or method. See “The @ operator” on page 49.

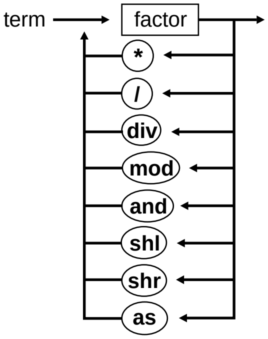
An unsigned constant has the following syntax:



These are some examples of factors:

```
X  { Variable reference }
@X                         { Pointer to a variable }
15                         { Unsigned constant }
(X + Y + Z)                { Subexpression }
Sin(X / 2)                 { Function call }
exit['0'..'9', 'A'..'Z']   { Set constructor }
not Done                   { Negation of a Boolean }
Char(Digit + 48)           { Value typecast }
```

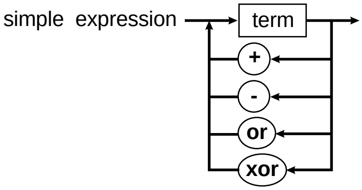
Terms apply the multiplying operators to factors:



Here are some examples of terms:

```
X * Y
Z / (1 - Z)
Y shl 2
(X <= Y) and (Y < Z)
```

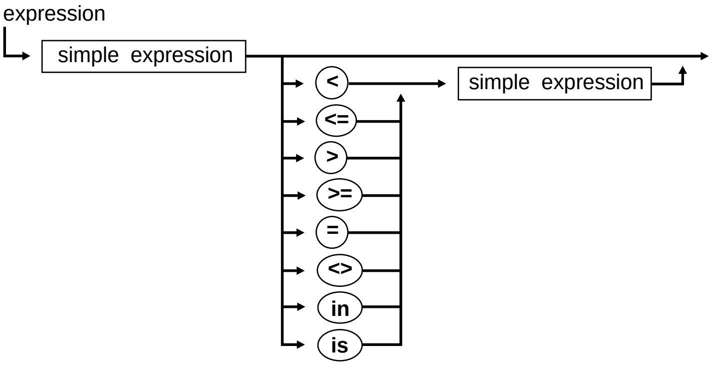
Simple expressions apply adding operators and signs to terms:



Here are some examples of simple expressions:

```
X + Y
-X
Hue1 + Hue2
I * J + 1
```

An expression applies the relational operators to simple expressions:



Here are some examples of expressions:

```
X = 1.5
```

```
Done <> Error
(I < J) = (J < K)
C in Hue1
```

Operators

Operators are classified as arithmetic operators, logical operators, string operators, character-pointer operators, set operators, relational operators, and the @ operator.

Arithmetic operators

The following tables show the types of operands and results for binary and unary arithmetic operations.

Table 5-2 Binary arithmetic operations

Operator	Operation	Operand types	Result type
+	addition	integer type	integer type
		real type	real type
-	subtraction	integer type	integer type
		real type	real type
*	multiplication	integer type	integer type
		real type	real type
/	division	integer type	real type
		real type	real type
div	integer division	integer type	integer type
mod	remainder	integer type	integer type

The + operator is also used as a string or set operator, and the +, -, and * operators are also used as set operators.

Table 5-3 Unary arithmetic operations

Operator	Operation	Operand types	Result type
+	sign identity	integer type	integer type
		real type	real type
-	sign negation	integer type	integer type
		real type	real type

Any operand whose type is a subrange of an ordinal type is treated as if it were of the ordinal type.

If both operands of a **+, -**,*, **div, or mod** operator are of an integer type, the result type is of the common type of the two operands. For a definition of common types, see page 12.

If one or both operands of a **+**, **-, or** * operator are of a real type, the type of the result is *Real* in the {**$N-**} state or *Extended* in the {**$N+**} state.

If the operand of the sign identity or sign negation operator is of an integer type, the result is of the same integer type. If the operator is of a real type, the type of the result is *Real* or *Extended*.

The value of *X* **/** *Y* is always of type *Real* or *Extended* regardless of the operand types. A run-time error occurs if *Y* is zero.

The value of *I* **div** *J* is the mathematical quotient of *I* / *J*, rounded in the direction of zero to an integer-type value. A run-time error occurs if *J* is zero.

The **mod** operator returns the remainder obtained by dividing its two operands; that is,

```
I mod J = I - (I div J) * J
```

The sign of the result of **mod** is the same as the sign of *I*. A run-time error occurs if *J* is zero.

Logical operators

The types of operands and results for logical operations are shown in the following table.

Table 5-4 Logical operations

Operator	**Operation**	**Operand types**	**Result type**
not	bitwise negation	integer type	*Boolean*
and	bitwise and	integer type	*Boolean*
or	bitwise or	integer type	*Boolean*
xor	bitwise xor	integer type	*Boolean*
shl	Operation	integer type	*Boolean*
shr	Operation	integer type	*Boolean*

If the operand of the **not** operator is of an integer type, the result is of the same integer type. The **not** operator is a unary operator.

If both operands of an **and**, **or, or xor** operator are of an integer type, the result type is the common type of the two operands.

The operations *I shl J* and Ishr *J* shift the value of *I* to the left right by *J* bits. The result type is the same as the type of *I.*

Boolean operators

The types of operands and results for Boolean operations are shown in the following table.

Table 5-5 Boolean operations

Operator	**Operation**	**Operand types**	**Result type**
not	negation	Boolean type	*Boolean*
and	logical and	Boolean type	*Boolean*
or	logical or	Boolean type	*Boolean*
xor	logical xor	Boolean type	*Boolean*

Normal Boolean logic governs the results of these operations. For instance, *A* **and** *B* **is** *True* only if both *A* and *B* are *True*.

Object Pascal supports two different models of code generation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation.

Complete evaluation means that every operand of a Boolean expression built from the **and** and **or** operators is guaranteed to be evaluated, even when the result of the entire expression is already known. This model is convenient when one or more operands of an expression are functions with side effects that alter the meaning of the program.

Short-circuit evaluation guarantees strict left-to-right evaluation and that evaluation stops as soon as the result of the entire expression becomes evident. This model is convenient in most cases because it guarantees minimum execution time, and usually minimum code size. Short-circuit evaluation also makes possible the evaluation of constructs that would not otherwise be legal. For example,

```
while (I <= Length(S)) and (S[I] <> ' ') do
  Inc(I);
while (P <> nil) and (P^.Value <> 5) do
  P := P^.Next;
```

In both cases, the second test isn't evaluated if the first test is *False*.

The evaluation model is controlled through the **$B** compiler directive. The default state is {**$B-**}, and in this state, the compiler generates short-circuit evaluation code. In the {**$B+**} state, the compiler generates complete evaluation.

Because Standard Pascal doesn't specify which model should be used for Boolean expression evaluation, programs dependent on either model aren't truly portable. You may decide, however, that sacrificing portability is worth the gain in execution speed and simplicity provided by the short-circuit model.

String operator

The types of operands and results for string operation are shown in the following table.

Table 5-6 String operation

Operator	Operation	Operand types	Result type
+	concatenation	string type, *Char* type, or packed string type	string type

Object Pascal allows the **+** operator to be used to concatenate two string operands. The result of the operation *S* **+** *T***,** where S and T are of a string type, a *Char* type, or a packed string type, is the concatenation of *S* and *T*. The result is compatible with any string type (but not with *Char* types and packed string types). If the resulting string is longer than 255 characters, it's truncated after character 255.

Character-pointer operators

The extended syntax (enabled using a **{$X+}** compiler directive) supports a number of character-pointer operations. The plus (+) and minus (-) operators can be used to increment and decrement the offset part of a pointer value, and the minus operator can be used to calculate the distance (difference) between the offset parts of two character pointers. Assuming that P and *Q* are values of type *PChar* and I is a value of type Word, these constructs are allowed:

Table 5-7 Permitted PChar constructs

Operation	Result
P + *I*	Add *I* to the offset part of *P*
I + *P*	Add *I* to the offset part of *P*
P **-** *I*	Subtract *I* from the offset part of *P*
P **-** *Q*	Subtract offset part of *Q* from offset part of *P*

The operations *P* **+** *I* and *I* + *P* adds I to the address given by P, producing a pointer that points I characters after P. The operation P **-** *I* subtracts *I* from the address given by *P*, producing a pointer that points *I* characters before *P*.

The operation *P*- *Q* computes the distance between *Q (the lower address) and P* (the higher address), resulting in a value of type *Word* that gives the number of characters between *Q* and *P*. This operation assumes that *P* and *Q* point within the same character array. If the two character pointers point into different character arrays, the result is undefined.

Set operators

The types of operands for set operations are shown in the following table.

Table 5-8 Set operations

Operator	Operation	Operand types
+	union	compatible set types
-	difference	compatible set types
*	intersection	compatible set types

The results of set operations conform to the rules of set logic:

- An ordinal value *C* is in *A* **+** *B* only if *C* is in *A* or *B*.
- An ordinal value *C* is in *A* **-** *B* only if *C* is in *A* and not in *B*
- An ordinal value *C* is in *A* * *B* only if *C* is in both *A* and *B*.

If the smallest ordinal value that is a member of the result of a set operation is *A* and the largest is *B*, then the type of the result is **set of** *A*..*B.*

Relational operators

The types of operands and results for relational operations are shown in the following table.

Table 5-9 Relational operations

Operator type	Operation	Operand types	Result type
=	equal	compatible simple, class, class reference, pointer, set, string, or packed string	*Boolean*

		types	
<>	not equal	compatible simple, class, class reference, pointer, set, string, or packed string types	*Boolean*
<	less than	compatible simple, string, packed string types, or *PChar*	*Boolean*
>	greater than	compatible simple, string, packed string types, or *PChar*	*Boolean*
<=	less than or equal to	compatible simple, string, packed string types, or *PChar*	*Boolean*
>=	greater than or equal to	compatible simple, string, or packed string types, or *PChar*	*Boolean*
<=	subset of	compatible set types	*Boolean*
>=	superset of	compatible set types	*Boolean*
in	member of	left operand, any ordinal type *T;* right operand, set whose base is compatible with *T*	*Boolean*

Comparing simple types

When the operands **=, <>**, **<**, **>**, **>=**, or **<=** are of simple types, they must be compatible types; however, if one operand is of a real type, the other can be of an integer type.

Comparing strings

The relational operators **=**, **<>**, **<**, **>>**, **>=,** and **<=** compare strings according to the ordering of the extended ASCII character set. Any two string values can be compared because all string values are compatible.

A character-type value is compatible with a string-type value. When the two are compared, the character-type value is treated as a string-type value with length 1. When a packed string-type value with *N* components is compared with a string-type value, it's treated as a string-type value with length *N*.

Comparing packed strings

The relational operators =, <>, <, >, >=, and <= can also be used to compare two packed string-type values if both have the same number of components. If the number of components is N, then the operation corresponds to comparing two strings, each of length N.

Comparing pointers and references

The operators **=** and **<>** can be used on compatible pointer-type, class-type, class-reference-type operands. Two pointers are equal only if they point to the same object.

Comparing character pointers

The extended syntax (enabled using a {**$X+**} compiler directive) allows the **>, <**, **>=**, and **<=** operators to be applied to *PChar* values. Note, however, that these relational tests assume that the two pointers being compared point within the same character array, and for that reason, the operators only compare the offset parts of the two pointer values. If the two character pointers point into different character arrays, the result is undefined.

Comparing sets

If *A* and *B* are set operands, their comparisons produce these results**:**

- *A* **=** *B* is *True* only if *A* and *B* contain exactly the same members; otherwise, *A* **<>** *B*.
- *A* **<=** *B* is *True* only if every member of *A* is also a member of *B*.
- *A* **>=** *B* is *True* only if every member of *B* is also a member of *A*.

Testing set membership

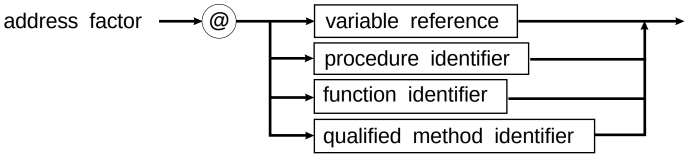
The **in** operator returns *True* when the value of the ordinal-type operand is a member of the set-type operand; otherwise, it returns *False*.

Class operators

Object Pascal defines two operators, **is** and **as**, that operate on class and object references. See Chapter 9, "Class types."

The @ operator

The @ operator is used in an address factor to compute the address of a variable, procedure, function, or method.



The @ operator returns the address of its operand, that is, it constructs a pointer value that points to the operand.

@ with a variable

When applied to a variable reference, @ returns a pointer to the variable. The type of the resulting pointer value is controlled through the **$T** compiler directive: In the {**$T-**} state (the default), the result type is *Pointer*. In other words, the result is an untyped pointer, which is compatible with all other pointer types. In the {**$T+**} state, the type of the result is ^*T,* where *T* is the type of the variable reference. In other words, the result is of a type that is compatible only with other pointers to the type of the variable.

Special rules apply to use of the @ operator with a procedural variable. For more details, see "Procedural types in expressions" on page 52.

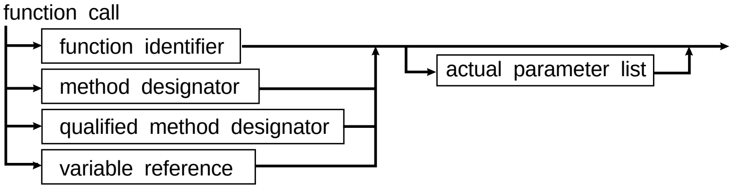
@ with a procedure, function, or method

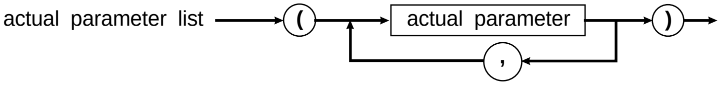
You can apply @ to a procedure, function, or method to produce a pointer to the routine's entry point. The type of the resulting pointer is always *Pointer*, regardless of the state of the **$T** compiler directive. In other words, the result is always an untyped pointer, which is compatible with all other pointer types.

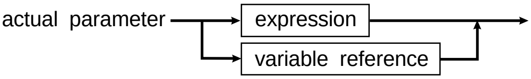
When @ is applied to a method, the method must be specified through a *qualified-method identifier* (a class-type identifier, followed by a period, followed by a method identifier).

Function calls

A function call activates a function specified by a function identifier, a method designator, a qualified-method designator, or a procedural-type variable reference. The function call must have a list of actual parameters if the corresponding function declaration contains a list of formal parameters. Each parameter takes the place of the corresponding formal parameter according to parameter rules explained in Chapter 8, "Procedures and functions," on page 75.







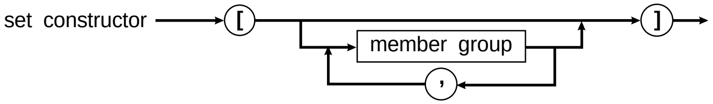
These are some examples of function calls:

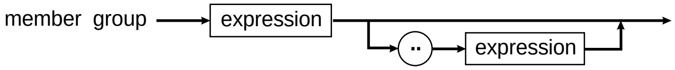
```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
```

In the extended syntax {**$X+**} mode, function calls can be used as statements; that is, the result of a function call can be discarded. See "Method activations" on page 93, and "Procedural types" on page 22.

Set constructors

A set constructor denotes a set-type value, and is formed by writing expressions within brackets ([]). Each expression denotes a value of the set.





The notation **[ ]** denotes the empty set, which is assignment-compatible with every set type. Any member group *X..Y* denotes as set members all values in the range *X..Y*. If *X* is greater than *Y*, then *X..Y* doesn't denote any members and [*X..Y]* denotes the empty set.

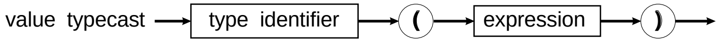
All expression values in member groups in a particular set constructor must be of the same ordinal type.

These are some examples of set constructors:

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

Value typecasts

The type of an expression can be changed to another type through a value typecast.



The expression type and the specified type must both be either ordinal types or pointer types. For ordinal types, the resulting value is obtained by converting the expression. The conversion may involve truncation or extension of the original value if the size of the specified type is different from that of the expression. In cases where the value is extended, the sign of the value is always preserved; that is, the value is sign-extended.

The syntax of a value typecast is almost identical to that of a variable typecast. Value typecasts operate on values, however, not on variables, and therefore they can't participate in variable references; that is, a value typecast can't be followed by qualifiers. In particular, value typecasts can't appear on the left side of an assignment statement. See "Variable typecasts" on page 34.

These are some examples of value typecasts:

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
BytePtr(Ptr($40, $49))
```

Procedural types in expressions

Usually, using a procedural variable in a statement or an expression calls the procedure or function stored in the variable. There is one exception: When the compiler sees a procedural variable on the left side of an assignment statement, it knows that the right side has to represent a procedural value. For example, consider the following program:

```
type
  IntFunc = function: Integer;

var
  F: IntFunc;
  N: Integer;

function ReadInt: Integer; far;
var
  I: Integer;
begin
  Read(I);
  ReadInt := I;
end;

begin
  F := ReadInt; { Assign procedural value }
  N := ReadInt; { Assign function result }
end.
```

The first statement in the main program assigns the procedural value (address of) ReadInt to the procedural variable *F*, where the second statement calls *ReadInt* and assigns the returned value to *N*. The distinction between getting the procedural value or calling the function is made by the type of the variable being assigned (*F* or *N*).

Unfortunately, there are situations where the compiler can't determine the desired action from the context. For example, in the following statement there is no obvious way the compiler can know if it should compare the procedural value in *F* to the procedural value of *ReadInt* to determine if *F* currently points to *ReadInt*, or if it should call *F* and *ReadInt* and then compare the returned values.

```
if F = ReadInt then
  Edit1.Text := 'Equal';
```

Object Pascal syntax, however, specifies that the occurrence of a function identifier in an expression denotes a call to that function, so the effect of the preceding statement is to call *F* and *ReadInt*, and then compare the returned values. To compare the procedural value in *F* to the procedural value of *ReadInt*, the following construct must be used:

```
if @F = @ReadInt then
  Edit1.Text := 'Equal';
```

When applied to a procedural variable or a procedure or function identifier, the address *(@)* operator prevents the compiler from calling the procedure, and at the same time converts the argument into a pointer. *@F* converts *F* into an untyped pointer variable that contains an address, and *@ReadInt* returns the address of *ReadInt*; the two pointer values can then be compared to determine if *F* currently refers to *ReadInt*.

The **@** operator is often used when assigning an untyped pointer value to a procedural variable. For example, the *GetProcAddress* function defined by Windows (in the *WinProcs* unit) returns the address of an exported function in a DLL as an untyped pointer value. Using the **@** operator, the result of a call to *GetProcAddress* can be assigned to a procedural variable:

```
type
  TStrComp = function(Str1, Str2: Pchar): Integer;
var
  StrComp: TStrComp;
    ⋮
begin
    ⋮
  @StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
    ⋮
end.
```

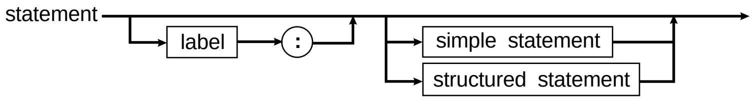
Note To get the memory address of a procedural variable rather than the address stored in it, use a double address (**@@**) operator. For example, where *@P* means convert *P* into an untyped pointer variable, *@@P* means return the physical address of the variable *P*.

Chapter

6

Statements

Statements describe algorithmic actions that can be executed. Labels can prefix statements, and these labels can be referenced by **goto** statements.

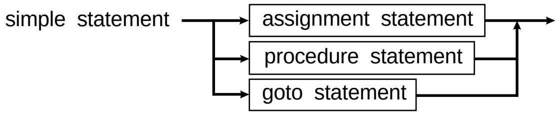


A label is either a digit sequence in the range 0 to 9999 or an identifier.

There are two main types of statements: simple statements and structured statements.

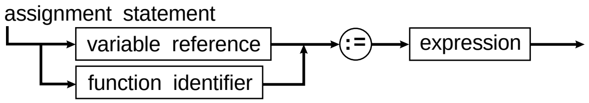
Simple statements

A simple statement is a statement that doesn't contain any other statements.



Assignment statements

Assignment statements replace the current value of a variable with a new value specified by an expression. They can be used to set the return value of the function also.



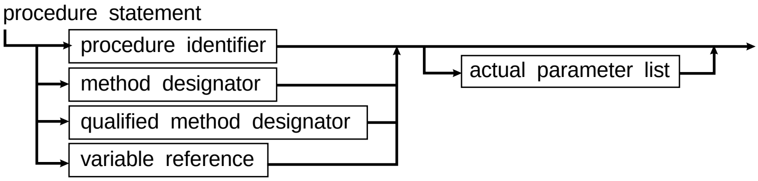
The expression must be assignment-compatible with the type of the variable or the type of the function result. See the section "Type compatibility" on page 25.

Some examples of assignment statements follow:

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
```

Procedure statements

A procedure statement activates a procedure specified by a procedure identifier, a method designator, a qualified-method designator, or a procedural-type variable reference. If the corresponding procedure declaration contains a list of formal parameters, then the procedure statement must have a matching list of (parameters listed in definitions are *formal parameters*; in the calling statement, they are *actual parameters*). The actual parameters are passed to the formal parameters as part of the call. For more information, see Chapter 8, "Procedures and functions."

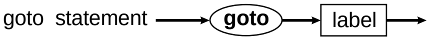


Some examples of procedure statements follow:

```
PrintHeading;
Transpose(A, N, M);
Find(Name, Address);
```

Goto statements

A **goto** statement transfers program execution to the statement marked by the specified label. The syntax diagram of a **goto** statement follows:



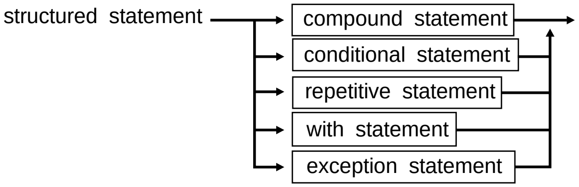
When using **goto** statements, observe the following rules:

- The label referenced by a **goto** statement must be in the same block as the **goto** statement. In other words, it's not possible to jump into or out of a procedure or function.
- Jumping into a structured statement from outside that structured statement (that is, jumping to a deeper level of nesting) can have undefined effects, although the compiler doesn't indicate an error. For example, you shouldn't jump into the middle of a **for** loop.

Good programming practices recommend that you use goto statements as little as possible.

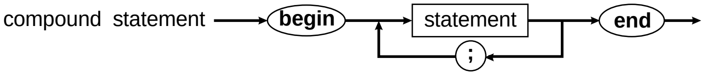
Structured statements

Structured statements are constructs composed of other statements that are to be executed in sequentially (compound and **with** statements), conditionally (conditional statements), or repeatedly (repetitive statements).



Compound statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The component statements are treated as one statement, crucial in contexts where the Object Pascal syntax only allows one statement. **begin** and **end** bracket the statements, which are separated by semicolons.

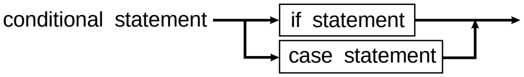


Here's an example of a compound statement:

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

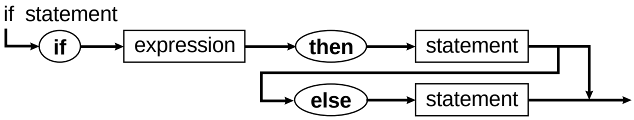
Conditional statements

A conditional statement selects for execution a single one (or none) of its component statements.



If statements

The syntax for an **if** statement reads like this:



The expression must yield a result of the standard type *Boolean*. If the expression produces the value *True*, then the statement following **then** is executed.

If the expression produces *False* and the **else** part is present, the statement following **else** is executed; if the **else** part isn't present, execution continues at the next statement following the **if** statement.

The syntactic ambiguity arising from the construct

```
if e1 then if e2 then s1 else s2;
```

is resolved by interpreting the construct as follows:

Note

No semicolon is allowed preceding an else clause.

```
if e1 then
begin
  if e2 then
    s1
  else
    s2
end;
```

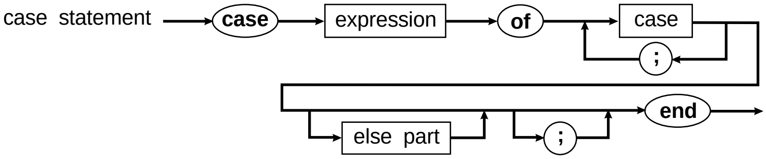
Usually, an **else** is associated with the closest **if** not already associated with an **else**.

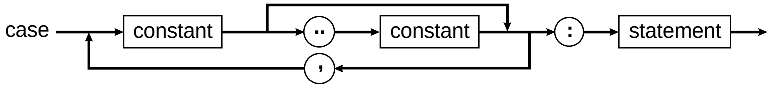
Two examples of **if** statements follow:

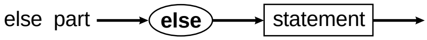
```
if X < 1.5 then
  Z := X + Y
else
  Z := 1.5;

if P1 <> nil then
  P1 := P1^.Father;
```

Case statements

The **case** statement consists of an expression (the selector) and a list of statements, each prefixed with one or more constants (called *case constants*) or with the word **else**. The selector must be of a byte-sized or word-sized ordinal type, so string types and the integer type *Longint* are invalid selector types. All **case** constants must be unique and of an ordinal type compatible with the selector type.







The **case** statement executes the statement prefixed by a **case** constant equal to the value of the selector or a **case** range containing the value of the selector. If no such **case** constant of the **case** range exists and an **else** part is present, the statement following **else** is executed. If there is no **else** part, execution continues with the next statement following the **if** statement.

These are examples of **case** statements:

```
case Operator of
  Plus: X := X + Y;
  Minus: X := X - Y;
  Times: X := X * Y;
end;
```

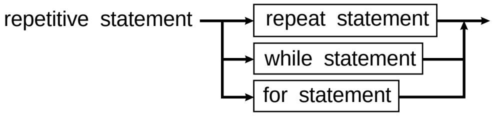
```
case I of
  0, 2, 4, 6, 8: Edit1.Text := 'Even digit';
  1, 3, 5, 7, 9: Edit1.Text := 'Odd digit';
  10..100: Edit1.Text := 'Between 10 and 100';
else
  Edit1.Text := 'Negative or greater than 100';
end;
```

Ranges in case statements must not overlap. So for example, the following case statement is not allowed:

```
case MySelector of
  5: Edit1.Text := 'Special case';
  1..10: Edit1.Text := 'General case';
end;
```

Repetitive statements

Repetitive statements specify certain statements to be executed repeatedly.

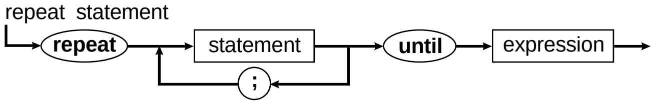


If the number of repetitions is known beforehand, the **for** statement is the appropriate construct. Otherwise, the **while** or **repeat** statement should be used.

The *Break* and *Continue* standard procedures can be used to control the flow of repetitive statements: *Break* terminates a repetitive statement, and *Continue* continues with the next iteration of a repetitive statement. For more details on these standard procedures, see the *Visual Component Library Reference*.

Repeat statements

A **repeat** statement contains an expression that controls the repeated execution of a statement sequence within that **repeat** statement.



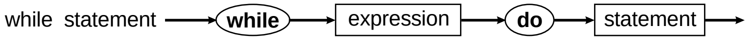
The expression must produce a result of type *Boolean*. The statements between the symbols **repeat** and **until** are executed in sequence until, at the end of a sequence, the expression yields *True*. The sequence is executed at least once because the expression is evaluated *after* the execution of each sequence.

These are examples of **repeat** statements:

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

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

While statements

A **while** statement contains an expression that controls the repeated execution of a statement (which can be a compound statement).



The expression controlling the repetition must be of type *Boolean*. It is evaluated *before* the contained statement is executed. The contained statement is executed repeatedly as long as the expression is *True*. If the expression is *False* at the beginning, the statement isn't executed at all.

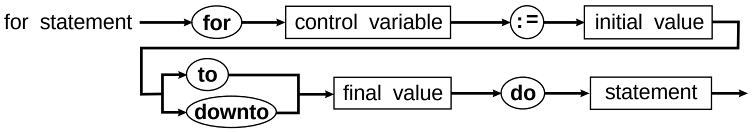
These are examples of **while** statements:

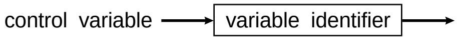
```
while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z  * X;
  I := I div 2;
  X := Sqr(X);
end;

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

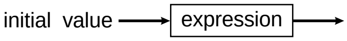
```
end;
```

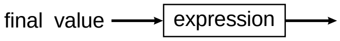
For statements

The **for** statement causes a statement to be repeatedly executed while a progression of values is assigned to a control variable. Such a statement can be a compound statement.









The control variable must be a variable identifier (without any qualifier) that is local in scope to the block containing the **for** statement. The control variable must be of an ordinal type. The initial and final values must be of a type assignment-compatible with the ordinal type. See Chapter 7 for a discussion of locality and scope.

When a **for** statement is entered, the initial and final values are determined once for the remainder of the execution of the **for** statement.

The statement contained by the **for** statement is executed once for every value in the range *initial value* to *final value*. The control variable always starts off at *initial value*. When a **for** statement uses **to**, the value of the control variable is incremented by one for each repetition. If *initial value* is greater than *final value*, the contained statement isn't executed. When a **for** statement uses **downto**, the value of the control variable is decremented by one for each repetition. If *initial value* value is less than *final value*, the contained statement isn't executed.

If the contained statement alters the value of the control variable, your results will probably not be what you expect. After a **for** statement is executed, the value of the control variable value is undefined, unless execution of the **for** statement was interrupted by a **goto** from the **for** statement.

With these restrictions in mind, the **for** statement

```
for V := Expr1 to Expr2 do Body;
```

is equivalent to

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 <= Temp2 then
  begin
    V := Temp1;
    Body;
    while V <> Temp2 do
    begin
      V := Succ(V);
      Body;
    end;
  end;
end;
```

and the **for** statement

```
for V := Expr1 downto Expr2 do Body;
```

is equivalent to

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 >= Temp2 then
  begin
    V := Temp1;
    Body;
    while V <> Temp2 do
    begin
      V := Pred(V);
      Body;
    end;
  end;
end;
```

where *Temp1* and *Temp2* are auxiliary variables of the host type of the variable *V* and don't occur elsewhere in the program.

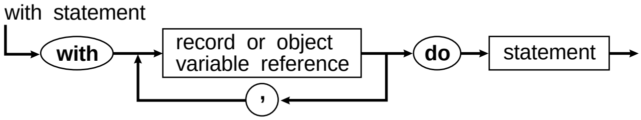
These are examples of **for** statements:

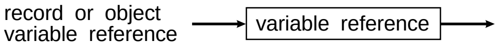
```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I]
```

```
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);
```

With statements

The **with** statement is shorthand for referencing the fields of a record, and the fields and methods of an object. Within a **with** statement, the fields of one or more specific record variables can be referenced using their field identifiers only. The syntax of a **with** statement follows:





Given this type declaration,

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

var OrderDate: TDate;
```

here is an example of a **with** statement:

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

This is equivalent to

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

Within a **with** statement, each variable reference is first checked to see if it can be interpreted as a field of the record. If so, it's always interpreted as such, even if a variable with the same name is also accessible. Suppose the following declarations have been made:

```
type
  TPoint = record
  X, Y: Integer;
  end;
var
  X: TPoint;
  Y: Integer;
```

In this case, both *X* and *Y* can refer to a variable or to a field of the record. In the statement

```
with X do
begin
  X := 10;
  Y := 25;
end;
```

the *X* between **with** and **do** refers to the variable of type *TPoint*, but in the compound statement, *X* and *Y* refer to *X.X* and *X.Y*.

The statement

```
with V1, V2, ... Vn do S;
```

is equivalent to

```
with V1 do
  with V2 do
   ⋮
      with Vn do
        S;
```

In both cases, if *Vn* is a field of both *V1* and *V2*, it's interpreted as *V2.Vn*, not *V1.Vn*.

If the selection of a record variable involves indexing an array or dereferencing a pointer, these actions are executed once before the component statement is executed.

Chapter

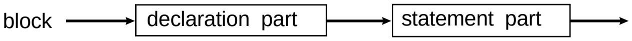
7

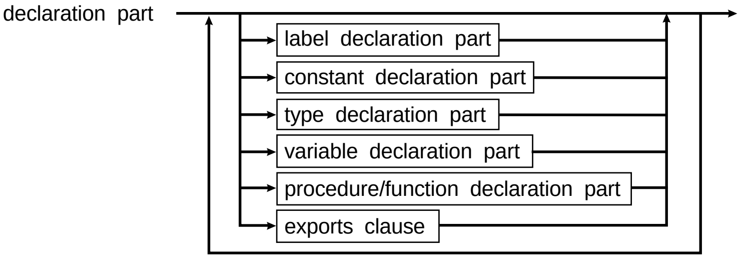
Blocks, locality, and scope

A *block* is made up of declarations, which are written and combined in any order, and statements. Each block is part of a procedure declaration, a function declaration, a method declaration, or a program or unit. All identifiers and labels declared in the declaration part are local to the block.

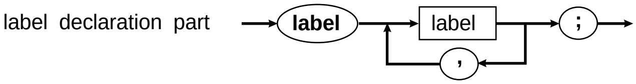
Blocks

The overall syntax of any block follows this format:



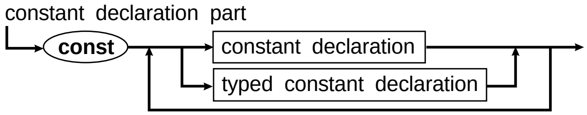

declaration part
label declaration part
constant declaration part
type declaration part
variable declaration part
procedure/function declaration part
exports clause

Labels that mark statements in the corresponding statement part are declared in the *label declaration part*. Each label must mark only one statement.

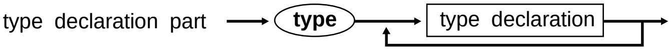


A label must be an identifier or a digit sequence in the range 0 to 9999.

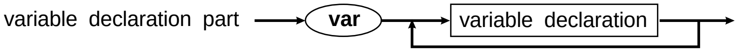
The *constant declaration part* consists of constant declarations local to the block.



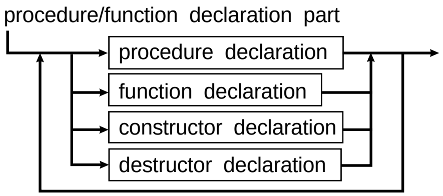
The *type declaration part* includes all type declarations local to the block.



The *variable declaration part* is composed of variable declarations local to the block.

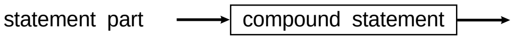


The procedure and function declarations local to the block make up the *procedure and function declaration part*.



The **exports** clause lists all procedures and functions that are exported by the current program or dynamic-link library. An **exports** clause is allowed only in the outermost declaration part of a program or dynamic-link library—it isn't allowed in the declaration part of a procedure, function, or unit. See "The exports clause" on page 135.

The *statement part* defines the statements or algorithmic actions to be executed by the block.



Rules of scope

The presence of an identifier or label in a declaration defines the identifier or label. Each time the identifier or label occurs again, it must be within the *scope* of this declaration.

Block scope

The scope of an identifier or label declared in a label, constant, type, variable, procedure, or function declaration stretches from the point of declaration to the end of the current block, and includes all blocks enclosed by the current block.

An identifier or label declared in an outer block can be *redeclared* in an inner block enclosed by the outer block. Before the point of declaration in the inner block, and after the end of the inner block, the identifier or label represents the entity declared in the outer block.

```
program Outer;          { Start of outer scope }
type
  I = Integer;          { define I as type Integer }
var
  T: I;                 { define T as an Integer variable }

procedure Inner;        { Start of inner scope }
type
  T = I;                { redefine T as type Integer }
var
  I: T;                 { redefine I as an Integer variable }
begin
  I := 1;
end;                    { End of inner scope }

begin
  T := 1;
end.                    { End of outer scope }
```

Record scope

The scope of a field identifier declared in a record-type definition extends from the point of declaration to the end of the record-type definition. Also, the scope of field identifiers includes field designators and **with** statements that operate on variable references of the given record type. See “Record types” on page 19.

Class scope

The scope of a component identifier declared in a class type extends from the point of declaration to the end of the class-type definition, and extends over all descendants of the class type and the blocks of all method declarations of the class type. Also, the scope of component identifiers includes field, method, and property designators, and **with** statements that operate on variables of the given class type. For more information on classes, see Chapter 9, Class Types.

Unit scope

The scope of identifiers declared in the interface section of a unit follows the rules of block scope and extends over all *clients* of the unit. In other words, programs or units containing **uses** clauses have access to the identifiers belonging to the interface parts of the units in those **uses** clauses.

Each unit in a **uses** clause imposes a new scope that encloses the remaining units used and the program or unit containing the **uses** clause. The first unit in a **uses** clause represents the outermost scope, and the last unit represents the innermost scope. This implies that if two or more units declare the same identifier, an unqualified reference to the identifier selects the instance declared by the last unit in the **uses** clause. If you use a qualified identifier (a unit identifier, followed by a period, followed by the identifier), every instance of the identifier can be selected.

The identifiers of Object Pascal's predefined constants, types, variables, procedures, and functions act as if they were declared in a block enclosing all used units and the entire program. In fact, these standard objects are defined in a unit called *System*, which is used by any program or unit before the units named in the **uses** clause. This means that any unit or program can redeclare the standard identifiers, but a specific reference can still be made through a qualified identifier, for example, *System.Integer* or *System.Writeln*.

Chapter

8

Procedures and functions

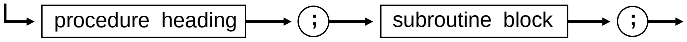
Procedures and functions let you nest additional blocks in the main program block. Each procedure or function declaration has a heading followed by a block. See Chapter 7, “Blocks, locality, and scope,” for a definition of a block. A procedure is activated by a procedure statement; a function is activated by the evaluation of an expression that contains its call and returns a value to that expression.

This chapter discusses the different types of procedure and function declarations and their parameters.

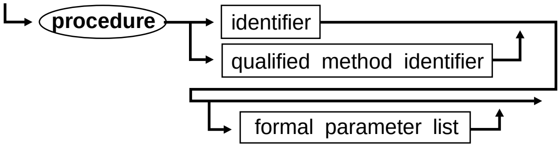
Procedure declarations

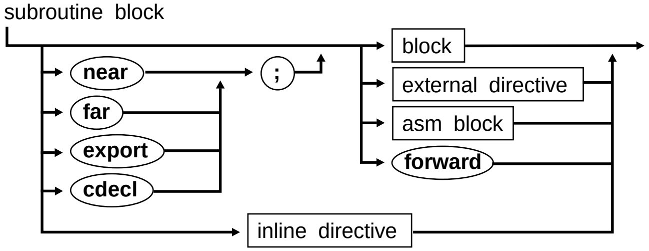
A *procedure declaration* associates an identifier with a block as a procedure; that procedure can then be activated by a procedure statement.

procedure declaration



procedure heading





The procedure heading names the procedure's identifier and specifies the formal parameters (if any). The syntax for a formal parameter list is shown in the section "Parameters" on page 75.

A procedure is activated by a procedure statement, which states the procedure's identifier and actual parameters, if any. The statements to be executed on activation are noted in the statement part of the procedure's block. If the procedure's identifier is used in a procedure statement within the procedure's block, the procedure is executed recursively (it calls itself while executing).

Here's an example of a procedure declaration:

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

Near and far declarations

Object Pascal supports two procedure and function call models: **near** and **far**. In terms of code size and execution speed, the near call model is the more efficient, but **near** procedures and functions can only be called from within the module they are declared in. On the other hand, **far** procedures and functions can be called from any module, but the code for a far call is slightly less efficient.

The compiler automatically selects the correct call model based on a procedure's or function's declaration: Procedures and functions declared in the **interface** part of a unit use the far call model—they can be called from other modules. Procedures and functions declared in a program or in the **implementation** part of a unit use the near call model—they can only be called from within that program or unit.

For some purposes, a procedure or function may be required to use the far call model. For example, if a procedure or function is to be assigned to a procedural variable, it has to use the far call model. The **$F** compiler directive can be used to override the compiler's automatic call model selection. Procedures and functions compiled in the {**$F+**} state always use the far call model; in the {**$F-**} state, the compiler automatically selects the correct model. The default state is {**$F-**}.

To force a specific call model, a procedure or function declaration can optionally specify a **near** or **far** directive before the block—if such a directive is present, it overrides the setting of the **$F** compiler directive as well as the compiler's automatic call model selection.

Export declarations

The **export** directive makes a procedure or function exportable by forcing the routine to use the far call model and generating special procedure entry and exit code.

Procedures and functions must be exportable in these cases:

- Procedures and functions that are exported by a DLL (dynamic-link library)
- Callback procedures and functions in a Windows program

Chapter 12, "Dynamic-link libraries," discusses how to export procedures and functions in a DLL. Even though a procedure or function is compiled with an **export** directive, the actual exporting of the procedure or function doesn't occur until the routine is listed in a library's **exports** clause.

Callback procedures and functions are routines in your application that are called by Windows and not by your application itself. Callback routines must be compiled with the **export** directive, but they don't have to be listed in an **exports** clause. Here are some examples of common callback procedures and functions:

- Window procedures
- Dialog procedures
- Enumeration callback procedures
- Memory-notification procedures
- Window-hook procedures (filters)

Object Pascal automatically generates *smart callbacks* for procedures and functions that are exported by a Windows program. Smart callbacks alleviate the need to use the *MakeProcInstance* and *FreeProcInstance* Windows API routines when creating callback routines. See "Entry and exit code" on page 176.

cdecl declarations

The **cdecl** directive specifies that a procedure or function should use *C calling conventions*. C calling conventions differ from Pascal calling conventions in that parameters are pushed on the stack in reverse order, and that the caller (as opposed to the callee) is responsible for removing the parameters from the stack after the call. The **cdecl** directive is useful for interfacing with dynamic-link libraries written in C or C++, but for regular (non-imported) procedures and functions, Pascal calling conventions are more efficient.

Forward declarations

A procedure or function declaration that specifies the directive **forward** instead of a block is a **forward** declaration. Somewhere after this declaration, the procedure must be defined by a *defining declaration*. The defining declaration can omit the formal parameter list and the function result, or it can optionally repeat it. In the latter case, the defining declaration's heading must match exactly the order, types, and names of parameters, and the type of the function result, if any.

No **forward** declarations are allowed in the **interface** part of a unit.

The **forward** declaration and the defining declaration must appear in the same procedure and function declaration part. Other procedures and functions can be declared between them, and they can call the forward-declared procedure. Therefore, mutual recursion is possible.

The **forward** declaration and the defining declaration constitute a complete procedure or function declaration. The procedure or function is considered declared at the **forward** declaration.

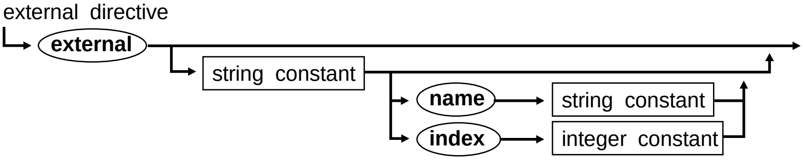
This is an example of a **forward** declaration:

```
procedure Walter(M, N: Integer); forward;

procedure Clara(X, Y: Real);
begin
   ⋮
  Walter(4, 5);
   ⋮
end;

procedure Walter;
begin
   ⋮
  Clara(8.3, 2.4);
   ⋮
end;
```

A procedure's or function's defining declaration can be an **external** or **assembler** declaration; however, it can't be a **near**, **far**, **export**, **interrupt**, or **inline** declaration or another **forward** declaration.

External declarations

With **external** declarations, you can interface with separately compiled procedures and functions written in assembly language. They also allow you to import procedures and functions from DLLs.



External directives consisting only of the reserved word **external** are used in conjunction with {**$L** filename} directives to link with **external** procedures and functions implemented in .OBJ files. For more details about linking with assembly language, see Chapter 20.

These are examples of **external** procedure declarations:

```
procedure MoveWord(var Source, Dest; Count: Word); external;
procedure MoveLong(var Source, Dest; Count: Word); external;

procedure FillWord(var Dest; Data: Integer; Count: Word); external;
procedure FillLong(var Dest; Data: Longint; Count: Word); external;

{$L BLOCK.OBJ}
```

External directives that specify a dynamic-link library name (and optionally an import name or an import ordinal number) are used to import procedures and functions from dynamic-link libraries. For example, this **external** declaration imports a function called *GlobalAlloc* from the DLL called KERNEL (the Windows kernel):

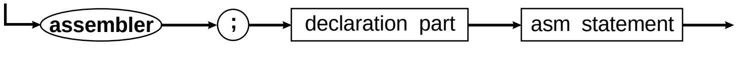
```
function GlobalAlloc(Flags: Word; Bytes: Longint): THandle; far;
external 'KERNEL' index 15;
```

To read more about importing procedures and functions from a DLL, see Chapter 12.

The **external** directive takes the place of the declaration and statement parts in an imported procedure or function. Imported procedures and functions must use the far call model selected by using a **far** procedure directive or a {**$F+**} compiler directive. Aside from this requirement, imported procedures and functions are just like regular procedures and functions.

Assembler declarations

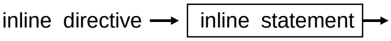
With **assembler** declarations, you can write entire procedures and functions in inline assembly language.



For more details on assembler procedures and functions, see Chapter 19.

Inline declarations

The **inline** directive enables you to write machine code instructions in place of a block of Object Pascal code.



See the inline statement syntax diagram on page 213.

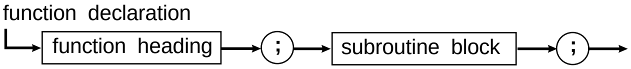
When a normal procedure or function is called, the compiler generates code that pushes the procedure's or function's parameters onto the stack and then generates a CALL instruction to call the procedure or function. When you call an **inline** procedure or function, the compiler generates code from the **inline** directive instead of the CALL. Therefore, an **inline** procedure or function is expanded every time you refer to it, just like a macro in assembly language.

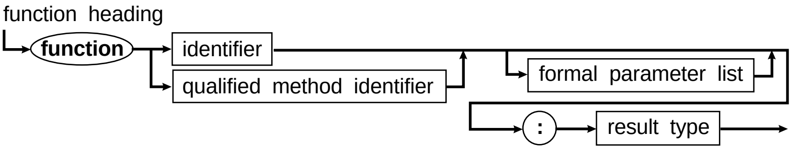
Here's a short example of two **inline** procedures:

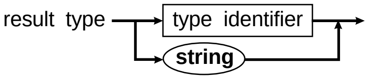
```
procedure DisableInterrupts; inline($FA);   { CLI }
procedure EnableInterrupts; inline($FB);    { STI }
```

Function declarations

A function declaration defines a block that computes and returns a value.







The function heading specifies the identifier for the function, the formal parameters (if any), and the function result type.

A function is activated by the evaluation of a function call. The function call gives the function's identifier and actual parameters, if any, required by the function. A function call appears as an operand in an expression. When the expression is evaluated, the function is executed, and the value of the operand becomes the value returned by the function.

The statement part of the function's block specifies the statements to be executed upon activation of the function. The block should contain at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value assigned. If no such assignment statement exists or if it isn't executed, the value returned by the function is undefined.

If the function's identifier is used in a function call within the function's block, the function is executed recursively.

Every function implicitly has a local variable *Result* of the same type as the function's return value. Assigning to *Result* has the same effect as assigning to the name of the function. In addition, however, you can refer to *Result* in an expression, which refers to the current return value rather than generating a recursive function call.

Functions can return any type, whether simple or complex, standard or user-defined, except old-style objects (as opposed to classes), and files of type text or file of. The only way to handle objects as function results is through object pointers.

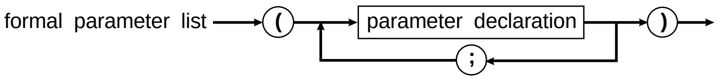
Following are examples of function declarations:

```
function Max(A: Vector; N: Integer): Extended;
var
  X: Extended;
  I: Integer;
begin
  X := A[1];
  for I := 2 to N do
    if X < A[I] then X := A[I];
  Max := X;
end;

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

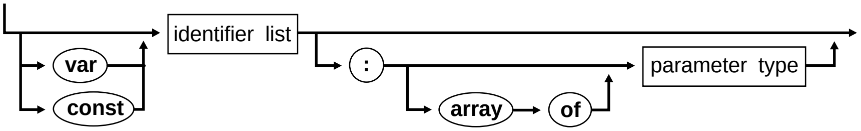
Like procedures, functions can be declared as **near**, **far**, **export**, **forward**, **external**, **assembler**, or **inline**.

Parameters

The declaration of a procedure or function specifies a *formal parameter list*. Each parameter declared in a formal parameter list is local to the procedure or function being declared and can be referred to by its identifier in the block associated with the procedure or function.



parameter declaration



There are four kinds of parameters: *value*, *constant*, *variable*, and *untyped*. These are characterized as follows:

- A parameter group without a preceding **var** and followed by a type is a list of value parameters.
- A parameter group preceded by **const** and followed by a type is a list of constant parameters.
- A parameter group preceded by **var** and followed by a type is a list of variable parameters.
- A parameter group preceded by **var** or **const** and *not* followed by a type is a list of untyped parameters.

String and array-type parameters can be *open parameters*. Open parameters are described on page 78. A variable parameter declared using the *OpenString* identifier, or using the **string** keyword in the {**$P+**} state, is an *open-string parameter*. A value, constant, or variable parameter declared using the syntax **array of** *T* is an *open-array parameter*.

Value parameters

A formal value parameter acts like a variable local to the procedure or function, except it gets its initial value from the corresponding actual parameter upon activation of the procedure or function. Changes made to a formal value parameter don't affect the value of the actual parameter.

A value parameter's corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of file type or of any structured type that contains a file type.

The actual parameter must be assignment-compatible with the type of the formal value parameter. If the parameter type is **string**, then the formal parameter is given a size attribute of 255.

Constant parameters

A formal constant parameter acts like a local read-only variable that gets its value from the corresponding actual parameter upon activation of the procedure or function. Assignments to a formal constant parameter are *not* allowed, and likewise a formal constant parameter *can't* be passed as an actual variable parameter to another procedure or function.

A constant parameter's corresponding actual parameter in a procedure statement or function must follow the same rules as an actual value parameter.

In cases where a formal parameter never changes its value during the execution of a procedure or function, a constant parameter should be used instead of a value parameter. Constant parameters allow the implementor of a procedure or function to protect against accidental assignments to a formal parameter. Also, for structured- and string-type parameters, the compiler can generate more efficient code when constant parameters are used instead of value parameters.

Variable parameters

A variable parameter is used when a value must be passed from a procedure or function to the caller. The corresponding actual parameter in a procedure statement or function call must be a variable reference. The formal variable parameter represents the actual variable during the activation of the procedure or function, so any changes to the value of the formal variable parameter are reflected in the actual parameter.

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The type of the actual parameter must be identical to the type of the formal variable parameter (you can bypass this restriction through untyped parameters).

Note File types can be passed only as variable parameters.

The **$P** compiler directive controls the meaning of a variable parameter declared using the **string** keyword. In the default {**$P+**} state, **string** indicates that the parameter is an open-string parameter. In the {**$P-**} state, **string** corresponds to a string type with a size attribute of 255. See page 78 for information on open-string parameters.

If referencing an actual variable parameter involves indexing an array or finding the object of a pointer, these actions are executed before the activation of the procedure or function.

Untyped parameters

When a formal parameter is an untyped parameter, the corresponding actual parameter can be any variable or constant reference, regardless of its type. An untyped parameter declared using the **var** keyword can be modified, whereas an untyped parameter declared using the **const** keyword is read-only.

Within the procedure or function, the untyped parameter is typeless; that is, it is incompatible with variables of all other types, unless it is given a specific type through a variable typecast.

This is an example of untyped parameters:

```
function Equal(var Source, Dest; Size: Word): Boolean;
type
  TBytes = array[0..65534] of Byte;
var
  N: Word;
begin
```

```
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

This function can be used to compare any two variables of any size. For example, given these declarations,

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

the function then calls

```
Equal(Vec1, Vec2, SizeOf(TVector))
Equal(Vec1, Vec2, SizeOf(Integer) * N)
Equal(Vec[1], Vec1[6], SizeOf(Integer) * 5)
Equal(Vec1[1], P, 4)
```

which compares *Vec1* to *Vec2*, the first *N* components of *Vec1* to the first *N* components of *Vec2*, the first five components of *Vec1* to the last five components of *Vec1*, and *Vec1[1]* to *P.X* and *Vec1[2]* to *P.Y*.

While untyped parameters give you greater flexibility, they can be riskier to use. The compiler can't verify that operations on untyped variables are valid.

Open parameters

Open parameters allow strings and arrays of varying sizes to be passed to the same procedure or function.

Open-string parameters

Open-string parameters can be declared in two ways:

- Using the **string** keyword in the {**$P+**} state
- Using the *OpenString* identifier

By default, parameters declared with the **string** keyword are open-string parameters. If, for reasons of backward compatibility, a procedure or function is compiled in the {$**P-**} state, the *OpenString* identifier can be used to declare open-string parameters. *OpenString* is declared in the *System* unit and denotes a special string type that can only be used in the declaration of string parameters. *OpenString* is not a reserved word; therefore, *OpenString* can be redeclared as a user-defined identifier.

For an open-string parameter, the actual parameter can be a variable of any string type. Within the procedure or function, the size attribute (maximum length) of the formal parameter will be the same as that of the actual parameter.

Open-string parameters behave exactly as variable parameters of a string type, except that they can't be passed as regular variable parameters to other procedures and functions. They can, however, be passed as open-string parameters again.

In this example, the *S* parameter of the *AssignStr* procedure is an open-string parameter:

```
procedure AssignStr(var S: OpenString);
begin
  S := '0123456789ABCDEF';
end;
```

Because *S* is an open-string parameter, variables of any string type can be passed to *AssignStr*:

```
var
  S1: string[10];
  S2: string[20];
begin
  AssignStr(S1);      { S1 = '0123456789' }
  AssignStr(S2);      { S2 = '0123456789ABCDEF' }
end;
```

Within *AssignStr*, the maximum length of the *S* parameter is the same as that of the actual parameter. Therefore, in the first call to *AssignStr*, the assignment to the *S* parameter truncates the string because the declared maximum length of *S1* is 10.

When applied to an open-string parameter, the *Low* standard function returns zero, the *High* standard function returns the declared maximum length of the actual parameter, and the *SizeOf* function returns the size of the actual parameter.

In the next example, the *FillString* procedure fills a string to its maximum length with a given character. Notice the use of the *High* standard function to obtain the maximum length of an open-string parameter.

```
procedure FillString(var S: OpenString; Ch: Char);
begin
  S[0] := Chr(High(S));          { Set string length }
  FillChar(S[1], High(S), Ch);   { Set string characters }
end;
```

Note Value and constant parameters declared using the *OpenString* identifier or the **string** keyword in the {**$P+**} state are *not* open-string parameters. Instead, such parameters behave as if they were declared using a string type with a maximum length of 255 and the *High* standard function always returns 255 for such parameters.

Open-array parameters

A formal parameter declared using the syntax

```
array of T
```

is an *open-array parameter*. *T* must be a type identifier, and the actual parameter must be a variable of type *T*, or an array variable whose element type is *T*. Within the procedure or function, the formal parameter behaves as if it was declared as

```
array[0..N - 1] of T
```

where *N* is the number of elements in the actual parameter. In effect, the index range of the actual parameter is mapped onto the integers 0 to *N* - 1. If the actual parameter is a simple variable of type *T*, it is treated as an array with one element of type *T*.

A formal open-array parameter can be accessed by element only. Assignments to an entire open array aren't allowed, and an open array can be passed to other procedures and functions only as an open-array parameter or as an untyped variable parameter.

Open-array parameters can be value, constant, and variable parameters and have the same semantics as regular value, constant, and variable parameters. In particular, assignments to elements of a formal open array constant parameter are not allowed, and assignments to elements of a formal open array value parameter don't affect the actual parameter.

Note For an open array value parameter, the compiler creates a local copy of the actual parameter within the procedure or function's stack frame. Therefore, be careful not to overflow the stack when passing large arrays as open array value parameters.

When applied to an open-array parameter, the *Low* standard function returns zero, the *High* standard function returns the index of the last element in the actual array parameter, and the *SizeOf* function returns the size of the actual array parameter.

The *Clear* procedure in the next example assigns zero to each element of an array of *Double*, and the *Sum* function computes the sum of all elements in an array of *Double*. Because the *A* parameter in both cases is an open-array parameter, the subroutines can operate on any array with an element type of *Double*.

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

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

When the element type of an open-array parameter is *Char*, the actual parameter may be a string constant. For example, given the procedure declaration,

```
procedure PrintStr(const S: array of Char);
var
  I: Integer;
begin
  for I := 0 to High(S) do
    if S[I] <> #0 then Write(S[I]) else Break;
end;
```

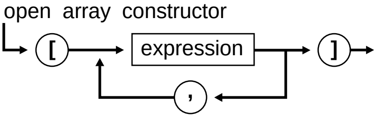
the following are valid procedure statements:

```
PrintStr('Hello world');
PrintStr('A');
```

When passed as an open-character array, an empty string is converted to a string with one element containing a NULL character, so the statement *PrintStr('')* is identical to the statement *PrintStr(#0)*.

Open-array constructors

Open-array constructors allow open-array parameters to be constructed directly within procedure and function calls. When a formal parameter of a procedure or function is an open-array value parameter or an open-array constant parameter, the corresponding actual parameter in a procedure or function call can be an open-array constructor.



An open-array constructor consists of one or more expressions separated by commas and enclosed in square brackets. Each expression must be assignment compatible with the element type of the open-array parameter. The use of an open-array constructor corresponds to creating a temporary array variable, and initializing the elements of the temporary array with the values given by the list of expressions. For example, given the declaration of the *Sum* function above, the statement

```
X := Sum([A, 3.14159, B + C]);
```

corresponds to

```
Temp[0] := A;
Temp[1] := 3.14159;
Temp[2] := B + C;
X := Sum(Temp);
```

where *Temp* is a temporary array variable with three elements of type *Double*.

Type variant open-array parameters

A type variant open-array parameter allows an open array of expressions of varying types to be passed to a procedure or function. A type variant open-array parameter is declared using the syntax

```
array of const
```

The **array of const** syntax is analogous to **array of** *TVarRec*. The *TVarRec* type is a variant record type which can represent values of integer, boolean, character, real, string, pointer, class, and class reference types. The *TVarRec* type is declared in the *System* unit as follows

```
type
  TVarRec = record
    case VType: Byte of
      vtInteger:  (VInteger: Longint);
      vtBoolean:  (VBoolean: Boolean);
      vtChar:     (VChar: Char);
      vtExtended: (VExtended: PExtended);
      vtString:   (VString: PString);
      vtPointer:  (VPointer: Pointer);
      vtPChar:    (VPChar: PChar);
      vtObject:   (VObject: TObject);
      vtClass:    (VClass: TClass);
  end;
```

The *VType* field determines which value field is currently defined. Notice that for real or string values, the *VExtended* or *VString* field contains a pointer to the value rather than the value itself. The *vtXXXX* value type constants are also declared in the *System* unit:

```
const
  vtInteger  = 0;
  vtBoolean  = 1;
  vtChar     = 2;
  vtExtended = 3;
  vtString   = 4;
  vtPointer  = 5;
  vtPChar    = 6;
  vtObject   = 7;
  vtClass    = 8;
```

The *MakeStr* function below takes a type variant open-array parameter and returns a string that is the concatenation of the string representations of the arguments. The *AppendStr*, *IntToStr*, *FloatToStr*, and *StrPas* functions used by *MakeStr* are defined in the *SysUtils* unit.

```
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:  AppendStr(Result, IntToStr(VInteger));
          vtBoolean:  AppendStr(Result, BoolChars[VBoolean]);
          vtChar:     AppendStr(Result, VChar);
          vtExtended: AppendStr(Result, FloatToStr(VExtended^));
          vtString:   AppendStr(Result, VString^);
          vtPChar:    AppendStr(Result, StrPas(VPChar));
          vtObject:   AppendStr(Result, VObject.ClassName);
          vtClass:    AppendStr(Result, VClass.ClassName);
        end;
  end;
```

When a formal parameter of a procedure or function is a type variant open-array value parameter or a type variant open-array constant parameter, the corresponding actual parameter in a procedure or function call can be an open-array constructor. The use of an open-array constructor for a type variant open-array parameter creates a temporary **array of** *TVarRec* and initializes the elements according to the type and value of each expression listed in the open-array constructor. For example, given the above function *MakeStr*, the statement

```
  S := MakeStr(['Test ', 100, '-', True, '-', 3.14159]);
```

produces the following string

```
  'Test 100-T-3.14159'
```

The following table lists the possible expression types in a type variant open-array constructor, and the corresponding value type codes.

Table 8-1 Type variant open-array expressions

Type code	Expression type
vtInteger	Any integer type
vtBoolean	Any boolean type
vtChar	Any character type
vtExtended	Any real type
vtString	Any string type
vtPointer	Any pointer type except *PChar*
vtPChar	*PChar* or **array**[0..X] **of** *Char*
vtObject	Any class type
vtClass	Any class reference type

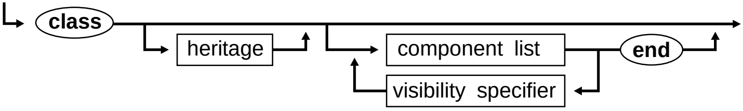
Chapter

9

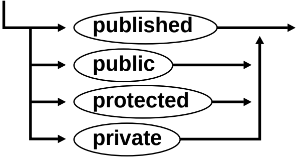
Class types

A class type is a structure consisting of a fixed number of components. The possible components of a class are *fields*, *methods*, and *properties*. Unlike other types, a class type can be declared only in a type declaration part in the outermost scope of a program or unit. Therefore, a class type can't be declared in a variable declaration part or within a procedure, function, or method block.

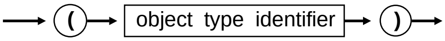
object type



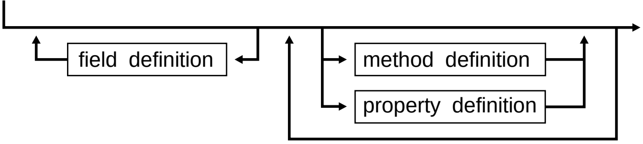
visibility specifier



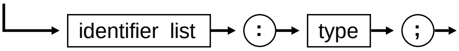
heritage



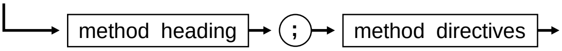
component list



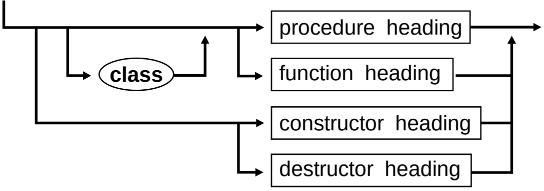
field definition



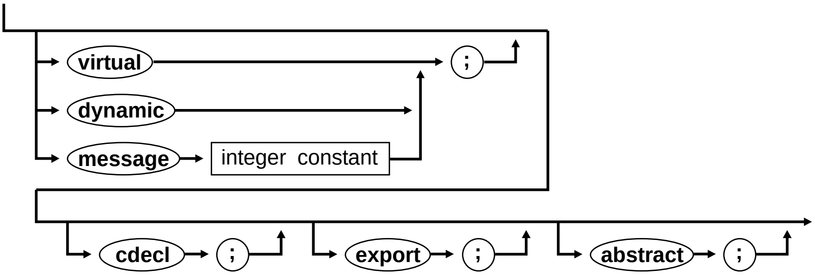
method definition



method heading



method directives



Instances and references

An *instance* of a class type is a dynamically allocated block of memory with a layout defined by the class type. Instances of a class type are also commonly referred to as *objects*. Objects are created using constructors, and destroyed using destructors. Each object of a class type has a unique copy of the fields declared in the class type, but all share the same methods.

A variable of a class type contains a *reference* to an object of the class type. The variable doesn't contain the object itself, but rather is a pointer to the memory block that has been allocated for the object. Like pointer variables, multiple class type variables can refer to the same object. Furthermore, a class type variable can contain the value **nil**, indicating that it doesn't currently reference an object.

Note Unlike a pointer variable, it isn't necessary to de-reference a class type variable to gain access to the referenced object. In other words, while it is necessary to write *Ptr^.Field* to access a field in a dynamically allocated record, the ^ operator is implied when accessing a component of an object, and the syntax is simply *Instance.Field*.

Throughout this book, the term *object reference* is used to denote a value of a class type. For example, a class-type variable contains an object reference, and a constructor returns an object reference.

In addition, the term *class reference* is used to denote a value of a class-reference type. For example, a class-type identifier is a class reference, and a variable of a class-reference type contains a class reference. Class-reference types are described in further detail on page 108.

Class components

The components of a class are fields, methods, and properties. Class components are sometimes referred to as members.

Fields

A field declaration in a class defines a data item that exists in each instance of the class. This is similar to a field of a record.

Methods

A method is a procedure or function that performs an operation on an object. Part of the call to a method specifies the object the method should operate on.

The declaration of a method within a class type corresponds to a **forward** declaration of that method. This means that somewhere after the class-type declaration, and within the same module, the method must be implemented by a defining declaration.

Within the implementation of a method, the identifier *Self* represents an implicit parameter that references the object for which the method was invoked.

Constructors and *destructors* are special methods that control construction and destruction of objects.

A constructor defines the actions associated with creating an object. When invoked, a constructor acts as a function that returns a reference to a newly allocated and initialized instance of the class type.

A destructor defines the actions associated with destroying an object. When invoked, a destructor will deallocate the memory that was allocated for the object.

A *class method* is a procedure or function that operates on a class reference instead of an object reference.

Properties

A property declaration in a class defines a named attribute for objects of the class and the actions associated with reading and writing the attribute. Properties are described in depth beginning on page 101

Inheritance

A class type can inherit components from another class type. If *T2* inherits from *T1*, then *T2* is a *descendant* of *T1*, and *T1* is an *ancestor* of *T2*. Inheritance is transitive; that is, if *T3* inherits from *T2*, and *T2* inherits from *T1*, then *T3* also inherits from *T1*. The *domain* of a class type consists of itself and all of its descendants.

A descendant class implicitly contains all the components defined by its ancestor classes. A descendant class can add new components to those it inherits. However, it can’t remove the definition of a component defined in an ancestor class.

The predefined class type *TObject* is the ultimate ancestor of all class types. If the declaration of a class type doesn't specify an ancestor type (that is, if the heritage part of the class declaration is omitted), the class type will be derived from *TObject*. *TObject* is declared by the *System* unit, and defines a number of methods that apply to all classes. For a description of these methods, see "The TObject and TClass types" on page 111.

Components and scope

The scope of a component identifier declared in a class type extends from the point of declaration to the end of the class-type definition, and extends over all descendants of the class type and the blocks of all method declarations of the class type. Also, the scope of component identifiers includes field, method, and property designators, and **with** statements that operate on variables of the given class type.

A component identifier declared in a class type can be *redeclared* in the block of a method declaration of the class type. In that case, the *Self* parameter can be used to access the component whose identifier was redeclared.

A component identifier declared in an ancestor class type can be redeclared in a descendant of the class type. Such redeclaration effectively hides the inherited component, although the **inherited** keyword can be used to bring the inherited component back into scope.

Forward references

The declaration of a class type can specify the reserved word **class** and nothing else, in which case the declaration is a forward declaration. A forward declaration must be resolved by a normal declaration of the class within the same type declaration part. Forward declarations allow mutually dependent classes to be declared. For example:

```
type
  TFigure = class;
  TDrawing = class
    Figure: TFigure;
    :
  end;
  TFigure = class
    Drawing: TDrawing;
    :
  end;
```

Class type compatibility rules

A class type is assigment-compatible with any ancestor class type; therefore, during program execution, a class-type variable can reference an instance of that type or an instance of any descendant type. For example, given the declarations

```
type
```

```
TFigure = class
  :
end;
TRectangle = class(TFigure)
  :
end;
TRoundRect = class(TRectangle)
  :
end;
TEllipse = class(TFigure)
  :
end;
```

a value of type *TRectangle* can be assigned to variables of type *TRectangle*, *TFigure*, and *TObject*, and during execution of a program, a variable of type *TFigure* might be either **nil** or reference an instance of *TFigure*, *TRectangle*, *TRoundRect*, *TEllipse*, or any other instance of a descendant of *TFigure*.

Component visibility

The visibility of a component identifier is governed by the visibility attribute of the component section that declares the identifier. The four possible visibility attributes are **published**, **public**, **protected**, and **private**.

Component identifiers declared in the component list that immediately follows the class type heading have the published visibility attribute if the class type is compiled in the {**$M+**} state or is derived from a class that was compiled in the {**$M+**} state. Otherwise, such component identifiers have the public visibility attribute.

Public components

Component identifiers declared in **public** sections have no special restrictions on their visibility.

Published components

The visibility rules for published components are identical to those of public components. The only difference between published and public components is that *run-time type information* is generated for fields and properties that are declared in a **published** section. This run-time type information enables an application to dynamically query the fields and properties of an otherwise unknown class type.

Note The Delphi Visual Component Library uses run-time type information to access the values of a component's properties when saving a loading form files. Also, the Delphi development environment uses a component's run-time type information to determine the list of properties shown in the Object Inspector.

A class type cannot have **published** sections unless it is compiled in the {**$M+**} state or is derived from a class that was compiled in the {**$M+**} state. The **$M** compiler directive controls the generation of run-time type information for a class. For further details on **$M**, see Appendix B.

Fields defined in a **published** section must be of a class type. Fields of all other types are restricted to **public**, **protected**, and **private** sections.

Properties defined in a **published** section cannot be array properties. Furthermore, the type of a property defined in a **published** section must be an ordinal type, a real type (*Single*, *Double*, *Extended*, or *Comp*, but not *Real*), a string type, a small set type, a class type, or a method pointer type. A small set type is a set type with a base type whose lower and upper bounds have ordinal values between 0 and 15. In other words, a small set type is a set that fits in a byte or a word.

Protected components

When accessing through a class type declared in the current module, the protected component identifiers of the class and its ancestors are visible. In all other cases, protected component identifiers are hidden.

Access to protected components of a class is restricted to the implementation of methods of the class and its descendants. Therefore, components of a class that are intended for use only in the implementation of derived classes are usually declared as protected.

Private components

The visibility of a component identifier declared in a **private** component section is restricted to the module that contains the class-type declaration. In other words, private component identifiers act like normal public component identifiers within the module that contains the class-type declaration, but outside the module, any private component identifiers are unknown and inaccessible. By placing related class types in the same module, these class types can gain access to each other's private components without making the private components known to other modules.

Static methods

Methods declared in a class type are by default *static*. When a static method is called, the *declared* (compile-time) type of the class or object used in the method call determines which method implementation to activate. In the following example, the *Draw* methods are static:

```
type
  TFigure = class
    procedure Draw;
    :
  end;
  TRectangle = class(TFigure)
    procedure Draw;
    :
  end;
```

The following section of code illustrates the effect of calling a static method. Even though in the second call to *Figure.Draw* the *Figure* variable references an object of class *TRectangle*, the call invokes the implementation of *TFigure.Draw* because the declared type of the *Figure* variable is *TFigure*.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw;                                 { Invokes TFigure.Draw }
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;                                 { Invokes TFigure.Draw }
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;                              { Invokes TRectangle.Draw }
  Rectangle.Destroy;
end;
```

Virtual methods

A method can be made *virtual* by including a **virtual** directive in its declaration. When a virtual method is called, the *actual* (run-time) type of the class or object used in the method call determines which method implementation to activate. The following is an example of a declaration of a virtual method:

```
type
  TFigure = class
    procedure Draw; virtual;
    :
  end;
```

A virtual method can be *overridden* in a descendant class. When an **override** directive is included in the declaration of a method, the method overrides the inherited implementation of the method. An override of a virtual method must match exactly the order and types of the parameters, and the function result type (if any), of the original method.

The *only* way a virtual method can be overridden is through the **override** directive. If a method declaration in a descendant class specifies the same method identifier as an inherited method, but doesn't specify an **override** directive, the new method declaration will *hide* the inherited declaration, but not override it.

Assuming the declaration of class *TFigure* above, the following two descendant classes override the *Draw* method:

```
type
  TRectangle = class(TFigure)
    procedure Draw; override;
    :
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
    :
  end;
```

The following section of code illustrates the effect of calling a virtual method through a class-type variable whose actual type varies at run time:

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;                              { Invokes TRectangle.Draw }
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;                              { Invokes TEllipse.Draw }
  Figure.Destroy;
end;
```

Dynamic methods

A method is made dynamic by including a **dynamic** directive in its declaration. Dynamic methods are semantically identical to virtual methods. Virtual and dynamic methods differ only in the implementation of method call dispatching at run time; for all other purposes, the two types of methods can be considered equivalent.

In the implementation of virtual methods, the compiler favors speed of call dispatching over code size. The implementation of dynamic methods on the other hand favors code size over speed of call dispatching.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful only in situations where a base class declares a large number of virtual methods, and an application declares a large number of descendant classes with few overrides of the inherited virtual methods.

Abstract methods

An *abstract* method is a virtual or dynamic method whose implementation isn’t defined in the class declaration in which it appears; its definition is instead deferred to descendant classes. An abstract method in effect defines an interface, but not the underlying operation.

A method is abstract if an **abstract** directive is included in its declaration. A method can be declared **abstract** only if it is first declared **virtual** or **dynamic**. The following is an example of a declaration of an abstract method.

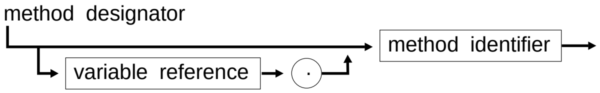
```
type
  TFigure = class
    procedure Draw; virtual; abstract;
    :
  end;
```

An override of an abstract method is identical to an override of a normal virtual or dynamic method, except that in the implementation of the overriding method, an **inherited** method isn't available to call.

Calling an abstract method through an object that hasn't overridden the method will generate an exception at run time.

Method activations

A method is activated (or called) through a function call or procedure statement consisting of a *method designator* followed by an actual parameter list. This type of call is known as *method activation*.



The variable reference specified in a method designator must denote an object reference or a class reference, and the method identifier must denote a method of that class type.

The instance denoted by a method designator becomes an implicit actual parameter of the method; it corresponds to a formal parameter named *Self* that possesses the class type corresponding to the activated method.

Within a **with** statement that references an object or a class, the variable-reference part of a method designator can be omitted. In that case, the implicit *Self* parameter of the method activation becomes the instance referenced by the **with** statement.

Method implementations

The declaration of a method within a class type corresponds to a **forward** declaration of that method. Somewhere after the class-type declaration, and within the same module, the method must be implemented by a defining declaration. For example, given the class-type declaration

```
type
  TFramedLabel = class(TLabel)
  protected
    procedure Paint; override;
  end;
```

the *Paint* method must later be implemented by a defining declaration. For example,

```
procedure TFramedLabel.Paint;
begin
  inherited Paint;
  with Canvas do
  begin
    Brush.Color := clWindowText;
    Brush.Style := bsSolid;
    FrameRect(ClientRect);
    end;
  end;
```

For procedure and function methods, the defining declaration takes the form of a normal procedure or function declaration, except that the procedure or function identifier in the heading is a *qualified method identifier*. A qualified method identifier consists of a class-type identifier followed by a period (.) and then by a method identifier.

For constructors and destructors, the defining declaration takes the form of a procedure method declaration, except that the **procedure** reserved word is replaced by **constructor** or **destructor**.

A method's defining declaration can optionally repeat the formal parameter list of the method heading in the class type. The defining declaration's method heading must match exactly the order, types, and names of the parameters, and the type of the function result, if any.

In the defining declaration of a method, there is always an implicit parameter with the identifier *Self*, corresponding to a formal parameter of the class type. Within the method block, *Self* represents the instance for which the method was activated.

The scope of a component identifier in a class type extends over any procedure, function, constructor, or destructor block that implements a method of the class type. The effect is the same as if the entire method block was embedded in a **with** statement of the form:

```
with Self do
begin
  :
  :
end;
```

Within a method block, the reserved word **inherited** can be used to access redeclared and overridden component identifiers. For example, in the implementation of the *TFramedLabel.Paint* method above, **inherited** is used to invoke the inherited implementation of the *Paint* method. When an identifier is prefixed with **inherited**, the search for the identifier begins with the immediate ancestor of the enclosing method's class type.

Constructors and destructors

Constructors and destructors are special methods that control construction and destruction of objects. A class can have zero or more constructors and destructors for objects of the class type. Each is specified as a component of the class in the same way as a procedure or function method, except that the reserved words **constructor** and **destructor** begin each declaration instead of **procedure** and **function**. Like other methods, constructors and destructors can be inherited.

Constructors

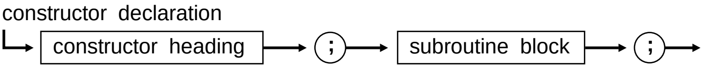
Constructors are used to create and initialize new objects. Typically, the initialization is based on values passed as parameters to the constructor.

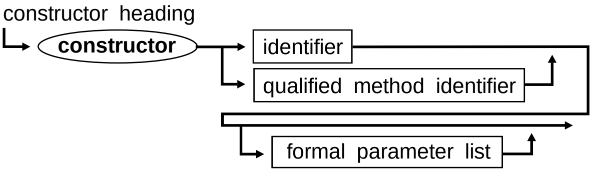
Unlike an ordinary method, which must be invoked on an object reference, a constructor can be invoked on either a class reference or an object reference.

To create a new object, a constructor must be invoked on a class reference. When a constructor is invoked on a class reference, the following actions take place:

- Storage for a new object is allocated from the heap.
- The allocated storage is cleared. This causes the ordinal value of all ordinal type fields to become zero, the value of all pointer and class-type fields to become **nil**, and the value of all string fields to become empty.
- The user-specified actions of the constructor are executed.
- A reference to the newly allocated and initialized object is returned from the constructor. The type of the returned value is the same as the class type specified in the constructor call.

When a constructor is invoked on an object reference, the constructor acts like a normal procedure method. This means that a new object is *not* allocated and cleared, and that the constructor call does *not* return an object reference. Instead, the constructor operates on the specified object reference, and only executes the user specified actions given in the constructor's statement part. A constructor is typically invoked on an object reference only in conjunction with the **inherited** keyword to execute an inherited constructor.





An example of a class type and its associated constructor follows below:

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    :
  end;
```

```
constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);                { Initialize inherited parts }
  Width := 65;                            { Change inherited properties }
  Height := 65;
  FPen := TPen.Create;                    { Initialize new fields }
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

The first action of a constructor is almost always to call an inherited constructor to initialize the inherited fields of the object. Following that, the constructor then initializes the fields of the object that were introduced in the class. Because a constructor always clears the storage it allocates for a new object, all fields automatically have a default value of zero (ordinal types), **nil** (pointer and class types), or empty (string types). Unless a field's default value is non-zero, there is no need to initialize the field in a constructor.

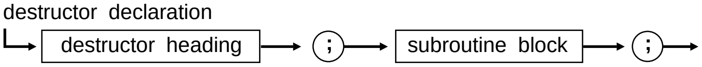
If an exception occurs during execution of a constructor that was invoked on a class reference, the *Destroy* destructor is automatically called to destroy the unfinished object. The effect is the same as if the entire statement part of the constructor were embedded in a **try**...**finally** statement of this form:

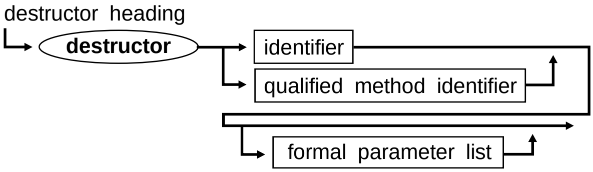
```
try
  :                 { User defined actions }
  :
except              { On any exception }
  Destroy;          { Destroy unfinished object }
  raise;            { Re-raise exception }
end;
```

Like other methods, constructors can be virtual. When invoked through a class-type identifier, as is usually the case, a virtual constructor is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects—that is, construction of objects whose types aren’t known at compile time, as described on page 109.

Destructors

Destructors are used to destroy objects. When a destructor is invoked, the user-defined actions of the destructor are executed, and then the storage that was allocated for the object is disposed of. The user-defined actions of a destructor typically consist of destroying any embedded objects and releasing any resources that were allocated by the object.





The following example shows how the destructor that was declared for the *TShape* class in the preceding section might be implemented.

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

The last action of a destructor is typically to call the inherited destructor to destroy the inherited fields of the object.

While it is possible to declare multiple destructors for a class, it is recommended that classes only implement overrides of the inherited *Destroy* destructor. *Destroy* is a parameterless virtual destructor declared in *TObject*, and because *TObject* is the ultimate ancestor of every class, the *Destroy* destructor always available for any object.

As described in the preceding section on constructors, if an exception occurs during the execution of a constructor, the *Destroy* destructor is invoked to destroy the unfinished object. This means that destructors must be prepared to handle destruction of *partially constructed* objects. Because a constructor sets all fields of a new object to null values before executing any user defined actions, any class-type or pointer-type fields in a partially constructed object are always **nil**. A destructor should therefore always check for **nil** values before performing operations on class-type or pointer-type fields.

Referring to the *TShape.Destroy* destructor mentioned earlier, note that the *Free* method (which is inherited from *TObject*) is used to destroy the objects referenced by the *FPen* and *FBrush* fields. The implementation of the *Free* method is

```
procedure TObject.Free;
begin
  if Self <> nil then Destroy;
end;
```

The *Free* method is a convenient way of checking for **nil** before invoking *Destroy* on an object reference. By calling *Free* instead of *Destroy* for any class-type fields, a destructor is automatically prepared to handle partially constructed objects resulting from constructor exceptions. For that same reason, direct calls to *Destroy* aren't recommended.

Class operators

Object Pascal defines two operators, **is** and **as**, that operate on class and object references.

The is operator

The **is** operator is used to perform *dynamic type checking*. Using the **is** operator, you can check whether the *actual* (run-time) type of an object reference belongs to a particular class. The syntax of the **is** operator is

```
ObjectRef is ClassRef
```

where *ObjectRef* is an object reference and *ClassRef* is a class reference. The **is** operator returns a boolean value. The result is *True* if *ObjectRef* is an instance of the class denoted by *ClassRef* or an instance of a class derived from the class denoted by *ClassRef*. Otherwise, the result is *False*. If *ObjectRef* is **nil**, the result is always *False*. If the declared types of *ObjectRef* and *ClassRef* are known not to be related—that is if the declared type of *ObjectRef* is known not to be an ancestor of, equal to, or a descendant of *ClassRef*—the compiler reports a type-mismatch error.

The **is** operator is often used in conjunction with an **if** statement to perform a *guarded typecast*. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

Here, if the **is** test is *True*, it is safe to typecast *ActiveControl* to be of class *TEdit*.

The rules of operator precedence group the **is** operator with the relational operators (=, <>, <, >, <=, >=, and **in**). This means that when combined with other boolean expressions using the **and** and **or** operators, **is** tests must be enclosed in parentheses:

```
if (Sender is TButton) and (TButton(Sender).Tag <> 0) then ...;
```

The as operator

The **as** operator is used to perform *checked typecasts*. The syntax of the **as** operator is

```
ObjectRef as ClassRef
```

where *ObjectRef* is an object reference and *ClassRef* is a class reference. The resulting value is a reference to the same object as *ObjectRef*, but with the type given by *ClassRef*. When evaluated at run time, *ObjectRef* must be **nil**, an instance of the class denoted by *ClassRef*, or an instance of a class derived from the class denoted by *ClassRef*. If none of these conditions are *True*, an exception is raised. If the declared types of *ObjectRef* and *ClassRef* are known not to be related—that is, if the declared type of *ObjectRef* is known not to be an ancestor of, equal to, or a descendant of *ClassRef*—the compiler reports a type-mismatch error.

The **as** operator is often used in conjunction with a **with** statement. For example,

```
with Sender as TButton do
begin
  Caption := '&Ok';
```

```
    OnClick := OkClick;
  end;
```

The rules of operator precedence group the **as** operator with the multiplying operators (*, /, **div**, **mod**, **and**, **shl**, and **shr**). This means that when used in a variable reference, an **as** typecast must be enclosed in parentheses:

```
  (Sender as TButton).Caption := '&Ok';
```

Message handling

Message handler methods are used to implement user-defined responses to *dynamically dispatched messages*. Delphi's Visual Class Library uses message handler methods to implement Windows message handling.

Message handler declarations

A message handler method is defined by including a **message** directive in the method declaration.

```
  type
    TTextBox = class(TCustomControl)
    private
      procedure WMChar(var Message: TWMChar); message WM_CHAR;
      ...
    end;
```

A message handler method must be a procedure that takes a single variable parameter, and the **message** keyword must be followed by an integer constant between 0 and 32767 which specifies the *message ID*.

Note When declaring a message handler method for a VCL control, the integer constant specified in the message directive must be a Windows message ID. The *Messages* unit defines all Windows message IDs and their corresponding message records.

In contrast to a regular method, a message handler method does not have to specify an **override** directive in order to override an inherited message handler. In fact, a message handler override doesn't even have to specify the same method identifier and parameter name and type as the method it overrides. The message ID solely determines which message the method will respond to, and whether or not it is an override.

Message handler implementations

The implementation of a message handler method corresponds to that of a normal method. For example, the *TTextBox.WMChar* message handler method defined above might be implemented as follows

```
  procedure TTextBox.WMChar(var Message: TWMChar);
  begin
    if Chr(Message.CharCode) = #13 then
```

```
      ProcessEnter
    else
      inherited;
  end;
```

The implementation of a message handler method can call the inherited implementation using an **inherited** statement as shown above. The **inherited** statement automatically passes the message record as a parameter to the inherited method. The **inherited** statement invokes the first message handler with the same message ID found in the most derived ancestor class (that is, the ancestor class that is closest to the class in the inheritance hierarchy). If none of the ancestor classes implement a message handler for the given message ID, the **inherited** statement instead calls the *DefaultHandler* virtual method, which is inherited from *TObject* and therefore present in any class.

The effect of the **inherited** statement is that for a particular message ID, a class does not need to know whether its parent classes implement a handler for the message–an inherited implementation always appears to be available.

Message dispatching

Message handler methods are typically not called directly. Instead, messages are dispatched to an object using the *Dispatch* method defined in the *TObject* class. *Dispatch* is declared as

```
procedure TObject.Dispatch(var Message);
```

The *Message* parameter passed to *Dispatch* must be a record, and the first entry in the record must be a field of type *Cardinal* which contains the message ID of the message being dispatched. For example

```
type
  TMessage = record
    Msg: Cardinal;
    ...
  end;
```

A message record can contain any number of additional fields that define message specific information.

A call to *Dispatch* invokes the most derived implementation of a message handler for the given message ID. In other words, *Dispatch* invokes the first message handler with a matching message ID found by examining the class itself, its ancestor, its ancestor's ancestor, and so on until *TObject* is reached. If the class and its ancestors do not define a handler for the given message ID, *Dispatch* will instead invoke the *DefaultHandler* method.

The *DefaultHandler* method is declared in *TObject* as follows

```
procedure DefaultHandler(var Message); virtual;
```

The implementation of *DefaultHandler* in *TObject* simply returns without performing any actions. By overriding *DefaultHandler*, a class can implement default handling of messages. As described above, *DefaultHandler* is invoked when *Dispatch* is called to dispatch a message for which the class implements no message handler. *DefaultHandler* is also invoked when a message handler method executes an **inherited** statement for which no inherited message handler exists.

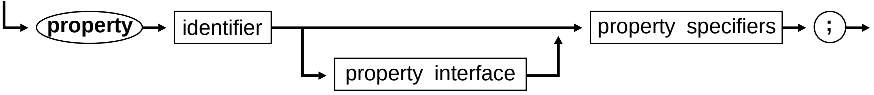
Note The *DefaultHandler* method for a VCL control invokes the *DefWindowProc* default message handling function defined by Windows.

Properties

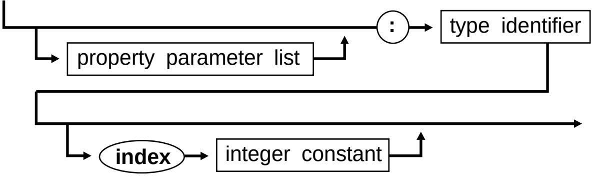
A property definition in a class declares a named attribute for objects of the class and the actions associated with reading and writing the attribute. Examples of properties are the caption of a form, the size of a font, the name of a database table, and so on.

Properties are a natural extension of fields in an object. Both can be used to express attributes of an object, but whereas fields are merely storage locations which can be examined and modified at will, properties provide greater control over *access* to attributes, they provide a mechanism for associating *actions* with the reading and writing of attributes, and they allow attributes to be *computed*.

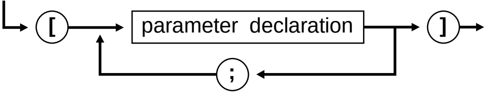
property definition



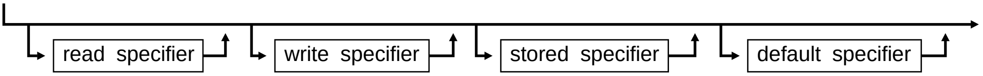
property interface



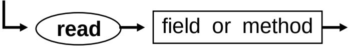
property parameter list



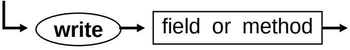
property specifiers



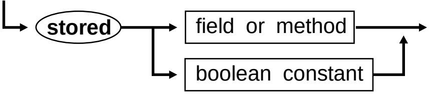
read specifier

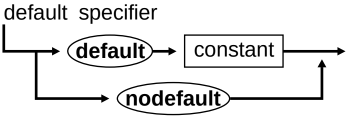


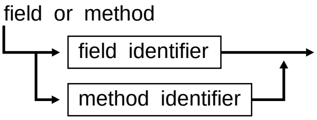
write specifier



stored specifier







Property definitions

The definition of a property specifies the name and type of the property, and the actions associated with reading (examining) and writing (modifying) the property. A property can be of any type except a file type.

The declarations below define an imaginary *TCompass* control which has a *Heading* property that can assume values from 0 to 359 degrees. The definition of the *Heading* property further states that its value is read from the *FHeading* field, and that its value is written using the *SetHeading* method.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    :
  end;
```

Property access

When a property is referenced in an expression, its value is read using the field or method listed in the **read** specifier, and when a property is referenced in an assignment statement, its value is written using the field or method listed in the **write** specifier. For example, assuming that *Compass* is an object reference of the *TCompass* type defined above, the statements

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the *TCompass* class, no action is associated with reading the *Heading* property. The read operation simply consists of examining the value stored in the *FHeading* field. On the other hand, assigning a value to the *Heading* property translates into a call to the *SetHeading* method, which not only stores the new value in the *FHeading* field, but also performs whatever actions are required to update the user interface of the compass control. For example, *SetHeading* might be implemented as this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;
  end;
end;
```

Note Unlike fields, properties can't be passed as variable parameters, and it isn't possible to take the address of a property using the @ operator. This is true even if the **read** and **write** specifiers both list a field identifier, thereby ensuring that a future implementation of the property is free to change one or both access specifiers to list a method.

Access specifiers

The **read** and **write** specifiers of a property definition control how a property is accessed. A property definition must include at least a **read** or a **write** specifier, but isn't required to include both. If a property definition includes only a **read** specifer, then the property is said to be *read-only* property, and if a property definition includes only a **write** specifier, then the property is said to be *write-only property*. If a property definition includes both a **read** and a **write** specifier, the property is said to be *read-write* property. It is an error to assign a value to a read-only property. Likewise, it is an error to use a write-only property in an expression.

The **read** or **write** keyword in an access specifier must be followed by a field identifier or a method identifier. The field or method can belong to the class type in which the property is defined, in which case the field or method definition must precede the property definition, or it can belong to an ancestor class, in which case the field or method must be visible in the class containing the property definition.

Fields and methods listed in access specifiers are governed by the following rules:

- If an access specifier lists a field identifier, then the field type must be identical to the property type.
- If a **read** specifier lists a method identifier, then the method must be a parameterless function method, and the function result type must be identical to the property type.
- If a **write** specifier lists a method identifier, then the method must be a procedure method that takes a single value or constant parameter of the same type as the property type.

For example, if a property is defined as

```
property Color: TColor read GetColor write SetColor;
```

then preceding the property definition, the *GetColor* method must be defined as

```
function GetColor: TColor;
```

and the *SetColor* method must be defined as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

Array properties

Array properties allow the implementation of *indexed* properties. Examples of array properties include the items of a list, the child controls of a control, and the pixels of a bitmap.

The definition of an array property includes an *index parameter list* which specifies the names and types of the indexes of the array property. For example,

```
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;
```

The format of an index parameter list is the same as that of a procedure or function's formal parameter list, except that the parameter declarations are enclosed in square brackets instead of parentheses. Note that unlike array types, which can only specify ordinal type indexes, array properties allow indexes of any type. For example, the *Values* property declared previously might represent a lookup table in which a string index is used to look up a string value.

An access specifier of an array property must list a method identifier. In other words, the **read** and **write** specifiers of an array property aren’t allowed to specify a field name. The methods listed in array property access specifiers are governed by the following rules:

- The method listed in the **read** specifier of an array property must be a function that takes the same number and types of parameters as are listed in the property's index parameter list, and the function result type must be identical to the property type.
- The method listed in the **write** specifier of an array property must be a procedure with the same number and types of parameters as are listed in the property's index parameter list, plus an additional value or constant parameter of the same type as the property type.

For example, for the array properties declared above, the access methods might be declared as

```
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);
```

An array property is accessed by following the property identifier with a list of actual parameters enclosed in square brackets. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

The definition of an array property can be followed by a **default** directive, in which case the array property becomes the *default array property* of the class. For example

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    :
  end;
```

If a class has a default array property, then access to the array property of the form *Instance.Property[...]* can be abbreviated to *Instance[...]*. For example, given that *StringArray* is an object reference of the *TStringArray* class defined above, the construct

```
StringArray.Strings[Index]
```

can be abbreviated to

```
StringArray[Index]
```

When an object reference is followed by a list of indexes enclosed in square brackets, the compiler automatically selects the class type's default array property, or issues an error if the class type has no default array property.

If a class defines a default array property, derived classes automatically inherit the default array property. It isn’t possible for a derived class to redeclare or hide the default array property.

Index specifiers

The definition of a property can optionally include an *index specifier*. Index specifiers allow a number of properties to share the same access methods. An index specifier consists of the directive **index** followed by an integer constant with a value between –32767 and 32767. For example

```
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;
```

An access specifier of a property with an **index** specifier must list a method identifier. In other words, the **read** and **write** specifiers of a property with an **index** specifier aren't allowed to list a field name.

When accessing a property with an **index** specifier, the integer value specified in the property definition is automatically passed to the access method as an extra parameter. For that reason, an access method for a property with an **index** specifier must take an extra value parameter of type *Integer*. For a property read function, the extra parameter must be the last parameter. For a property write procedure, the extra parameter must be the second to last parameter, that is it must immediately precede the parameter that specifies the new property value.

Assuming that *Rectangle* is an object reference of the *TRectangle* type defined above, the statement

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage specifiers

The optional **stored**, **default**, and **nodefault** specifiers of a property definition are called *storage specifiers*. They control certain aspects of the run-time type information that is generated for **published** properties. Storage specifiers are only supported for normal (non-array) properties.

Storage specifiers have no semantic effects on a property, that is they do not affect how a property is used in program code. The Delphi Visual Component Library, however, uses the information generated by storage specifiers to control *filing* of a component's properties—the automatic saving and loading of a component's property values in a form file. The **stored** directive controls whether a property is filed, and the **default** and **nodefault** properties control the value that is considered a property's default value.

If present in a property definition, the **stored** keyword must be followed by a boolean constant (*True* or *False*), the identifier of a field of type *Boolean*, or the identifier of a parameterless function method with returns a value of type *Boolean*. If a property definition doesn't include a **stored** specifier, the results are the same as if a **stored** *True* specifier were included.

The **default** and **nodefault** specifiers are supported only for properties of ordinal types and small set types. If present in a property definition, the **default** keyword must be followed by a constant of the same type as the property. If a property definition doesn’t (or can’t) include a **default** or **nodefault** specifier, the results are the same as if a **nodefault** specifier were included.

When saving a component's state, the Delphi Visual Component Library iterates over all of the component's **published** properties. For each property, the result of evaluating the boolean constant, field, or function method of the **stored** specifier controls whether the property is saved. If the result is *False*, the property isn’t saved. If the result is *True*, the property's current value is compared to the value given in the **default** specifier (if present). If the current value is equal to the default value, the property isn’t saved. Otherwise, if current value is different from the default value, or if the property has no default value, the property is saved.

Property overrides

A property definition that doesn’t include a property interface is called a *property override*. A property override allows a derived class to change the visibility, access specifiers, and storage specifiers of an inherited property.

In its simplest form, a property override specifies only the reserved word **property** followed by an inherited propery identifier. This form is used to change the visibility of a property. If, for example, a base class defines a property in a **protected** section, a derived class can raise the visibility of the property by declaring a property override in a **public** or **published** section.

A property override can include a **read**, **write**, **stored**, and **default** or **nodefault** specifier. Any such specifier overrides the corresponding inherited specifier. Note that a property override can change an inherited access specifier or add a missing access specifier, but it can’t remove an access specifier.

The following declarations illustrate the use of property overrides to change the visibility, access specifiers, and storage specifiers of inherited properties:

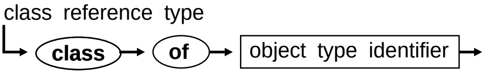
```
type
  TBase = class
    :
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    :
  end;

type
  TDerived = class(TBase)
    :
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
```

```
      :
    end;
```

The property override of *Size* adds a **write** specifier to allow the *Size* property to be modified. The property overrides of *Text* and *Color* change the visibility of the properties from protected to published. In addition, the property override of *Color* specifies that the property should be filed if its value isn't *clBlue*.

Class-reference types

Class-reference types allow operations to be performed directly on classes. This contrasts with class types, which allow operations to be performed on instances of classes. Class-reference types are sometimes referred to as *metaclasses* or *metaclass types*.



Class-reference types are useful in the following situations:

- With a virtual constructor to create an object whose actual type is unknown at compile time
- With a class method to perform an operation on a class whose actual type is unknown at compile time
- As the right operand of an **is** operator to perform a dynamic type check with a type that is unknown at compile time
- As the right operand of an **as** operator to perform a checked typecast to a type that is unknown at compile time

The declaration of a class-reference type consists of the reserved words **class of** followed by a class-type identifier. For example,

```
type
  TComponent = class(TPersistent)
    :
  end;
  TComponentClass = class of TComponent;
  TControl = class(TComponent)
    :
  end;
  TControlClass = class of TControl;

var
  ComponentClass: TComponentClass;
  ControlClass: TControlClass;
```

The previous declarations define *TComponentClass* as a type that can reference class *TComponent*, or any class that derives from *TComponent*, and *TControlClass* as a type that can reference class *TControl*, or any class that derives from *TControl*.

Class-type identifiers function as *values* of their corresponding class-reference types. For example, in addition to its other uses, the *TComponent* identifier functions as a value of type *TComponentClass*, and the *TControl* identifier functions as a value of type *TControlClass*.

A class-reference type value is assignment-compatible with any ancestor class-reference type. Therefore, during program execution, a class-reference type variable can reference the class it was defined for or any descendant class of the class it was defined for. Referring to the previous declarations, the assignments

```
ComponentClass := TComponent;      { Valid }
ComponentClass := TControl;        { Valid }
```

are both valid. Of these assignments,

```
ControlClass := TComponent;        { Invalid }
ControlClass := TControl;          { Valid }
```

only the second one is valid, however. The first assignment is an error because *TComponent* isn't a descandant of *TControl*, and therefore not a value of type *TControlClass*.

A class-reference type variable can be **nil**, which indicates that the variable doesn't currently reference a class.

Every class inherits (from *TObject*) a method function called *ClassType*, which returns a reference to the class of an object. The type of the value returned by *ClassType* is *TClass*, which is declared as **class of** *TObject*. This means that the value returned by *ClassType* may have to be typecast to a more specific descendant type before it can be used, for example

```
if Control <> nil then
  ControlClass := TControlClass(Control.ClassType) else
  ControlClass := nil;
```

Constructors and class references

A constructor can be invoked on a variable reference of a class-reference type. This allows *polymorphic construction* of objects, that is construction of objects whose actual type isn't known at compile time. For example,

```
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;
```

The *CreateControl* function uses a class-reference type parameter to specify which class of control to create. It subsequently uses the class-reference type parameter to invoke the *Create* constructor of the class. Because class-type identifiers also function as class-reference type values, calls to *CreateControl* can simply specify the identifier of the class to create an instance for. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
CreateControl(TButton, 'Button1', 120, 10, 80, 30);
```

A constructor can be invoked on a variable reference of a class-reference type. This allows *polymorphic construction* of objects, that is construction of objects whose actual type isn't known at compile time.

Constructors that are invoked through class-reference types are usually virtual. That way, the constructor implementation that is called depends on the *actual* (run-time) class type selected by the class reference.

Class methods

A class method is a method that operates on a class reference instead of an object reference. The definition of a class method must include the reserved word **class** before the **procedure** or **function** keyword that starts the definition. For example,

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    :
  end;
```

The defining declaration of a class method must also start with the reserved word **class**. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  :
end;
```

In the defining declaration of a class method, the identifier *Self* represents the class for which the method was activated. The type of *Self* in a class method is **class of** *ClassType*, where *ClassType* is the class type for which the method is implemented. Because *Self* doesn't represent an object reference in a class method, it isn't possible to use *Self* to access fields, properties, and normal methods. It is possible, however, to call constructors and other class methods through *Self*.

A class method can be invoked through a class reference or an object reference. When invoked through an object reference, the class of the given object reference is passed as the *Self* parameter.

The TObject and TClass types

The *System* unit defines two types, *TObject* and *TClass*, which serve as the *root* types for all class types and class-reference types. The declarations of the two types are shown below.

```
type
  TObject = class;
  TClass = class of TObject;
  TObject = class
    constructor Create;
    destructor Destroy; virtual;
    class function ClassInfo: Pointer;
    class function ClassName: string;
    class function ClassParent: TClass;
    function ClassType: TClass;
    procedure DefaultHandler(var Message); virtual;
    procedure Dispatch(var Message);
    function FieldAddress(const Name: string): Pointer;
    procedure Free;
    procedure FreeInstance; virtual;
    class function InheritsFrom(AClass: TClass): Boolean;
    class procedure InitInstance(Instance: Pointer): TObject;
    class function InstanceSize: Word;
    class function NewInstance: TObject; virtual;
    class function MethodAddress(const Name: string): Pointer;
    class function MethodName(Address: Pointer): string;
  end;
```

Chapter 10

Exceptions

An exception is generally an error condition or other event that interrupts normal flow of execution in an application. When an exception is *raised*, it causes control to be transferred from the current point of execution to an *exception handler*. Object Pascal's exception handling support provides a structured means of separating normal program logic from error handling logic, greatly increasing the maintainability and robustness of applications.

Object Pascal uses objects to represent exceptions. This has several advantages, the key ones of which are

- Exceptions can be grouped into hierarchies using inheritance
- New exceptions can be introduced without affecting existing code
- An exception object can carry information (such as an error message or an error code) from the point where it was raised to the point where it is handled

Using exception handling

The *SysUtils* unit implements exception generation and handling for the Object Pascal run-time library. When an application uses the *SysUtils* unit, all run-time errors are automatically converted into exceptions. This means that error conditions such as out of memory, division by zero, and general protection fault, which would otherwise terminate an application, can be caught and handled in a structured fashion.

Note Delphi's Visual Class Library fully supports exception handling. An application that uses VCL automatically also uses the *SysUtils* unit, thus enabling exception handling.

Exception declarations

An exception in Object Pascal is simply a class, and the declaration of an exception is no different than the declaration of an ordinary class. Although it is possible to use an instance of any class as an exception object, it is recommended that all exceptions be derived from the *Exception* class defined in the *SysUtils* unit. For example

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

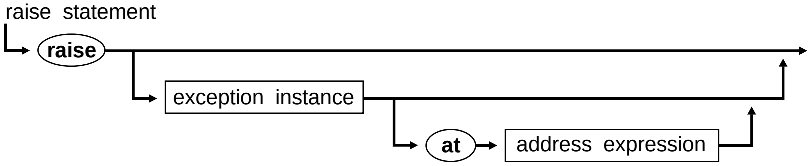
Exceptions are often grouped into families of related exceptions using inheritance. The above declarations (which were extracted from the *SysUtils* unit) define a family of related math error exceptions. Using families, it is possible to handle an entire group of exceptions under one name. For example, an exception handler for *EMathError* will also handle *EInvalidOp*, *EZeroDivide*, *EOverflow*, and *EUnderflow* exceptions, and any user-defined exceptions that directly or indirectly derive from *EMathError*.

Exception classes sometimes define additional fields, methods, and properties used to convey additional information about the exception. For example, the *EInOutError* class defined in the *SysUtils* unit introduces an *ErrorCode* field which contains the file I/O error code that caused the exception.

```
type
  EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

The raise statement

An exception is raised using a **raise** statement.



The argument to a **raise** statement must be an object. In other words, the **raise** keyword must be followed by an expression of a class type. When an exception is raised, the exception handling logic takes ownership of the exception object. Once the exception is handled, the exception object is automatically destroyed through a call to the object's *Destroy* destructor. An application should never attempt to manually destroy a raised exception object.

Note A **raise** statement raises an object, not a class. The argument to a **raise** statement is typically constructed "on the fly" through a call to the *Create* constructor of the appropriate exception class.

A **raise** statement that omits the exception object argument will *re-raise* the current exception. This form of a **raise** statement is allowed only in an *exception block*, and is described further in the section entitled "Re-raising exceptions".

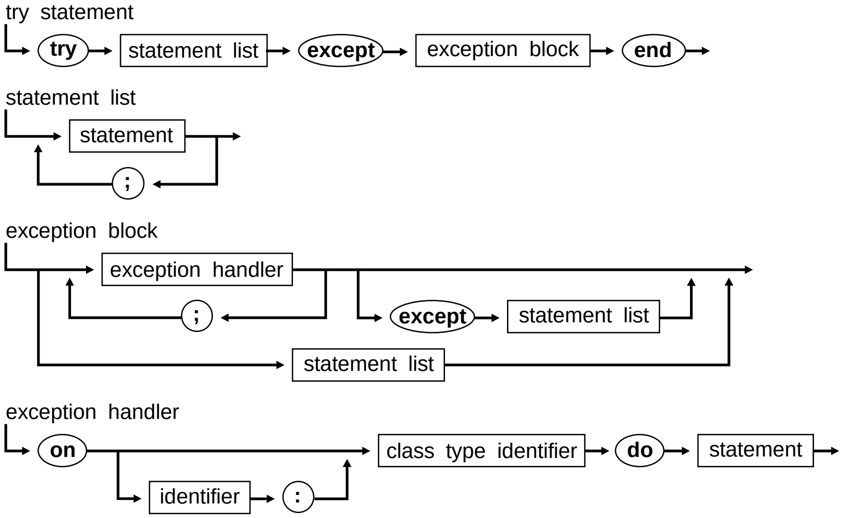
Control *never* returns from a **raise** statement. Instead, **raise** transfers control to the innermost exception handler that can handle exceptions of the given class. Innermost in this case means the handler whose **try**...**except** block was most recently entered and not yet exited.

The *StrToIntRange* function below converts a string to an integer, and raises an *ERangeError* exception if the resulting value is not within a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

The try...except statement

Exceptions are handled using **try**...**except** statements.



A **try**...**except** statement executes the statements in the *try statement list* in sequential order. If the statements execute without any exceptions being raised, the *exception* *block* is ignored, and control is passed to the statement following the **end** keyword that ends the **try**...**except** statement.

The *exception block* in the **except**...**end** section defines exception handlers for the *try statement list*. An exception handler can be invoked only by a **raise** statement executed in the *try statement list* or by a procedure or function called from the *try statement list*.

When an exception is raised, control is transferred to the innermost exception handler that can handle exceptions of the given class. The search for an exception handler starts with the most recently entered and not yet exited **try**...**except** statement. If that **try**...**except** statement cannot handle exceptions of the given class, the next most recently entered **try**...**except** statement is examined. This *propagation* of the exception continues until an appropriate handler is found, or there are no more active **try**...**except** statements. In the latter case a run-time error occurs and the application is terminated.

To determine whether the *exception block* of a **try**...**except** statement can handle a particular exception, the **on**...**do** exception handlers are examined in order of appearance. The first exception handler that lists the exception class or a base class of the exception is considered a match. If an *exception block* contains an **else** part, and if none of the **on**...**do** exception handlers match the exception, the **else** part is considered a match. An *exception block* that contains only a statement list is considered a match for any exception.

Once a matching exception handler is found, the stack is "unwound" to the procedure or function that contains the handler, and control is transferred to the handler. The unwinding process will discard all procedure and function calls that occurred since entering the **try**...**except** statement containing the exception handler.

Following execution of an exception handler, the exception object is automatically destroyed through a call to the object's *Destroy* destructor, and control is passed to the statement following the **end** keyword that ends the **try**...**except** statement.

In the example below, the first **on**...**do** handles division by zero exceptions, the second **on**...**do** handles overflow exceptions, and the final **on**...**do** handles all other math exceptions.

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

As described earlier, to locate a handler for a given exception class, the **on**...**do** handlers are processed in order of appearance. This means that handlers for the most derived classes should be listed first. For example, since *EZeroDivide* and *EOverflow* are both derived from *EMathError*, a handler for *EMathError* will also handle *EZeroDivide* and *EOverflow* exceptions. If *EMathError* was listed before these exceptions, the more specific handlers would never be invoked.

An **on**...**do** exception handler can optionally specify an identifier and a colon before the exception class identifier. This declares the identifier to represent the exception object during execution of the statement that follows **on**...**do**. For example

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

The scope of an identifier declared in an exception handler is the statement that follows **on**...**do**. The identifier hides any similarly named identifier in an outer scope.

If a **try**...**except** statement specifies an **else** part, then any exceptions that aren't handled by the **on**...**do** exception handlers will be handled by the **else** part. For example

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the **else** part will handle any exception that isn't an *EMathError*.

An *exception block* that contains no **on**...**do** handlers, but instead consists only of a list of statements, will handle all exceptions.

```
try
  ...
except
  HandleException;
end;
```

Here, any exception that occurs as a result of executing the statements between **try** and **except** will be handled by the *HandleException* procedure.

Re-raising exceptions

In some situations, a procedure or function may need to perform clean-up operations when an exception occurs, but the procedure or function may not be prepared to actually handle the exception. For example, consider the *GetFileList* function below, which allocates a *TStringList* object and fills it with the filenames matching a given search path.

```
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;
```

The function first allocates a new *TStringList* object, and then uses the *FindFirst* and *FindNext* functions (defined in the *SysUtils* unit) to initialize the string list. If, for any reason, the initialization of the string list fails, for exampe because the given search path is invalid, or because there is not enough memory to fill in the string list, *GetFileList* needs to dispose the newly allocated string list, since the caller does not yet know of its existence. For that reason, the initialization of the string list is performed in a **try**...**except** statement. If an exception occurs, the exception handler part of the **try**...**except** statement disposes the string list, and then *re-raises* the exception.

Nested exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as exceptions raised in an exception handler are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the *Tan* function shown below.

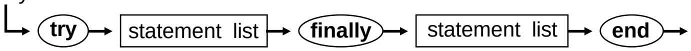
```
type
  ETrigError = class(EMathError);

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;
```

If an *EMathError* is raised in the computation of the tangent of the given angle, such as the *EZeroDivide* that would result if *Cos*(*X*) is zero, the exception handler raises an *ETrigError* exception. Since the *Tan* function does not provide a handler for the *ETrigError* exception that it raises, the *ETrigError* will propagate beyond the original exception handler, causing the *EMathError* exception to be disposed. To the caller, the result is simply that the *Tan* function has raised an *ETrigError* exception.

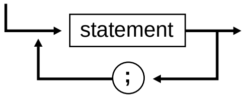
The try...finally statement

When a section of code acquires a resource, it is often necessary to ensure that the resource be released again, regardless of whether the code completes normally or is interrupted by an exception. For example, a section of code that opens and processes a file will normally want to ensure that the file is closed no matter how the code terminates. The **try**...**finally** statement can be used in such situations.

try statement



statement list



A **try**...**finally** statement executes the statements in the *try statement list* in sequential order. If no exceptions are raised in the *try statement list*, the *finally statement list* is executed. If an exception is raised in the *try statement list*, control is transferred to the *finally statement list*, and once the *finally statement list* completes execution, the exception is re-raised. The resulting effect is that the *finally statement list* is always executed, regardless of how the *try statement list* terminates.

The section of code shown below illustrates how a **try**...**finally** statement can be used to ensure that a file that was opened is always closed.

```
Reset(F);
try
  ProcessFile(F);
finally
  CloseFile(F);
end;
```

Note In a typical application, **try...finally** statements such as the one above tend to be much more common than **try...except** statements. Applications written using Delphi's Visual Class Library, for example, normally rely on VCL's default exception handling mechanisms, thus seldom needing to use **try...except** statements. Resource allocations will however often have to be guarded against exceptions, and the use of **try...finally** statements will therefore be much more frequent.

If an exception is raised but not handled in the *finally statement list*, that exception is propagated out of the **try**...**finally** statement, and any original exception is lost.

Note It is strongly recommended that a *finally statement list* always handle all local exceptions, so as to not disturb the propagation of an external exception.

Exit, Break, and Continue procedures

If a call to one of the *Exit*, *Break*, or *Continue* standard procedures cause control to leave the *try statement list* of a **try**...**finally** statement, the *finally statement list* is automatically executed. Likewise, if one of these standard procedures are used to leave an exception handler, the exception object is automatically disposed.

Predefined exceptions

The *SysUtils* unit declares a number of exception classes, including a class named *Exception* which serves as the ultimate ancestor of all exception classes. The public interface of *Exception* is declared as follows

```
type
  Exception = class(TObject)
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string; const Args: array of const);
    constructor CreateRes(Ident: Word);
    constructor CreateResFmt(Ident: Word; const Args: array of const);
    constructor CreateHelp(const Msg: string; HelpContext: Longint);
    constructor CreateFmtHelp(const Msg: string; const Args: array of const;
      HelpContext: Longint);
    constructor CreateResHelp(Ident: Word; HelpContext: Longint);
    constructor CreateResFmtHelp(Ident: Word; const Args: array of const;
      HelpContext: Longint);
    destructor Destroy; override;
    property HelpContext: Longint;
    property Message: string;
  end;
```

The *Exception* class establishes two properties, *Message* and *HelpContext*, that every exception object inherits. This means that an arbitrary exception object can at least provide a descriptive message of the exception condition, and possibly a help context that refers to further on-line help on the topic.

The constructors defined by *Exception* provide various ways of initializing the *Message* and *HelpContext* properties. In general

- Constructors that don't include "Res" in their names require the exception message to be specified as a string parameter. Those that do include "Res" in their names will initialize the *Message* property from the string resource with the given ID.
- Constructors that include "Fmt" in their names interpret the specified exception message as a format string, and require an extra *Args* parameter which supplies the format arguments. The initial value of the *Message* property is constructed through a call to the *Format* function in the *SysUtils* unit.
- Constructors that include "Help" in their names require an extra *HelpContext* parameter which supplies the on-line help context for the exception.

The following table lists all exceptions defined by the *SysUtils* unit, and the situations in which the exceptions are raised. Unless otherwise noted, the exceptions are direct descendants of the *Exception* class.

Table 10-1 Predefined exception classes

Exception class	Description
EAbort	The "silent exception" raised by the *Abort* procedure.
EOutOfMemory	Raised if there is not enough memory for a particular operation.
EInOutError	Raised if a file I/O operation causes an error. The *EInOutError* exception defines an *ErrorCode* field that contains the I/O error code, corresponding to the value returned by the *IOResult* standard function.
EIntError	The ancestor class for all integer arithmetic exceptions.
EDivByZero	Derived from *EIntError*. Raised if an integer divide operation with a zero divisor is attempted.
ERangeError	Derived from *EIntError*. Raised if a range check operation fails in a section of code that was compiled in the {$**R**+} state.
EIntOverflow	Derived from *EIntError*. Raised if an overflow check operation fails in a section of code that was compiled in the {$**Q**+} state.
EMathError	The ancestor class for all floating-point math exceptions.
EInvalidOp	Derived from *EMathError*. Raised if an invalid math operation is performed, such as taking the square root of a negative number.
EZeroDivide	Derived from *EMathError*. Raised if a divide operation with a zero divisor is attempted.
EOverflow	Derived from *EMathError*. Raised if a floating-point operation produces an overflow.
EUnderflow	Derived from *EMathError*. By default, floating-point operations that underflow simply produce a zero result. An application must manually change the control word of the 80x87 co-processor to enable underflow exceptions.
EInvalidPointer	Raised if an application attempts to free an invalid pointer.
EInvalidCast	Raised if the object given on the left hand side of an **as** operator is not of the class given on the right hand side of the operator.
EConvertError	Raised if a conversion function cannot perform the required conversion. A number of functions in the *SysUtils* unit, such as *StrToInt*, *StrToFloat*, and *StrToDateTime*, may raise this exception.
EProcessorException	The ancestor class for all hardware exceptions.
EFault	Derived from *EProcessorException*. The ancestor class for all processor fault exceptions.
EGPFault	Derived from *EFault*. Raised if an application tries to access an invalid memory address. This exception typically indicates that the application tried to access an object through an uninitialized object reference, or tried to dereference an uninitialized pointer.
EStackFault	Derived from *EFault*. Raised if there is not enough stack space to allocate the local variables for a procedure or function. Stack overflow checking must be enabled, using a {$**S**+} directive, for this exception to occur.
EPageFault	Derived from *EFault*. Raised if the CPU reports a page fault.
EInvalidOpCode	Derived from *EFault*. Raised if the CPU detects an invalid instruction.
EBreakpoint	Derived from *EProcessorException*. Raised if the CPU encounters a breakpoint instruction.
ESingleStep	Derived from *EProcessorException*. Raised after the CPU has executed an instruction in single-step mode.

Exception handling support routines

The *SysUtils* unit defines a number of exception handling support routines. A brief description of each is presented here. For further information, see the *Visual Component Library Reference.*

Table 10-2 Exception support routines

Routine	Description
ExceptObject	Returns a reference to the current exception object, that is the object associated with the currently raised exception. If there is no current exception, *ExceptObject* returns **nil**.
ExceptAddr	Returns the address at which the current exception was raised. If there is no current exception, *ExceptAddr* returns **nil**.
ShowException	Displays an exception dialog box for a given exception object and exception address.
Abort	Raises an *EAbort* exception. VCL's standard exception handler treats *EAbort* as a "silent exception", and does not display an exception dialog box when it is handled.
OutOfMemoryError	Raises an *EOutOfMemory* error. *OutOfMemoryError* uses a pre-allocated *EOutOfMemory* exception object, thus avoiding any dynamic memory allocations as part of raising the exception.

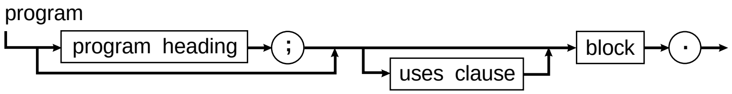
Chapter

11

Programs and units

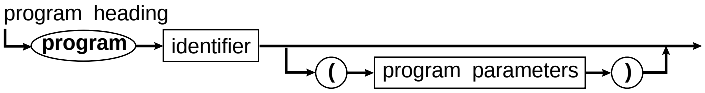
Program syntax

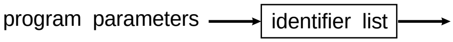
A Object Pascal program consists of a program heading, an optional **uses** clause, and a block.



The program heading

The program heading specifies the program's name and its parameters.

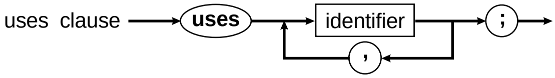




The program heading, if present, is ignored by the compiler.

The uses clause

The **uses** clause identifies all units used by the program.



The *System* unit is always used automatically. *System* implements all low-level, run-time routines to support such features as file input and output (I/O), string handling, floating point, dynamic memory allocation, and others.

Apart from *System*, Object Pascal implements many standard units that aren't used automatically; you must include them in your **uses** clause. For example,

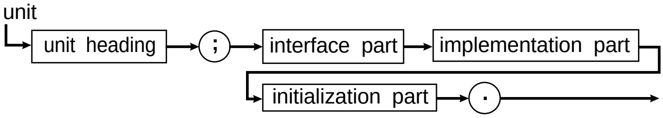
```
uses SysUtils;                    { Can now use SysUtils }
```

The order of the units listed in the **uses** clause determines the order of their initialization (see "The initialization part" on page 125).

Note To find the unit file containing a compiled unit, the compiler truncates the unit name listed in the **uses** clause to the first eight characters and adds the file extension. The file extension for Delphi units is .DCU.

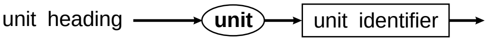
Unit syntax

Units are the basis of in Object Pascal. They're used to create libraries you can include in various programs without making the source code available, and to divide large programs into logically related modules.



The unit heading

The unit heading specifies the unit's name.

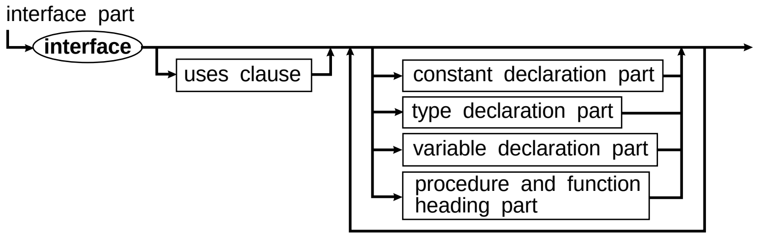


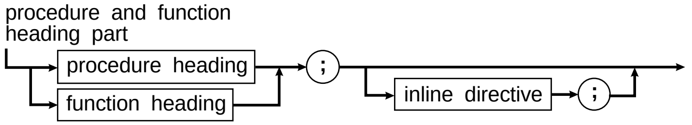
The unit name is used when referring to the unit in a **uses** clause. The name must be unique: Two units with the same name can't be used at the same time.

Note The name of a unit's source file and binary file must be the same as the unit identifier, truncated to the first eight characters. If this isn't the case, the compiler can't find the source and/or binary file when compiling a program or unit that uses the unit.

The interface part

The interface part declares constants, types, variables, procedures, and functions that are *public*; that is, available to the host (the program or unit using the unit). The host can access these entities as if they were declared in a block that encloses the host.

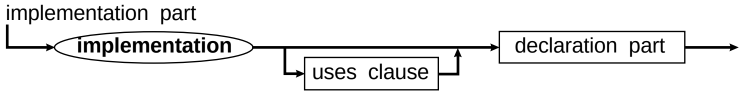

interface part
interface
uses clause
constant declaration part
type declaration part
variable declaration part
procedure and function heading part



Unless a procedure or function is **inline**, the interface part only lists the procedure or function heading. The block of the procedure or function follows in the implementation part.

The implementation part

The implementation part defines the block of all public procedures and functions. In addition, it declares constants, types, variables, procedures, and functions that are *private*; that is, not available to the host.

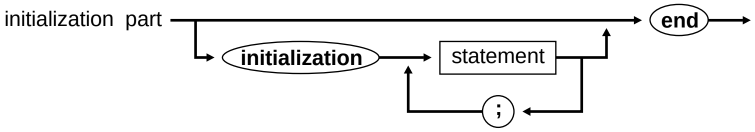


In effect, the procedure and function declarations in the interface part are like forward declarations, although the **forward** directive isn't specified. Therefore, these procedures and functions can be defined and referenced in any sequence in the implementation part.

Note Procedure and function headings can be duplicated from the interface part. You don't have to specify the formal parameter list, but if you do, the compiler will issue a compile-time error if the interface and implementation declarations don't match.

The initialization part

The initialization part is the last part of a unit. It consists either of the reserved word **end** (in which case the unit has no initialization code) or of the reserved word **initialization**, followed by a list of statements which initialize the unit, followed by the reserved word **end**.



The initialization parts of units used by a program are executed in the same order that the units appear in the **uses** clause.

Indirect unit references

The **uses** clause in a module (program or unit) need only name the units used directly by that module. Consider the following:

```
program Prog;
uses Unit2;
const a = b;
begin
end.

unit Unit2;
interface
uses Unit1;
const b = c;
implementation
end.

unit Unit1;
interface
const c = 1;
implementation
const d = 2;
end.
```

In the previous example, *Unit2* is directly dependent on *Unit1* and *Prog* is directly dependent on *Unit2*. Also, *Prog* is indirectly dependent on *Unit1* (through *Unit2*), even though none of the identifiers declared in *Unit1* are available to *Prog*.

To compile a module, the compiler must be able to locate all units the module depends upon, directly or indirectly. So, to compile *Prog*, the compiler must be able to locate both *Unit1* and *Unit2*, or else an error occurs.

Note for C and other language users: The **uses** clauses of an Object Pascal program provide the "make" logic information traditionally found in make or project files of other languages. With the uses clause, Object Pascal can build all the dependency information into the module itself and reduce the chance for error.

When changes are made in the interface part of a unit, other units using the unit must be recompiled. If you use Make or Build, the compiler does this for you automatically. If changes are made only to the implementation or the initialization part, however, other units that use the unit *don't* have to be recompiled. In the previous example, if the interface part of *Unit1* is changed (for example, *c* = 2) *Unit2* must be recompiled; changing the implementation part (for example, *d* = 1) doesn't require recompilation of *Unit2*.

Object Pascal can tell when the interface part of a unit has changed by computing a *unit version number* when the unit is compiled. In the preceding example, when *Unit2* is compiled, the current version number of *Unit1* is saved in the compiled version of

Unit2. When *Prog* is compiled, the version number of *Unit1* is checked against the version number stored in *Unit2*. If the version numbers don't match (indicating that a change was made in the interface part of *Unit1* because *Unit2* was compiled), the compiler reports an error or recompiles *Unit2*, depending on the mode of compilation.

Circular unit references

When two or more units reference each other in their **uses** clauses, the units are said to be mutually dependent. Any pattern of mutual dependencies is allowed as long as there are no circular references created by following the **uses** clauses in the **interface** parts of the units. In other words, for any given set of mutually dependent units, starting with the **uses** clause of the **interface** part of any unit it must not be possible to return to that unit by following references in the **uses** clauses of the **interface** parts. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the **uses** clause of the **implementation** part of at least one unit.

In the simplest case of two mutually dependent units, the implication of the above rule is that it is not possible for both units to reference the other in the **interface** part. It is however possible for one of the units to reference the other in the **interface** part, as long as the other references the first in the **implementation** part.

When the compiler detects an invalid circular reference path, it reports error 68, "Circular unit reference". In order to reduce the chances of invalid circular references occurring, it is recommended that a unit use other units in the **implementation** part whenever possible. Only when symbols from another unit are needed in the **interface** part should that unit be listed in the **interface** part **uses** clause.

Chapter

12

Dynamic-link libraries

Dynamic-link libraries (DLLs) permit several Windows applications to share code and resources. With Object Pascal, you can use DLLs as well as write your own DLLs to be used by other applications.

What is a DLL?

A DLL is an executable module containing code or resources for use by other applications or DLLs. Conceptually, a DLL is similar to a unit—both have the ability to provide services in the form of procedures and functions to a program. There are, however, many differences between DLLs and units. In particular, units are *statically linked*, whereas DLLs are *dynamically linked*.

When a program uses a procedure or function from a unit, a copy of that procedure or function's code is statically linked into the program's executable file. If two programs are running simultaneously and they use the same procedure or function from a unit, there will be two copies of that routine present in the system. It would be more efficient if the two programs could share a single copy of the routine. DLLs provide that ability.

In contrast to a unit, the code in a DLL isn't linked into a program that uses the DLL. Instead, a DLL's code and resources are in a separate executable file with a .DLL extension. This file must be present when the client program runs. The procedure and function calls in the program are dynamically linked to their entry points in the DLLs used by the application.

Another difference between units and DLLs is that units can export types, constants, data, and objects whereas DLLs can export procedures and functions only.

Note A DLL doesn't have to be written in Object Pascal for a Object Pascal application to be able to use it. Also, programs written in other languages can use DLLs written in Object Pascal. DLLs are therefore ideal for multi-language programming projects.

Using DLLs

For a module to use a procedure or function in a DLL, the module must import the procedure or function using an external declaration. For example, the following external declaration imports a function called *GlobalAlloc* from the DLL called KERNEL (the Windows kernel):

```
function GlobalAlloc(Flags: Word; Bytes: Longint): THandle; far;
external 'KERNEL' index 15;
```

In imported procedures and functions, the **external** directive takes the place of the declaration and statement parts that would otherwise be present. Imported procedures and functions must use the far call model selected by using a **far** procedure directive or a {**$F+**} compiler directive, but otherwise they behave no differently than normal procedures and functions. See "External declarations" on page 72.

Object Pascal imports procedures and functions in three ways:

- By name
- By new name
- By ordinal

The format of **external** directives for each of the three methods is demonstrated in the following examples.

When no **index** or **name** clause is specified, the procedure or function is imported by name. The name used is the same as the procedure or function's identifier. In this example, the *ImportByName* procedure is imported from 'TESTLIB' using the name 'IMPORTBYNAME'.

```
procedure ImportByName; external 'TESTLIB';
```

When a **name** clause is specified, the procedure or function is imported by a different name than its identifier. Here the *ImportByNewName* procedure is imported from 'TESTLIB' using the name 'REALNAME':

```
procedure ImportByNewName; external 'TESTLIB' name 'REALNAME';
```

Finally, when an **index** clause is present, the procedure or function is imported by ordinal. Importing by ordinal reduces the module's load time because the name doesn't have to be looked up in the DLL's name table. In the example, the *ImportByOrdinal* procedure is imported as the fifth entry in the 'TESTLIB' DLL:

```
procedure ImportByOrdinal; external 'TESTLIB' index 5;
```

The DLL name specified after the **external** keyword and the new name specified in a **name** clause don't have to be string literals. Any constant-string expression is allowed. Likewise, the ordinal number specified in an **index** clause can be any constant-integer expression.

```
const
  TestLib = 'TESTLIB';
  Ordinal = 5;
```

```
procedure ImportByName;    external TestLib;
procedure ImportByNewName; external TestLib name 'REALNAME';
procedure ImportByOrdinal; external TestLib index Ordinal;
```

Although a DLL can have variables, it's not possible to import them in other modules. Any access to a DLL's variables must take place through a procedural interface.

Import units

Declarations of imported procedures and functions can be placed directly in the program that imports them. Usually, though, they are grouped together in an *import unit* that contains declarations for all procedures and functions in a DLL, along with any constants and types required to interface with the DLL. The *WinTypes* and *WinProcs* units supplied with Delphi are examples of such import units. Import units aren't a requirement of the DLL interface, but they do simplify maintenance of projects that use multiple DLLs.

As an example, consider a DLL called DATETIME.DLL that has four routines to get and set the date and time, using a record type that contains the day, month, and year, and another record type that contains the second, minute, and hour. Instead of specifying the associated procedure, function, and type declarations in every program that uses the DLL, you can construct an import unit to go along with the DLL. This code creates a .DCU file, but it doesn't contribute code or data to the programs that use it:

```
unit DateTime;

interface

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

type
  TDateRec = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

procedure SetTime(var Time: TTimeRec);
procedure GetTime(var Time: TTimeRec);
procedure SetDate(var Date: TDateRec);
procedure GetDate(var Date: TDateRec);

implementation
```

```
procedure SetTime; external 'DATETIME' index 1;
procedure GetTime; external 'DATETIME' index 2;
procedure SetDate; external 'DATETIME' index 3;
procedure GetDate; external 'DATETIME' index 4;

end.
```

Any program that uses DATETIME.DLL can now simply specify *DateTime* in its **uses** clause. Here is a Windows program example:

```
program ShowTime;

uses WinCrt, DateTime;

var
  Time: TTimeRec;

begin
  GetTime(Time);
  with Time do
    WriteLn('The time is ', Hour, ':', Minute, ':', Second);
end.
```

Another advantage of an import unit such as *DateTime* is that when the associated DATETIME.DLL is modified, only one unit, the *DateTime* import unit, needs updating to reflect the changes.

When you compile a program that uses a DLL, the compiler doesn't look for the DLL so it need not be present. The DLL must be present when you run the program, however.

Note If you write your own DLLs, they aren't automatically compiled when you compile a program that uses the DLL. Instead, DLLs must be compiled separately.

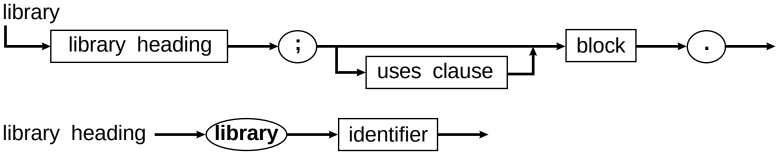
Static and dynamic imports

The **external** directive provides the ability to statically import procedures and functions from a DLL. A statically-imported procedure or function always refers to the same entry point in the same DLL. Windows also supports dynamic imports, whereby the DLL name and the name or ordinal number of the imported procedure or function is specified at run time. The *ShowTime* program shown here uses dynamic importing to call the *GetTime* procedure in DATETIME.DLL. Note the use of a procedural-type variable to represent the address of the *GetTime* procedure.

```
program ShowTime;

uses WinProcs, WinTypes, WinCrt;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
  TGetTime = procedure(var Time: TTimeRec);

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;

begin
  Handle := LoadLibrary('DATETIME.DLL');
  if Handle >= 32 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;
```

Writing DLLs

The structure of a Object Pascal DLL is identical to that of a program, except a DLL starts with a **library** header instead of a **program** header. The **library** header tells Object Pascal to produce an executable file with the extension .DLL instead of .EXE, and also ensures that the executable file is marked as being a DLL.



The example here implements a very simple DLL with two exported functions, *Min* and *Max*, that calculate the smaller and larger of two integer values.

```
library MinMax;

function Min(X, Y: Integer): Integer; export;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; export;
begin
```

```
  if X > Y then Max := X else Max := Y;
end;

exports
  Min index 1,
  Max index 2;

begin
end.
```

Note the use of the **export** procedure directive to prepare *Min* and *Max* for exporting, and the **exports** clause to actually export the two routines, supplying an optional ordinal number for each of them.

Although the preceding example doesn't demonstrate it, libraries can and often do consist of several units. In such cases, the library source file itself is frequently reduced to a **uses** clause, an **exports** clause, and the library's initialization code. For example,

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors index 1,
  DoneEditors index 2,
  InsertText index 3,
  DeleteSelection index 4,
  FormatSelection index 5,
  PrintSelection index 6,
   ⋮
  SetErrorHandler index 53;

begin
  InitLibrary;
end.
```

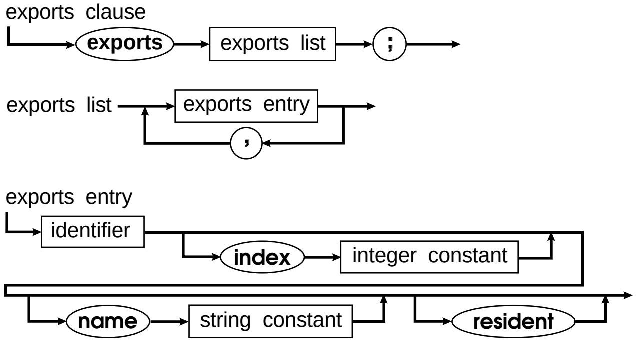
The export procedure directive

If procedures and functions are to be exported by a DLL, they must be compiled with the **export** procedure directive. The **export** directive belongs to the same family of procedure directives as the **near**, **far**, and **inline** directives. This means that an **export** directive, if present, must be specified upon the first introduction of procedure or function—it can't be supplied in the defining declaration of a forward declaration.

The **export** directive makes a procedure or function exportable. It forces the routine to use the far call model and prepares the routine for export by generating special procedure entry and exit code. Note, however, that the actual exporting of the procedure or function doesn't occur until the routine is listed in a library's **exports** clause. See "Export declarations" on page 71.

The exports clause

A procedure or function is exported by a DLL when it's listed in the library's **exports** clause.



An **exports** clause can appear anywhere and any number of times in a program or library's declaration part. Each entry in an **exports** clause specifies the identifier of a procedure or function to be exported. That procedure or function must be declared before the **exports** clause appears, however, and its declaration must contain the **export** directive. You can precede the identifier in the **exports** clause with a unit identifier and a period; this is known as a fully qualified identifier.

An exports entry can also include an **index** clause, which consists of the word **index** followed by an integer constant between 1 and 32,767. When an **index** clause is specified, the procedure or function to be exported uses the specified ordinal number. If no **index** clause is present in an exports entry, an ordinal number is automatically assigned. The quickest way to look up a DLL entry is by index.

An entry can also have a **name** clause, which consists of the word **name** followed by a string constant. When there is a **name** clause, the procedure or function to be exported uses the name specified by the string constant. If no **name** clause is present in an exports entry, the procedure or function is exported by its identifier and converted to all uppercase.

Finally, an exports entry can include the **resident** keyword. When **resident** is specified, the export information stays in memory while the DLL is loaded. The **resident** option significantly reduces the time it takes to look up a DLL entry by name, so if client programs that use the DLL are likely to import certain entries by name, those entries should be exported using the **resident** keyword.

A program can contain an **exports** clause, but it seldom does because Windows doesn't allow application modules to export functions for use by other applications.

Library initialization code

The statement part of a library constitutes the library's *initialization code*. The initialization code is executed once, when the library is initially loaded. When subsequent applications that use the library are loaded, the initialization code isn't executed again, but the DLL's use count is incremented.

A DLL is kept in memory as long as its use count is greater than zero. When the use count becomes zero, indicating that all applications that used the DLL have terminated, the DLL is removed from memory. At that point, the library's exit procedures are executed. Exit procedures are registered using the *ExitProc* variable, as described in Chapter 17, "Control issues."

A DLL's initialization code typically performs tasks like registering window classes for window procedures contained in the DLL and setting initial values for the DLL's global variables. The initialization code of a library can signal an error condition by setting the *ExitCode* variable to zero. (*ExitCode* is declared by the *System* unit.) *ExitCode* defaults to 1, indicating initialization was successful. If the initialization code sets *ExitCode* to zero, the DLL is unloaded from system memory and the calling application is notified of the failure to load the DLL.

When a library's exit procedures are executed, the *ExitCode* variable doesn't contain a process-termination code, as is the case with a program. Instead, *ExitCode* contains one of the values *wep_System_Exit* or *wep_Free_DLL*, which are defined in the *WinTypes* unit. *wep_System_Exit* indicates that Windows is shutting down, whereas *wep_Free_DLL* indicates that just this single DLL is being unloaded.

Here is an example of a library with initialization code and an exit procedure:

```
library Test;{$S-}

uses WinTypes, WinProcs;

var
  SaveExit: Pointer;

procedure LibExit; far;
begin
  if ExitCode = wep_System_Exit then
  begin
     ⋮
    { System shutdown in progress }
     ⋮
  end else
  begin
     ⋮
    { DLL is being unloaded }
     ⋮
  end;
  ExitProc := SaveExit;
end;

begin
   ⋮
  { Perform DLL initialization }
   ⋮
  SaveExit := ExitProc;      { Save old exit procedure pointer }
  ExitProc := @LibExit;      { Install LibExit exit procedure }
end.
```

When a DLL is unloaded, an exported function called WEP in the DLL is called, if it's present. A Object Pascal library automatically exports a WEP function, which continues to call the address stored in the *ExitProc* variable until *ExitProc* becomes **nil**. Because this works the same way exit procedures are handled in Object Pascal programs, you can use the same exit procedure logic in both programs and libraries.

Note Exit procedures in a DLL *must* be compiled with stack-checking disabled (the {**$S-**} state) because the operating system switches to an internal stack when terminating a DLL. Also, the operating system crashes if a run-time error occurs in a DLL exit procedure, so you must include sufficient checks in your code to prevent run-time errors.

Library programming notes

The following sections note important points you should keep in mind while working with DLLs.

Global variables in a DLL

A DLL has its own data segment and any variables declared in a DLL are private to that DLL. A DLL can't access variables declared by modules that call the DLL, and it's not possible for a DLL to export its variables for use by other modules. Such access must take place through a procedural interface.

Global memory and files in a DLL

As a rule, a DLL doesn't "own" any files that it opens or any global memory blocks that it allocates from the system. Such objects are owned by the application that (directly or indirectly) called the DLL.

When an application terminates, any open files owned by it are automatically closed, and any global memory blocks owned by it are automatically deallocated. This means that file and global memory-block handles stored in global variables in a DLL can become invalid at any time without the DLL being notified. For that reason, DLLs should refrain from making assumptions about the validity of file and global memory-block handles stored in global variables across calls to the DLL. Instead, such handles should be made parameters of the procedures and functions of the DLL, and the calling application should be responsible for maintaining them.

Note Global memory blocks allocated with the *gmem_DDEShare* attribute (defined in the *WinTypes* unit) are owned by the DLL, not by the calling applications. Such memory blocks remain allocated until they are explicitly deallocated by the DLL, or until the DLL is unloaded.

DLLs and the System unit

During a DLL's lifetime, the *HInstance* variable contains the instance handle of the DLL. The *HPrevInst* and *CmdShow* variables are always zero in a DLL, as is the

PrefixSeg variable, because a DLL doesn't have a Program Segment Prefix (PSP). *PrefixSeg* is never zero in an application, so the test *PrefixSeg* <> 0 returns *True* if the current module is an application, and *False* if the current module is a DLL.

To ensure proper operation of the heap manager contained in the *System* unit, the start-up code of a library sets the *HeapAllocFlags* variable to *gmem_Moveable* + *gmem_DDEShare*. Under Windows, this causes all memory blocks allocated via *New* and *GetMem* to be owned by the DLL instead of the applications that call the DLL.

For details about the heap manager, see Chapter 16.

Run-time errors in DLLs

If a run-time error occurs in a DLL, the application that called the DLL terminates. The DLL itself isn't necessarily removed from memory at that time because other applications might still be using it.

Because a DLL has no way of knowing whether it was called from a Object Pascal application or an application written in another programming language, it's not possible for the DLL to invoke the application's exit procedures before the application is terminated. The application is simply aborted and removed from memory. For this reason, make sure there are sufficient checks in any DLL code so such errors don't occur.

If a run-time error does occur in a DLL, the safest thing to do is to exit Windows entirely. If you simply try to modify and rebuild the faulty DLL code, when you run your program again, Windows won't load the new version if the buggy one is still in memory. Exiting Windows and then restarting Windows and Object Pascal ensures that your corrected version of the DLL is loaded.

DLLs and stack segments

Unlike an application, a DLL doesn't have its own stack segment. Instead, it uses the stack segment of the application that called the DLL. This can create problems in DLL routines that assume that the DS and SS registers refer to the same segment, which is the case in a Windows application module.

The Object Pascal compiler never generates code that assumes DS = SS, and none of the Object Pascal run-time library routines make this assumption. If you write assembly language code, don't assume that SS and DS registers contain the same value.

Chapter

13

Input and output

This chapter describes Delphi's standard (or built-in) input and output () procedures and functions. You'll find them in the *System* unit.

Table 13-1 Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
Readln	Does what a *Read* does and then skips to the beginning of the next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.

SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.
Writeln	Does the same as a *Write*, and then writes an end-of-line marker to the text file.

File input and output

An Object Pascal file variable is any variable whose type is a file type. There are three classes of Object Pascal files: *typed*, *text*, and *untyped*. The syntax for writing file types is given on page 21.

Before a file variable can be used, it must be associated with an external file through a call to the *AssignFile* procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be "opened" to prepare it for input or output. An existing file can be opened via the *Reset* procedure, and a new file can be created and opened via the *Rewrite* procedure. Text files opened with *Reset* are read-only and text files opened with *Rewrite* and *Append* are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. Each component has a component number. The first component of a file is considered to be component zero.

Files are normally accessed *sequentially*; that is, when a component is read using the standard procedure *Read* or written using the standard procedure *Write*, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly, however, using the standard procedure *Seek*, which moves the current file position to a specified component. The standard functions *FilePos* and *FileSize* can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure *CloseFile*. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the {**$I+**} and {**$I-**} compiler directives. When I/O checking is off--that is, when a procedure or function call is compiled in the {**$I-**} state--an I/O error doesn't cause an exception to be raised. To check the result of an I/O operation, you must call the standard function *IOResult* instead.

You must call the *IOResult* function to clear whatever error may have occurred, even if you aren't interested in the error. If you don't and {**$I+**} is the current state, the next I/O function call fails with the lingering *IOResult* error.

Text files

This section summarizes I/O using file variables of the standard type *Text*.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a linefeed character). In Object Pascal, the type *Text* is distinct from the type **file of** *Char*.

For text files, there are special forms of *Read* and *Write* that let you read and write values that are not of type *Char*. Such values are automatically translated to and from their character representation. For example, *Read(F, I)*, where *I* is a type *Integer* variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in *I*.

Object Pascal defines two standard text-file variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the operating system's standard input file (typically the keyboard). The standard file variable *Output* is a write-only file associated with the operating system's standard output file (typically the display):

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Because Windows doesn't directly support text-oriented I/O, the I/O files are unassigned in a Windows application, and any attempt to read or write to them will produce an I/O error. If a Windows application uses the *WinCrt* unit, however, *Input* and *Output* refer to a scrollable text window. *WinCrt* contains the complete control logic required to emulate a text screen in the Windows environment, and no Windows-specific programming is required in an application that uses *WinCrt*. See page 142 for more about the *WinCrt* unit.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, *Read(X)* corresponds to *Read(Input, X)* and *Write(X)* corresponds to *Write(Output, X)*.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using *AssignFile*, and opened using *Reset*, *Rewrite*, or *Append*. An exception is raised if you pass a file that was opened with *Reset* to an output-oriented procedure or function. An exception is also raised if you pass a file that was opened with *Rewrite* or *Append* to an input-oriented procedure or function.

Untyped files

Untyped files are low-level I/O channels primarily used for direct access to any disk file regardless of type and structuring. An untyped file is declared with the word **file** and nothing more. For example,

```
var
  DataFile: file;
```

For untyped files, the *Reset* and *Rewrite* procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file (no partial records are possible when the record size is 1).

Except for *Read* and *Write*, all typed-file standard procedures and functions are also allowed on untyped files. Instead of *Read* and *Write*, two procedures called *BlockRead* and *BlockWrite* are used for high-speed data transfers.

Input and output with the WinCrt unit

The *WinCrt* unit implements a terminal-like text screen in a window. With *WinCrt*, you can easily create a Windows program that uses the *Read*, *Readln*, *Write*, and *Writeln* standard procedures to perform input and output operations. *WinCrt* contains the complete control logic required to emulate a text screen in the Windows environment. You don't need to write "Windows-specific" code if your program uses *WinCrt*.

Using the WinCrt unit

To use the *WinCrt* unit, simply include it in your program's **uses** clause, just as you would any other unit:

```
uses WinCrt;
```

By default, the *Input* and *Output* standard text files defined in the *System* unit are unassigned, and any *Read*, *Readln*, *Write*, or *Writeln* procedure call without a file variable causes a run-time error. But when a program uses the *WinCrt* unit, the initialization code of the unit assigns the *Input* and *Output* standard text files to refer to a window that emulates a text screen. It's as if the following statements are executed at the beginning of your program:

```
AssignCrt(Input); Reset(Input);
AssignCrt(Output); Rewrite(Output);
```

When the first *Read*, *Readln*, *Write*, or *Writeln* call executes in the program, a CRT window opens on the Windows desktop. The default title of a CRT window is the full path of the program's .EXE file. When the program finishes (when control reaches the final **end** reserved word), the title of the CRT window is changed to "(Inactive *nnnnn*)", where *nnnnn* is the title of the window in its active state.

Even though the program has finished, the window stays up so that the user can examine the program's output. Just like any other Windows application, the program doesn't completely terminate until the user closes the window.

The *InitWinCrt* and *DoneWinCrt* routines give you greater control over the CRT window's life cycle. A call to *InitWinCrt* immediately creates the CRT window rather than waiting for the first call to *Read*, *Readln*, *Write*, or *Writeln*. Likewise, calling *DoneWinCrt* immediately destroys the CRT window instead of when the user closes it.

The CRT window is a scrollable panning window on a virtual text screen. The default dimensions of the virtual text screen are 80 columns by 25 lines, but the actual size of the CRT window may be less. If the size is less, the user can use the window's scroll bars or the cursor keys to move this panning window over the larger text screen. This is particularly useful for scrolling back to examine previously written text. By default, the panning window tracks the text screen cursor. In other words, the panning window automatically scrolls to ensure that the cursor is always visible. You can disable the autotracking feature by setting the *AutoTracking* variable to *False*.

The dimensions of the virtual text screen are determined by the *ScreenSize* variable. You can change the virtual screen dimensions by assigning new dimensions to *ScreenSize* before your program creates the CRT window. When the window is created, a screen buffer is allocated in dynamic memory. The size of this buffer is *ScreenSize.X* multiplied by *ScreenSize.Y*, and it can't be larger than 65,520 bytes. It's up to you to ensure that the values you assign to *ScreenSize.X* and *ScreenSize.Y* don't overflow this limit. If, for example, you assign 64 to *ScreenSize.X*, the largest allowable value for *ScreenSize.Y* is 1,023.

At any time while running a program that uses the *WinCrt* unit, the user can terminate the application by choosing the Close command on the CRT window's Control menu, double-clicking the Control-menu box, or pressing Alt+F4. The user can also press Ctrl+C or Ctrl+Break at any time to halt the application and force the window into its inactive state; you can disable these features by setting the *CheckBreak* variable to *False* at the beginning of the program.

Special characters

When writing to *Output* or a file that has been assigned to the CRT window, the following control characters have special meanings:

Table 13-2 Special characters in the WinCrt window

Char	Name	Description
#7	BELL	Emits a beep from the internal speaker.
#8	BS	Moves the cursor left one column and erases the character at that position. If the cursor is already at the left edge of the screen, nothing happens.
#10	LF	Moves the cursor down one line. If the cursor is already at the bottom of the virtual screen, the screen is scrolled up one line.
#13	CR	Returns the cursor to the left edge of the screen.

Line input

When your program reads from *Input* or a file that has been assigned to the CRT window, text is input one line at a time. The line is stored in the text file's internal buffer, and when variables are read, this buffer is used as the input source. When the buffer empties, a new line is read and stored in the buffer.

When entering lines in the CRT window, the user can use the Backspace key to delete the last character entered. Pressing Enter terminates the input line and stores an end-of-line marker (CR/LF) in the buffer. In addition, if the *CheckEOF* variable is set to *True*, a Ctrl+Z also terminates the input line and generates an end-of-file marker. *CheckEOF* is *False* by default.

To test keyboard status and input single characters under program control, use the *KeyPressed* and *ReadKey* functions.

WinCrt procedures and functions

The following tables list the procedures and functions defined by the *WinCrt* unit.

Table 13-3 WinCrt procedures and functions

Procedure or function	Description
AssignCrt	Associates a text file with the CRT window.
ClrEol	Clears all the characters from the cursor position to the end of the line.
ClrScr	Clears the screen and returns cursor to the upper left-hand corner.
CursorTo	Moves the cursor to the given coordinates within the virtual screen. The origin coordinates are 0,0.
DoneWinCrt	Destroys the CRT window.
GotoXY	Moves the cursor to the given coordinates within the virtual screen. The origin coordinates are 1,1.
InitWinCrt	Creates the CRT window.
KeyPressed	Returns *True* if a key has been pressed on the keyboard.
ReadBuf	Inputs a line from the CRT window.
ReadKey	Reads a character from the keyboard.
ScrollTo	Scrolls the CRT window to show a screen location.
TrackCursor	Scrolls the CRT window to keep the cursor visible.
WhereX	Returns the x-coordinate of the current cursor location. The origin coordinates are 1,1.
WhereY	Returns the y-coordinate of the current cursor location. The origin coordinates are 1,1.
WriteBuf	Writes a block of characters to the CRT window.
WriteChar	Writes a single character to the CRT window.

WinCrt unit variables

The *WinCrt* unit declares several variables:

Table 13-4 WinCrt variables

Variable	Description
WindowOrg	Determines the initial location of the CRT window.
WindowSize	Determines the initial size of the CRT window.

ScreenSize Determines the width and height in characters of the virtual screen within the CRT window.

Cursor Contains the current position of the cursor within the virtual screen. *Cursor* is 0,0 based.

Origin Contains the virtual screen coordinates of the character cell displayed in the upper left corner of the CRT window. *Origin* is 0,0 based.

InactiveTitle Points to a null-terminated string to use when constructing the title of an inactive CRT window.

AutoTracking Enables and disables the automatic scrolling of the window to keep the cursor visible.

CheckEOF Enables and disables the end-of-file character.

CheckBreak Enables and disables user termination of an application.

WindowTitle Determines the title of the CRT window.

Text-file device drivers

Object Pascal lets you define your own text-file device drivers for your Windows programs. A text-file device driver is a set of four functions that completely implement an interface between Object Pascal's file system and some device.

The four functions that define each device driver are *Open*, *InOut*, *Flush*, and *Close*. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where T*TextRec* is the text-file record-type defined on page 166. Each function must be compiled in the {**$F+**} state to force it to use the far call model. The return value of a device-interface function becomes the value returned by *IOResult*. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized *Assign* procedure. The *Assign* procedure must assign the addresses of the four device-interface functions to the four function pointers in the text-file variable. In addition, it should store the *fmClosed* "magic" constant in the *Mode* field, store the size of the text-file buffer in *BufSize*, store a pointer to the text-file buffer in *BufPtr*, and clear the *Name* string.

Assuming, for example, that the four device-interface functions are called *DevOpen*, *DevInOut*, *DevFlush*, and *DevClose*, the *Assign* procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

The device-interface functions can use the *UserData* field in the file record to store private information. This field isn't modified by the Delphi file system at any time.

The Open function

The *Open* function is called by the *Reset*, *Rewrite*, and *Append* standard procedures to open a text file associated with a device. On entry, the *Mode* field contains *fmInput*, *fmOutput*, or *fmInOut* to indicate whether the *Open* function was called from *Reset*, *Rewrite*, or *Append*.

The *Open* function prepares the file for input or output, according to the *Mode* value. If *Mode* specified *fmInOut* (indicating that *Open* was called from *Append*), it must be changed to *fmOutput* before *Open* returns.

Open is always called before any of the other device-interface functions. For that reason, *AssignDev* only initializes the *OpenFunc* field, leaving initialization of the remaining vectors up to *Open*. Based on *Mode*, *Open* can then install pointers to either input- or output-oriented functions. This saves the *InOut*, *Flush* functions and the *CloseFile* from determining the current mode.

The InOut function

The *InOut* function is called by the *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln*, and *CloseFile* standard procedures and functions whenever input or output from the device is required.

When *Mode* is *fmInput*, the *InOut* function reads up to *BufSize* characters into *BufPtr^*, and returns the number of characters read in *BufEnd*. In addition, it stores zero in *BufPos*. If the *InOut* function returns zero in *BufEnd* as a result of an input request, *Eof* becomes *True* for the file.

When *Mode* is *fmOutput*, the *InOut* function writes *BufPos* characters from *BufPtr^*, and returns zero in *BufPos*.

The Flush function

The *Flush* function is called at the end of each *Read, Readln*, *Write,* and *Writeln*. It can optionally flush the text-file buffer.

If *Mode* is *fmInput*, the *Flush* function can store zero in *BufPos* and *BufEnd* to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If *Mode* is *fmOutput*, the *Flush* function can write the contents of the buffer exactly like the *InOut* function, which ensures that text written to the device appears on the device immediately. If *Flush* does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

The Close function

The *Close* function is called by the *CloseFile* standard procedure to close a text file associated with a device. (The *Reset*, *Rewrite*, and *Append* procedures also call *Close* if the file they are opening is already open.) If *Mode* is *fmOutput*, then before calling

Close, Object Pascal's file system calls the *InOut* function to ensure that all characters have been written to the device.

Chapter 14

Using the 80x87

There are two kinds of numbers you can work with in Object Pascal: integers (*Shortint*, *Smallint*, *Longint*, *Byte*, *Word*, *Integer*, *Cardinal*) and reals (*Real*, *Single*, *Double*, *Extended*, *Comp*). Reals are also known as floating-point numbers. The 80x86 family of processors is designed to handle integer values easily, but handling reals is considerably more difficult. To improve floating-point performance, the 80x86 family of processors has a corresponding family of math coprocessors, the 80x87s.

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly, so if you use floating point often, you'll probably want a numeric coprocessor or a 486DX or Pentium processor, which has a numeric coprocessor built in.

By default, Delphi produces code that uses the 80x87 numeric coprocessor. This gives you access to all five real types (*Real*, *Single*, *Double*, *Extended*, and *Comp*), and performs all floating-point operations using the full *Extended* range of 3.4×10^{-4951} to 1.1×10^{4932} with 19 to 20 significant digits.

For compatibility with earlier versions of Object Pascal, Delphi provides a **$N** compiler switch which allows you to control floating-point code generation. The default state is {**$N+**}, and in this state Delphi produces code that uses the 80x87 numeric coprocessor. In the {**$N–**} state, Delphi supports only the *Real* type, and uses a library of software routines to handle floating-point operations. The *Real* type provides a range of 2.9×10^{-39} to 1.7×10^{38} with 11 to 12 significant digits.

Note The Delphi Visual Class Library requires that you compile your applications in the {**$N+**} state. Unless you are compiling an application that doesn't use VCL, you should refrain from using the {**$N–**} state.

To interface with the 80x87 coprocessor, Delphi applications use the WIN87EM.DLL support library that comes with Windows. If an 80x87 coprocessor isn't present in your system, WIN87EM.DLL will *emulate* it in software. Emulation is substantially slower than the real 80x87 coprocessor, but it does guarantee that an application using the 80x87 can be run on any machine.

The 80x87 data types

Delphi fully supports the the single, double, and extended precision native floating-point formats provided by the 80x87 coprocessor. In addition, Delphi supports the 80x87's 64-bit integer format.

- The *Single* type is the smallest format you can use with floating-point numbers. It occupies 4 bytes of memory, providing a range of 1.5×10^{-45} to 3.4×10^{38} with 7 to 8 significant digits.
- The *Double* type occupies 8 bytes of memory, providing a range of 5.0×10^{-324} to 1.7×10^{308} with 15 to 16 significant digits.
- The *Extended* type is the largest floating-point type supported by the 80x87. It occupies 10 bytes of memory, providing a range of 3.4×10^{-4932} to 1.1×10^{4932} with 19 to 20 significant digits. Any arithmetic involving real-type values is performed with the range and precision of the *Extended* type.
- The *Comp* type stores integral values in 8 bytes, providing a range of $-2^{63} +1$ to 2^{63} - 1, which is approximately -9.2×10^{18} to 9.2×10^{18}. *Comp* may be compared to a double-precision *Longint*, but it's considered a real type because all arithmetic done with *Comp* uses the 80x87 coprocessor. *Comp* is well suited for representing monetary values as integral values of cents or mils (thousandths) in business applications.

For backward compatibility, Delphi also provides the *Real* type, which occupies 6 bytes of memory, providing a range of $2.9 \text{ x } 10^{-39}$ to $1.7 \text{ x } 10^{38}$ with 11 to 12 significant digits.

Note that 80x87 floating-point operations on variables of type *Real* are slightly slower than on other types. Because the 80x87 can't directly process the *Real* format, calls must be made to library routines to convert *Real* values to *Extended* before operating on them. If you're concerned with optimum speed you should use the *Single*, *Double*, *Extended*, and *Comp* types exclusively.

Extended range arithmetic

The *Extended* type is the basis of all floating-point computations with the 80x87. Delphi uses the *Extended* format to store all non-integer numeric constants and evaluates all non-integer numeric expressions using extended precision. The entire right side of the following assignment, for example, is computed in *Extended* before being converted to the type on the left side:

```
var
  X, A, B, C: Double;
begin
  X := (B + Sqrt(B * B - A * C)) / A;
end;
```

Delphi automatically performs computations using the precision and range of the *Extended* type. The added precision means smaller round-off errors, and the additional range means overflow and underflow are less common.

You can go beyond Delphi's automatic *Extended* capabilities. For example, you can declare variables used for intermediate results to be of type *Extended*. The following example computes a sum of products:

```
var
  Sum: Single;
  X, Y: array [1..100] of Single;
  I: Integer;
  T: Extended;      { For intermediate results }
begin
  T := 0.0;
  for I := 1 to 100 do
  begin
    X[I] := I;
    Y[I[ := I;
    T := T + X[I] * Y[I];
  end;
  Sum := T;
end;
```

Had *T* been declared *Single*, the assignment to *T* would have caused a round-off error at the limit of single precision at each loop entry. But because *T* is *Extended*, all round-off errors are at the limit of extended precision, except for the one resulting from the assignment of *T* to *Sum*. Fewer round-off errors mean more accurate results.

You can also declare formal value parameters and function results to be of type *Extended*. This avoids unnecessary conversions between numeric types, which can result in loss of accuracy. For example,

```
function Area(Radius: Extended): Extended;
begin
  Area := Pi * Radius * Radius;
end;
```

Comparing reals

Because real-type values are approximations, the results of comparing values of different real types aren't always as expected. For example, if *X* is a variable of type *Single* and *Y* is a variable of type *Double*, then these statements are *False*:

```
X := 1 / 3;
Y := 1 / 3;
Writeln(X = Y);
```

This is because *X* is accurate only to 7 to 8 digits, where *Y* is accurate to 15 to 16 digits, and when both are converted to *Extended*, they will differ after 7 to 8 digits. Similarly, these statements,

```
X := 1 / 3;
Writeln(X = 1 / 3);
```

are *False*, because the result of 1/3 in the *Writeln* statement is calculated with 20 significant digits.

The 80x87 evaluation stack

The 80x87 coprocessor has an internal evaluation stack that can be as deep as eight levels. Accessing a value on the 80x87 stack is much faster than accessing a variable in memory. To achieve the best possible performance, Delphi uses the 80x87's stack for storing temporary results.

In theory, very complicated real-type expressions can overflow the 80x87 stack, but this isn't likely to occur because the expression would need to generate more than eight temporary results.

Detecting the 80x87

The Windows environment and the WIN87EM.DLL emulator library automatically detect the presence of an 80x87 chip. If an 80x87 is available in your system, it's used. If not, WIN87EM.DLL emulates it in software. You can use the *GetWinFlags* function (defined in the *WinProcs* unit) and the *wf_80x87* bit mask (defined in the *WinTypes* unit) to determine whether an 80x87 processor is present in your system. For example,

```
if GetWinFlags and wf_80x87 <> 0 then
  WriteLn('80x87 is present')
else WriteLn('80x87 is not present');
```

Emulation in assembly language

When linking in object files using {**$L** *filename*} directives, make sure that these object files were compiled with the 80x87 emulation enabled. For example, if you're using 80x87 instructions in assembly language **external** procedures, enable emulation when you assemble the .ASM files into .OBJ files. Otherwise, the 80x87 instructions can't be emulated on machines without an 80x87. Use Turbo Assembler's **/E** command-line switch to enable emulation.

Exception statements

The exceptions statements are the **raise** statement, the **try...except** statement, and the **try...finally** statement. These statements are described in Chapter 10, Exceptions.

Chapter

15

Using null-terminated strings

Object Pascal supports a class of character strings called *null-terminated strings*. With Object Pascal's extended syntax and the *SysUtils* unit, your Windows programs can use null-terminated strings by simply referring to the *SysUtils* unit with the **uses** clause in your program.

What is a null-terminated string?

The compiler stores a traditional Object Pascal **string** type as a length byte followed by a sequence of characters. The maximum length of a Pascal string is 255 characters, and a pascal string occupies from 1 to 256 bytes of memory.

A null-terminated string has no length byte; instead, it consists of a sequence of non-null characters followed by a NULL (#0) character. There is no inherent restriction on the length of a null-terminated string.

Using null-terminated strings

Null-terminated strings are stored as arrays of characters with a zero-based integer index type; that is, an array of the form

```
array[0..X] of Char
```

where *X* is a positive nonzero integer. These arrays are called *zero-based character arrays*. Here are some examples of declarations of zero-based character arrays that can be used to store null-terminated strings:

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..79] of Char;
  TMemoText = array[0..1023] of Char;
```

The biggest difference between using Pascal strings and null-terminated strings is the extensive use of pointers in the manipulation of null-terminated strings. Object Pascal performs operations on these pointers with a set of *extended syntax* rules.

Character pointers and string literals

When extended syntax is enabled, a string literal is *assignment-compatible* with the *PChar* type. This means that a string literal can be assigned to a variable of type *PChar*. For example,

```
var
  P: PChar;
   ⋮
begin
  P := 'Hello world...';
end;
```

The effect of such an assignment is that the pointer points to an area of memory that contains a null-terminated copy of the string literal. This example accomplishes the same thing as the previous example:

```
const
  TempString: array[0..14] of Char = 'Hello world...'#0;
var
  P: PChar;
   ⋮
begin
  P := @TempString;
end;
```

You can use string literals as actual parameters in procedure and function calls when the corresponding formal parameter is of type *PChar*. For example, given a procedure with the declaration

```
procedure PrintStr(Str: PChar);
```

the following procedure calls are valid:

```
PrintStr('This is a test');
PrintStr(#10#13);
```

Just as it does with an assignment, the compiler generates a null-terminated copy of the string literal. The compiler passes a pointer to that memory area in the *Str* parameter of the *PrintStr* procedure.

Finally, you can initialize a typed constant of type *PChar* with a string constant. You can do this with structured types as well, such as arrays of *PChar* and records and objects with *PChar* fields.

```
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');
```

A string constant expression is always evaluated as a Pascal-style string even if it initializes a typed constant of type *PChar*; therefore, a string constant expression is always limited to 255 characters in length.

Character pointers and character arrays

When the extended syntax is enabled, a zero-based character array is *compatible* with the *PChar* type. This means that whenever a *PChar* is expected, you can use a zero-based character array instead. When you use a character array in place of a *PChar* value, the compiler converts the character array to a pointer *constant* whose value corresponds to the address of the first element of the array. For example,

```
var
  A: array[0..63] of Char;
  P: PChar;
   ⋮
begin
  P := A;
  PrintStr(A);
  PrintStr(P);
end;
```

Because of this assignment statement, *P* now points to the first element of *A*, so *PrintStr* is called twice with the same value.

You can initialize a typed constant of a zero-based character array type with a string literal that is shorter than the declared length of the array. The remaining characters are set to NULL (#0) and the array effectively contains a null-terminated string.

```
type
  TFileName = array[0..79] of Char;
const
  FileNameBuf: TFileName = 'TEST.PAS';
  FileNamePtr: PChar = FileNameBuf;
```

Character pointer indexing

Just as a zero-based character array is compatible with a character pointer, so can a character pointer be indexed as if it were a zero-based character array:

```
var
  A: array[0..63] of Char;
  P: PChar;
  Ch: Char;
   ⋮
begin
  P := A;
  Ch := A[5];
  Ch := P[5];
end;
```

Both of the last two statements assign *Ch* the value contained in the sixth character element of *A*.

When you index a character pointer, the index specifies an unsigned *offset* to add to the pointer before it is dereferenced. Therefore, *P[0]* is equivalent to *P^* and specifies the character pointed to by *P*. *P[1]* specifies the character right after the one pointed to by *P*, *P[2]* specifies the next character, and so on. For purposes of indexing, a *PChar* behaves as if it were declared as this:

```
type
  TCharArray = array[0..65535] of Char;
  PChar = ^TCharArray;
```

The compiler performs no range checks when indexing a character pointer because it has no type information available to determine the maximum length of the null-terminated string pointed to by the character pointer. Your program must perform any such range checking.

The *StrUpper* function shown here illustrates the use of character pointer indexing to convert a null-terminated string to uppercase.

```
function StrUpper(Str: PChar): PChar;
var
  I: Word;
begin
  I := 0;
  while Str[I] <> #0 do
  begin
     Str[I] := UpCase(Str[I]);
     Inc(I);
  end;
  StrUpper := Str;
end;
```

Notice that *StrUpper* is a function, not a procedure, and that it always returns the value that it was passed as a parameter. Because the extended syntax allows the result of a function call to be ignored, *StrUpper* can be treated as if it were a procedure:

```
StrUpper(A);
PrintStr(A);
```

However, as *StrUpper* always returns the value it was passed, the preceding statements can be combined into one:

```
PrintStr(StrUpper(A));
```

Nesting calls to null-terminated string-handling functions can be very convenient when you want to indicate a certain interrelationship between a set of sequential string manipulations.

Note See page 46 for information about *PChar* operations.

Null-terminated strings and standard procedures

Object Pascal's extended syntax allows the *Read*, *Readln*, *Str*, and *Val* standard procedures to be applied to zero-based character arrays, and allows the *Write*, *Writeln*, *Val*, *Assign*, and *Rename* standard procedures to be applied to both zero-based character arrays and character pointers.

Chapter

16

Memory issues

This chapter is about Object Pascal and memory. This chapter explains how Windows programs use memory, and it also describes the internal data formats used in Object Pascal.

Windows memory management

This section explains how Delphi applications use memory.

Code segments

Each module (the main program or library and each unit) in a Delphi application or DLL has its own code segment. The size of a single code segment can't exceed 64K, but the total size of the code is limited only by the available memory.

Segment attributes

Each code segment has a set of attributes that determine the behavior of the code segment when it's loaded into memory.

MOVEABLE or FIXED

When a code segment is MOVEABLE, Windows can move the segment around in physical memory to satisfy other memory allocation requests. When a code segment is FIXED, it *never* moves in physical memory. The preferred attribute is MOVEABLE, and unless it's absolutely necessary to keep a code segment at the same address in physical memory (such as if it contains an interrupt handler), you should use the MOVEABLE attribute. When you do need a fixed code segment, keep that code segment as small as possible.

PRELOAD or DEMANDLOAD

A code segment that has the PRELOAD attribute is automatically loaded when the application or library is activated. The DEMANDLOAD attribute delays the loading of the segment until a routine in the segment is actually called. Although this takes longer, it allows an application to execute in less space.

DISCARDABLE or PERMANENT

When a segment is DISCARDABLE, Windows can free the memory occupied by the segment when it needs to allocate additional memory. When a segment is PERMANENT, it's kept in memory at all times.

When an application makes a call to a DISCARDABLE segment that's not in memory, Windows first loads it from the .EXE file. This takes longer than if the segment were PERMANENT, but it allows an application to execute in less space.

Changing attributes

The default attributes of a code segment are MOVEABLE, DEMANDLOAD, and DISCARDABLE, but you can change this with a **$C** compiler directive. For example,

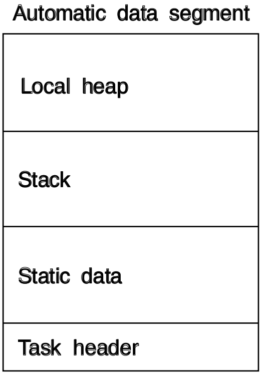
```
{$C MOVEABLE PRELOAD PERMANENT}
```

For details about the $C compiler directive, see Appendix B.

There is no need for a separate overlay manager. The Windows memory manager includes a full set of overlay management services, controlled through code segment attributes. These services are available to any Windows application.

The automatic data segment

Each application or library has one data segment called the "automatic data segment," which can be up to 64K in size. The automatic data segment is always pointed to by the data segment register (DS). It's divided into four sections: the *local heap*, the *stack*, the *static data*, and the *task header*.



The first 16 bytes of the automatic data segment always contain the *task header* in which Windows stores various system information.

The *static data* area contains all global variables and typed constants declared by the application or library.

The *stack* is used to store local variables allocated by procedures and functions. On entry to an application, the stack segment register (SS) and the stack pointer (SP) are loaded so that SS:SP points to the first byte past the stack area in the automatic data segment. When procedures and functions are called, SP is moved down to allocate space for parameters, the return address, and local variables. When a routine returns, the process is reversed by incrementing SP to the value it had before the call. The default size of the stack area in the automatic data segment is 16K, but this can be changed with a **$M** compiler directive.

Unlike an application, a library has no stack area in its automatic data segment. When a call is made to a procedure or function in a DLL, the DS register points to the library's automatic data segment, but the SS:SP register pair isn't modified. Therefore, a library always uses the stack of the calling application.

The last section in the automatic data segment is the *local heap*. It contains all local dynamic data that was allocated using the *LocalAlloc* function in Windows. The default size of the local heap section is 8K, but this can be changed with a **$M** compiler directive.

The local heap is used by Windows for the temporary storage of things such as edit control and list box buffers. Never set the local heap to zero.

The heap manager

Windows supports dynamic memory allocations on two different heaps: The *global heap* and the *local heap*.

The global heap is a pool of memory available to all applications. Although global memory blocks of any size can be allocated, the global heap is intended only for large memory blocks (256 bytes or more). Each global memory block carries an overhead of at least 20 bytes, and there is a system-wide limit of 8192 global memory blocks, only some of which are available to any given application.

The local heap is a pool of memory available only to your application or library. It exists in the upper part of an application's or library's data segment. The total size of local memory blocks that can be allocated on the local heap is 64K minus the size of the application's stack and static data. For this reason, the local heap is best suited for small memory blocks (256 bytes or less). The default size of the local heap is 8K, but you can change this with the **$M** compiler directive.

Delphi includes a *heap manager* which implements the *New*, *Dispose*, *GetMem*, and *FreeMem* standard procedures. The heap manager uses the global heap for all allocations. Because the global heap has a system-wide limit of 8192 memory blocks (which certainly is less than what some applications might require), Delphi's heap manager includes a *segment sub-allocator* algorithm to enhance performance and allow a substantially larger number of blocks to be allocated.

Note To read more about using the heap manager in a DLL, see Chapter 12.

This is how the segment sub-allocator works: When allocating a large block, the heap manager simply allocates a global memory block using the Windows *GlobalAlloc* routine. When allocating a small block, the heap manager allocates a larger global memory block and then divides (sub-allocates) that block into smaller blocks as required. Allocations of small blocks reuse all available sub-allocation space before the heap manager allocates a new global memory block, which, in turn, is further sub-allocated.

The *HeapLimit* variable defines the threshold between small and large heap blocks. The default value is 1024 bytes. The *HeapBlock* variable defines the size the heap manager uses when allocating blocks to be assigned to the sub-allocator. The default value of *HeapBlock* is 8192 bytes. You should have no reason to change the values of *HeapLimit* and *HeapBlock*, but if you do, make sure that *HeapBlock* is at least four times the size of *HeapLimit*.

The *HeapAllocFlags* variable defines the attribute flags value passed to *GlobalAlloc* when the heap manager allocates global blocks. In a program, the default value is *gmem_Moveable*, and in a library the default value is *gmem_Moveable* + *gmem_DDEShare*.

Internal data formats

The next several pages discuss the internal data formats of Object Pascal.

Integer types

The format selected to represent an integer-type variable depends on its minimum and maximum bounds:

- If both bounds are within the range -128..127 (*Shortint*), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (*Byte*), the variable is stored as an unsigned byte.
- If both bounds are within the range -32768..32767 (*Smallint*), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (*Word*), the variable is stored as an unsigned word.
- Otherwise, the variable is stored as a signed double word (*Longint*).

Char types

A *Char* or a subrange of a *Char* type is stored as an unsigned byte.

Boolean types

A *Boolean* type is stored as a *Byte*, a *ByteBool* is stored as a *Byte*, a *WordBool* type is stored as a *Word*, and a *LongBool* is stored as a *Longint*.

A *Boolean* can assume the values 0 (*False*) and 1 (*True*). *ByteBool*, *WordBool*, and *LongBool* types can assume the value of 0 (*False*) or nonzero (*True*).

Enumerated types

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values, and if the type was declared in the {$**Z**-} state (the default). If an enumerated type has more than 256 values, or if the type was declared in the {$**Z**+} state, it is stored as an unsigned word.

Floating-point types

The floating-point types (*Real*, *Single*, *Double*, *Extended*, and *Comp*) store the binary representations of a sign (**+**or **-**), an *exponent*, and a *significand*. A represented number has the value

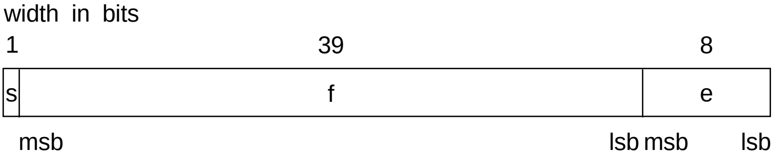
$$\texttt{+/- significand * } 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary decimal point (that is, 0 <= significand < 2).

In the figures that follow, *msb* means most significant bit and *lsb* means least significant bit. The leftmost items are stored at the highest addresses. For example, for a real-type value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real type

A 6-byte (48-bit) *Real* number is divided into three fields:



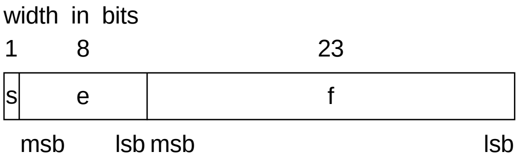
The value *v* of the number is determined by the following:

```
if 0 < e <= 255, then v = $(-1)^{s} * 2^{(e-129)} * (1.f)$.
if e = 0, then v = 0.
```

The *Real* type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a *Real*, and NaNs and infinities produce an overflow error if an attempt is made to store them in a *Real*.

The Single type

A 4-byte (32-bit) *Single* number is divided into three fields:

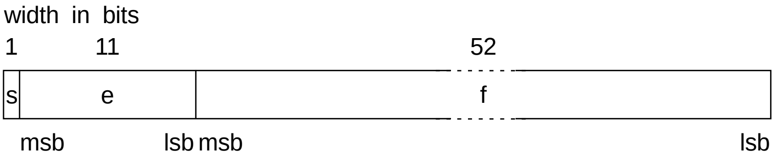


The value *v* of the number is determined by the following:

```
if 0 < e < 255, then v = $(-1)^{s} * 2^{(e-127)} * (1.f)$.
if e = 0   and f <> 0, then v = $(-1)^{s} * 2^{(-126)} * (0.f)$.
if e = 0   and f = 0, then v = $(-1)^{s} * 0$.
if e = 255 and f = 0, then v = $(-1)^{s}$ * Inf.
if e = 255 and f <> 0, then v is a NaN.
```

The Double type

An 8-byte (64-bit) *Double* number is divided into three fields:

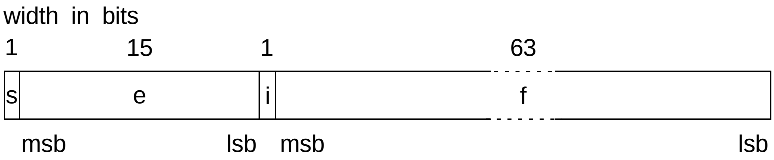


The value *v* of the number is determined by the following:

```
if 0 < e < 2047, then v = $(-1)^{s} * 2^{(e-1023)} * (1.f)$.
if e = 0    and f <> 0, then v = $(-1)^{s} * 2^{(-1022)} * (0.f)$.
if e = 0    and f = 0, then v = $(-1)^{s} * 0$.
if e = 2047 and f = 0, then v = $(-1)^{s}$ * Inf.
if e = 2047 and f <> 0, then v is a NaN.
```

The Extended type

A 10-byte (80-bit) *Extended* number is divided into four fields:

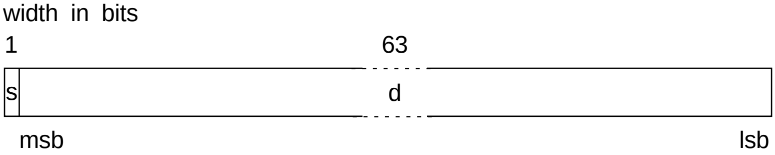


The value *v* of the number is determined by the following:

```
if 0 <= e < 32767, then v = $(-1)^{s} * 2^{(e-16383)} * (i.f)$.
if e = 32767 and f = 0, then v = $(-1)^{s}$ * Inf.
if e = 32767 and f <> 0, then v is a NaN.
```

The Comp type

An 8-byte (64-bit) *Comp* number is divided into two fields:



The value *v* of the number is determined by the following:

```
if s = 1 and d = 0, then v is a NaN
```

Otherwise, *v* is the two's complement 64-bit value.

Pointer types

A *Pointer* type is stored as two words (a double word), with the offset part in the low word and the segment part in the high word. The pointer value **nil** is stored as a double-word zero.

String types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (**string** [255]).

Set types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is calculated as

```
ByteSize = (Max div 8) - (Min div 8) + 1
```

where *Min* and *Max* are the lower and upper bounds of the base type of that set. The byte number of a specific element *E* is

```
ByteNumber = (E div 8) - (Min div 8)
```

and the bit number within that byte is

```
BitNumber = E mod 8
```

where *E* denotes the ordinal value of the element.

Array types

An array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

Record types

The fields of a record are stored as a contiguous sequence of variables. The first field is stored at the lowest memory address. If the record contains variant parts, then each variant starts at the same memory address.

File types

File types are represented as records. Typed files and untyped files occupy 128 bytes, which are laid out as follows:

```
type
  TFileRec = record
    Handle: Word;
    Mode: Word;
    RecSize: Word;
    Private: array [1..26] of Byte;
    UserData: array [1..16] of Byte;
    Name: array [0..79] of Char;
  end;
```

Text files occupy 256 bytes, which are laid out as follows:

```
type
  TTextBuf = array [0..127] of Char;
  TTextRec = record
    Handle: Word;
    Mode: Word;
    BufSize: Word;
    Private: Word;
    BufPos: Word;
    BufEnd: Word;
    BufPtr: ^TTextBuf;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array [1..16] of Byte;
    Name: array [0..79] of Char;
    Buffer: TTextBuf;
  end;
```

Handle contains the file's handle (when the file is open).

The *Mode* field can assume one of the following values:

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

fmClosed indicates that the file is closed. *fmInput* and *fmOutput* indicate that the file is a text file that has been reset (*fmInput*) or rewritten (*fmOutput*). *fmInOut* indicates that the file variable is a typed or an untyped file that has been reset or rewritten. Any other value indicates that the file variable hasn't been assigned (and thereby not initialized).

The *UserData* field is never accessed by Object Pascal and is free for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file. The section entitled "Text file device drivers" in Chapter 13 provides information on that subject.

Procedural types

A global procedure pointer type is stored as a 32-bit pointer to the entry point of a procedure or function.

A method pointer type is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

Class types

A class type value is stored as a 32-bit pointer to an instance of the class. An instance of a class is also known as an object.

The internal data format of an object resembles that of a record. The fields of an object are stored in order of declaration as a contiguous sequences of variables. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first four-byte field of every object is a pointer to the *virtual method table (VMT)* of the class. There is only one VMT per class (not one per instance), but two distinct class types never share a VMT, no matter how identical they appear to be. VMTs are built automatically by the compiler, and are never directly manipulated by a program. Likewise, pointers to VMTs are automatically stored in class instances by the class type's constructor(s) and are never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointers, one per user-defined virtual method in the class type, in order of declaration. Each slot contains the address of the corresponding virtual method's entry point. This layout is compatible with a C++ v-table, and the OLE Object Model used by Windows Object Linking and Embedding. At negative offsets, a VMT contains a number of fields that are internal to Object Pascal's implementation. These fields are listed here for informational purposes only. An application should use the methods defined in *TObject* to query this information, since the layout is likely to change in future implementations of Object Pascal.

Table 16-1 Virtual Method Table layout

Offset	Type	Description
-32	Word	Near pointer to type information table (or **nil**).
-30	Word	Near pointer to field definition table (or **nil**).
-28	Word	Near pointer to method definition table (or **nil**).
-26	Word	Near pointer to dynamic method table (or **nil**).
-24	Word	Near pointer to string containing class name.
-22	Word	Instance size in bytes.
-20	Pointer	Pointer to ancestor class (or **nil**).
-16	Pointer	Entry point of *DefaultHandler* method.
-12	Pointer	Entry point of *NewInstance* method.
-8	Pointer	Entry point of *FreeInstance* method.
-4	Pointer	Entry point of *Destroy* destructor.
0	Pointer	Entry point of first user-defined virtual method.
4	Pointer	Entry point of second user-defined virtual method.
...	...	...

Class reference types

A class reference type value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

Direct memory access

Object Pascal implements three predefined arrays, *Mem*, *MemW*, and *MemL*, which are used to directly access memory. Each component of *Mem* is a byte, each component of *MemW* is a *Word*, and each component of *MemL* is a *Longint*.

The *Mem* arrays use a special syntax for indexes: Two expressions of the integer type *Word*, separated by a colon, are used to specify the segment base and offset of the memory location to access. Here are two examples:

```
Mem[Seg0040:$0049] := 7;
Data := MemW[Seg(V):Ofs(V)];
```

The first statement stores the value 7 in the byte at $0040:$0049. The second statement moves the *Word* value stored in the first 2 bytes of the variable *V* into the variable *Data*.

Direct port access

For access to the 80x86 CPU data ports, Object Pascal implements two predefined arrays, *Port* and *PortW*. Both are one-dimensional arrays, and each element represents a data port, whose port address corresponds to its index. The index type is the integer type *Word*. Components of the *Port* array are of type *Byte* and components of the *PortW* array are of type *Word*.

When a value is assigned to a component of *Port* or *PortW*, the value is output to the selected port. When a component of *Port* or *PortW* is referenced in an expression, its value is input from the selected port.

Use of the *Port* and *PortW* arrays is restricted to assignment and reference in expressions only; that is, components of *Port* and *PortW* can't be used as variable parameters. Also, references to the entire *Port* or *PortW* array (reference without index) aren't allowed

Chapter

17

Control issues

This chapter describes in detail the various ways that Delphi implements program control. Included are calling conventions and exit procedures.

Calling conventions

Parameters are transferred to procedures and functions via the stack. Before calling a procedure or function, the parameters are pushed onto the stack in their order of declaration. Before returning, the procedure or function removes all parameters from the stack.

The skeleton code for a procedure or function call looks like this:

```
PUSH   Param1
PUSH   Param2
  .
  .
  .
PUSH   ParamX
CALL   ProcOrFunc
```

Parameters are passed either by *reference* or by *value*. When a parameter is passed by reference, a pointer that points to the actual storage location is pushed onto the stack. When a parameter is passed by value, the actual value is pushed onto the stack.

Variable parameters

Variable parameters (**var** parameters) are always passed by reference—a pointer that points to the actual storage location.

Value and constant parameters

Value parameters are passed by value or by reference depending on the type and size of the parameter. In general, if the value parameter occupies 1, 2, or 4 bytes, the value is pushed directly onto the stack. Otherwise a pointer to the value is pushed, and the procedure or function then copies the value into a local storage location.

The 8086 does not support byte-sized PUSH and POP instructions, so byte-sized parameters are always transferred onto the stack as words. The low-order byte of the word contains the value, and the high-order byte is unused (and undefined).

An integer type or parameter is passed as a byte, a word, or a double word, using the same format as an integer-type variable. (For double words, the high-order word is pushed before the low-order word so that the low-order word ends up at the lowest address.)

A *Char* parameter is passed as an unsigned byte.

A *Boolean* parameter is passed as a byte with the value 0 or 1.

An enumerated-type parameter is passed as an unsigned byte if the enumeration has 256 or fewer values; otherwise, it is passed as an unsigned word.

A floating-point type parameter (*Real*, *Single*, *Double*, *Extended*, and *Comp*) is passed as 4, 6, 8, or 10 bytes on the stack. This is an exception to the rule that only 1-, 2-, and 4-byte values are passed directly on the stack.

A pointer-type, class-type, or class-reference-type parameter is passed as two words (a double word). The segment part is pushed before the offset part so that the offset part ends up at the lowest address.

A string-type parameter is passed as a pointer to the value.

For a set type parameter, if the bounds of the element type of the set are both within the range 0 to 7, the set is passed as a byte. If the bounds are both within the range 0 to 15, the set is passed as a word. Otherwise, the set is passed as a pointer to an unpacked set that occupies 32 bytes.

Arrays and records with 1, 2, or 4 bytes are passed directly onto the stack. Other arrays and records are passed as pointers to the value.

A global procedure pointer type is passed as a pointer.

A method pointer type is passed as two pointers. The instance pointer is pushed before the method pointer so that the method pointer ends up at the lowest address.

Open parameters

Open string parameters are passed by first pushing a pointer to the string and then pushing a word containing the size attribute (maximum length) of the string.

Open array parameters are passed by first pushing a pointer to the array and then pushing a word containing the number of elements in the array less one.

When using the built-in assembler, the value that the *High* standard function returns for an open parameter can be accessed by loading the word just below the open parameter. In this example, the *FillString* procedure, which fills a string to its maximum length with a given character, demonstrates this.

```
procedure FillString(var Str: OpenString; Chr: Char); assembler;
asm
  LES      DI,Str           { ES:DI = @Str }
  MOV      CX,Str.Word[-2]  { CX = High(Str) }
  MOV      AL,CL
  CLD
  STOSB                     { Set Str[0] }
  MOV      AL,Chr
  REP      STOSB            { Set Str[1..High] }
end;
```

Function results

Ordinal-type function results are returned in the CPU registers: Bytes are returned in AL, words are returned in AX, and double words are returned in DX:AX (high-order word in DX, low-order word in AX).

Real-type function results (type *Real*) are returned in the DX:BX:AX registers (high-order word in DX, middle word in BX, low-order word in AX).

80x87-type function results (type *Single*, *Double*, *Extended*, and *Comp*) are returned in the 80x87 coprocessor's top-of-stack register (ST(0)).

Pointer-type, class-type, and class-reference-type function results are returned in DX:AX (segment part in DX, offset part in AX).

For a string-type function result, the caller pushes a pointer to a temporary storage location before pushing any parameters, and the function returns a string value in that temporary location. The function must not remove the pointer.

For array, record, and set type function results, if the value occupies one byte, it is returned in AL, if the value occupies two bytes, it is returned in AX, and if the value occupies four bytes, it is returned in DX:AX. Otherwise, the caller pushes a pointer to a temporary storage location of the appropriate size, and the function returns the result in that temporary location. Upon returning, the function leaves the temporary pointer on the stack.

A global procedure pointer type is returned in DX:AX.

A method pointer type is returned in BX:CX:DX:AX, where DX:AX contains the method pointer and BX:CX contains the instance pointer.

NEAR and FAR calls

The 80x86 family of CPUs supports two kinds of call and return instructions: NEAR and FAR. The NEAR instructions transfer control to another location within the same code segment, and the FAR instructions allow a change of code segment.

A NEAR CALL instruction pushes a 16-bit return address (offset only) onto the stack, and a FAR CALL instruction pushes a 32-bit return address (both segment and offset). The corresponding RET instructions pop only an offset or both an offset and a segment.

Delphi automatically selects the correct call model based on the procedure's declaration. Procedures declared in the interface section of a unit are far—they can be called from other units. Procedures declared in a program or in the **implementation** section of a unit are near—they can only be called from within that program or unit.

For some specific purposes, a procedure can be required to be far. For example, if a procedure or function is to be assigned to a procedural variable, it must far. The **$F** compiler directive is used to override the compiler's automatic call model selection. Procedures and functions compiled in the {**$F+**} state are always far; in the {**$F-**} state, Delphi automatically selects the correct model. The default state is {**$F-**}.

Nested procedures and functions

A procedure or function is said to be nested when it is declared within another procedure or function. By default, nested procedures and functions always use the near call model, because they are visible only within a specific procedure or function in the same code segment.

When calling a nested procedure or function, the compiler generates a PUSH BP instruction just before the CALL, in effect passing the caller's BP as an additional parameter. Once the called procedure has set up its own BP, the caller's BP is accessible as a word stored at [BP + 4], or at [BP + 6] if the procedure is far. Using this link at [BP + 4] or [BP + 6], the called procedure can access the local variables in the caller's stack frame. If the caller itself is also a nested procedure, it also has a link at [BP + 4] or [BP + 6], and so on. The following example demonstrates how to access local variables from an **inline** statement in a nested procedure:

```
procedure A; near;
var
  IntA: Integer;

procedure B; far;
var
  IntB: Integer;

procedure C; near;
var
  IntC: Integer;
begin
  asm
    MOV     AX,1
    MOV     IntC,AX                { IntC := 1 }
    MOV     BX,[BP+4]              { B's stack frame }
    MOV     SS:[BX+OFFSET IntB],AX { IntB := 1 }
    MOV     BX,[BP+4]              { B's stack frame }
    MOV     BX,SS:[BX+6]           { A's stack frame }
    MOV     SS:[BX+OFFSET IntA],AX  { IntA := 1 }
  end;
end;

begin C end;

begin B end;
```

Nested procedures and functions can't be declared with the external directive, and they cannot be procedural parameters.

Method calling conventions

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter, *Self*, which is a reference to the class or instance for which the method is invoked. The *Self* parameter is always passed as the last parameter, and is always a 32-bit pointer. For regular methods, *Self* is a class type value (a pointer to an instance). For class methods, *Self* is a class reference type value (a pointer to a virtual method table).

For example, given the declarations

```
type
  TMyObject = class(TObject)
    procedure Test(X, Y: Integer);
    procedure Foo; virtual;
    class procedure Bar; virtual;
  end;
  TMyClass = class of TMyObject;
var
  MyObject: TMyObject;
  MyClass: TMyClass;
```

the call *MyObject*.*Test*(10, 20) generates the following code:

```
PUSH    10
PUSH    20
LES     DI,MyObject
PUSH    ES
PUSH    DI
CALL    MyObject.Test
```

Upon returning, a method must remove the *Self* parameter from the stack, just as it must remove any normal parameters.

Methods always use the far call model, regardless of the setting of the $**F** compiler directive.

To call a virtual method, the compiler generates code that loads the VMT pointer from the object, and then calls via the slot associated with the method. Referring to the declarations above, the call *MyObject*.*Foo* generates the following code:

```
LES     DI,MyObject
PUSH    ES
PUSH    DI
```

```
  LES     DI,ES:[DI]
  CALL    DWORD PTR ES:[DI]
```

the call *MyObject.Bar* generates the following code:

```
  LES     DI,MyObject
  LES     DI,ES:[DI]
  PUSH    ES
  PUSH    DI
  CALL    DWORD PTR ES:[DI+4]
```

and the call *MyClass.Bar* generates the following code:

```
  LES     DI,MyClass
  PUSH    ES
  PUSH    DI
  CALL    DWORD PTR ES:[DI+4]
```

Constructors and destructors

Constructors and destructors use the same calling conventions as other methods, except that an aditional word-sized flag parameter is passed on the stack just before the *Self* parameter.

A zero in the flag parameter of a constructor call indicates that the constructor was called through an instance or using the **inherited** keyword. In this case, the constructor behaves like an ordinary method.

A non-zero value in the flag parameter of a constructor call indicates that the constructor was called through a class reference. In this case, the constructor creates an instance of the class given by *Self*, and returns a reference to the newly created object in DX:AX.

A zero in the flag parameter of a destructor call indicates that the destructor was called using the **inherited** keyword. In this case, the destructor behaves like an ordinary method.

A non-zero value in the flag parameter of a destructor call indicates that the destructor was called through an instance. In this case, the destructor deallocates the instance given by *Self* just before returning.

Entry and exit code

This is the standard entry and exit code for a procedure or function using the near call model:

```
  PUSH    BP                ;Save BP
  MOV     BP,SP             ;Set up stack frame
  SUB     SP,LocalSize      ;Allocate locals (if any)
    .
    .
    .
  MOV     SP,BP             ;Deallocate locals (if any)
```

```
POP     BP                ;Restore BP
RETN    ParamSize         ;Remove parameters and return
```

For information on using exit procedures in a DLL, see Chapter 12, "Dynamic-link libraries."

If the routine is compiled in the {**$W-**} state (the default), the entry and exit code for a routine using the far call model is the same as that of a routine using the near call model, except that a far-return instruction (RETF) is used to return from the routine.

In the {**$W+**} state, this is the entry and exit code for a routine using the far call model:

```
INC     BP                ;Indicate FAR frame
PUSH    BP                ;Save odd BP
MOV     BP,SP             ;Set up stack frame
PUSH    DS                ;Save DS
SUB     SP,LocalSize      ;Allocate locals (if any)
  .
  .
  .
MOV     SP,BP             ;Remove locals and saved DS
POP     BP                ;Restore odd BP
DEC     BP                ;Adjust BP
RETF    ParamSize         ;Remove parameters and return
```

This is the entry and exit code for an exportable routine (a procedure or function compiled with the **export** compiler directive):

```
MOV     AX,DS             ;Load DS selector into AX
NOP                       ;Additional space for patching
INC     BP                ;Indicate FAR frame
PUSH    BP                ;Save odd BP
MOV     BP,SP             ;Set up stack frame
PUSH    DS                ;Save DS
MOV     DS,AX             ;Initialize DS
SUB     SP,LocalSize      ;Allocate locals (if any)
PUSH    SI                ;Save SI
PUSH    DI                ;Save DI
  .
  .
  .
POP     DI                ;Restore DI
POP     SI                ;Restore SI
LEA     SP,[BP-2]         ;Deallocate locals (if any)
POP     DS                ;Restore DS
POP     BP                ;Restore odd BP
DEC     BP                ;Adjust BP
RETF    ParamSize         ;Remove parameters and return
```

For all call models, the instructions required to allocate and deallocate local variables are omitted if the routine has no local variables.

Occasionally you might find {**$W+**} useful while developing a Windows protected-mode application. Some non-Borland debugging tools require this state to work properly.

By default, Delphi automatically generates *smart callbacks* for procedures and functions that are exported by an application. When linking an application in the {**$K+**} state (the default), the linker looks for a MOV AX,DS instruction followed by a NOP instruction at every exported entry point and, for each such sequence it finds, it changes the MOV AX,DS to a MOV AX,SS. This change alleviates the need to use the Windows *MakeProcInstance* and *FreeProcInstance* API routines when creating callback routines (although it isn't harmful to do so), and also makes it possible to call exported entry points from within the application itself.

In the {**$K-**} state or when creating a dynamic-link library, the Delphi linker makes no modifications to the entry code of exported entry points. Unless a callback routine in an application is to be called from another application (which isn't recommended anyway), you shouldn't have to ever select the {**$K-**} state.

When loading an application or dynamic-link library, Windows looks for a MOV AX,DS followed by a NOP at each exported entry point. For applications, the sequence is changed into three NOP instructions to prepare the routine for use with *MakeProcInstance*. For libraries, the sequence is changed into a MOV AX,xxxx instruction, where xxxx is the selector (segment address) of the library's automatic data segment. Because smart callback entry points start with a MOV AX,SS instruction, they are left untouched by the Windows program loader.

Register-saving conventions

Procedures and functions should preserve the BP, SP, SS, and DS registers. All other registers can be modified. In addition, exported routines should preserve the SI and DI registers.

Exit procedures

By installing an exit procedure, you can gain control over a program's termination process. This is useful when you want to make sure specific actions are carried out before a program terminates; a typical example is updating and closing files.

The *ExitProc* pointer variable allows you to install an exit procedure. The exit procedure is always called as a part of a program's termination, whether it's a normal termination, a termination through a call to *Halt*, or a termination due to a run-time error.

An exit procedure takes no parameters and must be compiled with a **far** procedure directive to force it to use the far call model.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures. This chain makes it possible for units as well as programs to install exit procedures. Some units install an exit procedure as part of their initialization code and then rely on that specific procedure to be called to clean up after the unit.

Closing files is such an example. The procedures on the exit chain are executed in reverse order of installation. This ensures that the exit code of one unit isn't executed before the exit code of any units that depend upon it.

To keep the exit chain intact, you must save the current contents of *ExitProc* before changing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of *ExitProc*. The following code demonstrates a skeleton method of implementing an exit procedure:

```
var
  ExitSave: Pointer;

procedure MyExit; far;
begin
  ExitProc := ExitSave;            { Always restore old vector first }
  .
  .
  .
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  .
  .
  .
end.
```

On entry, the code saves the contents of *ExitProc* in *ExitSave*, and then installs the *MyExit* exit procedure. After having been called as part of the termination process, the first thing *MyExit* does is reinstall the previous exit procedure

The termination routine in the run-time library keeps calling exit procedures until *ExitProc* becomes **nil**. To avoid infinite loops, *ExitProc* is set to **nil** before every call, so the next exit procedure is called only if the current exit procedure assigns an address to *ExitProc*. If an error occurs in an exit procedure, it won't be called again.

An exit procedure can learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable.

In case of normal termination, *ExitCode* is zero and *ErrorAddr* is **nil**. In case of termination through a call to *Halt*, *ExitCode* contains the value passed to *Halt*, and *ErrorAddr* is **nil**. Finally, in case of termination due to a run-time error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the statement in error.

The last exit procedure (the one installed by the run-time library) closes the *Input* and *Output* files. If *ErrorAddr* is not **nil**, it outputs a run-time error message.

If you wish to present run-time error messages yourself, install an exit procedure that examines *ErrorAddr* and outputs a message if it's not **nil**. In addition, before returning, make sure to set *ErrorAddr* to **nil**, so that the error is not reported again by other exit procedures.

Once the run-time library has called all exit procedures, it returns to Windows, passing the value stored in *ExitCode* as a return code.

Chapter

18

Optimizing your code

Delphi performs several different types of code optimizations, ranging from constant folding and short-circuit Boolean expression evaluation, all the way up to smart linking. The following sections describe some of the types of optimizations performed and how you can benefit from them in your programs.

Constant folding

If the operand(s) of an operator are constants, Delphi evaluates the expression at compile time. For example,

```
X := 3 + 4 * 2
```

generates the same code as `X := 11` and

```
S := 'In' + 'Out'
```

generates the same code as S := 'InOut'.

Likewise, if an operand of an *Abs*, *Chr*, *Hi*, *Length*, *Lo*, *Odd*, *Ord*, *Pred*, *Ptr*, *Round*, *Succ*, *Swap*, or *Trunc* function call is a constant, the function is evaluated at compile time.

If an array index expression is a constant, the address of the component is evaluated at compile time. For example, accessing *Data* [5, 5] is just as efficient as accessing a simple variable.

Constant merging

Using the same string constant two or more times in a statement part generates only one copy of the constant. For example, two or more Write('Done') statements in the same statement part references the same copy of the string constant 'Done'.

Short-circuit evaluation

Delphi implements short-circuit Boolean evaluation, which means that evaluation of a Boolean expression stops as soon as the result of the entire expression becomes evident. This guarantees minimum execution time and usually minimum code size. Short-circuit evaluation also makes possible the evaluation of constructs that would not otherwise be legal. For example,

```
while (I <= Length(S)) and (S[I] <> ' ') do
  Inc(I);
while (P <> nil) and (P^.Value <> 5) do
  P := P^.Next;
```

In both cases, the second test isn't evaluated if the first test is *False*.

The opposite of short-circuit evaluation is complete evaluation, which is selected through a {**$B+**} compiler directive. In this state, every operand of a Boolean expression is guaranteed to be evaluated.

Constant parameters

Whenever possible, you should use constant parameters instead of value parameters. Constant parameters are at least as efficient as value parameters and, in many cases, more efficient. In particular, constant parameters generate less code and execute faster than value parameters for structured and string types.

Constant parameters are more efficient than value parameters because the compiler doesn't have to generate copies of the actual parameters upon entry to procedures or functions. Value parameters have to be copied into local variables so that modifications made to the formal parameters won't modify the actual parameters. Because constant formal parameters can't be modified, the compiler has no need to generate copies of the actual parameters and code and stack space is saved. Read more about constant parameters on page 76.

Redundant pointer-load elimination

In certain situations, Delphi’s code generator can eliminate redundant pointer-load instructions, shrinking the size of the code and allowing for faster execution. When the code generator can guarantee that a particular pointer remains *constant* over a stretch of linear code (code with no jumps into it), and when that pointer is already loaded into a register pair (such as ES:DI), the code generator eliminates more redundant pointer load instructions in that block of code.

A pointer is considered constant if it's obtained from a variable parameter (variable parameters are always passed as pointers) or from the variable reference of a **with** statement. Because of this, using **with** statements is often more efficient (but never less efficient) than writing the fully qualified variable for each component reference.

Constant set inlining

When the right operand of the **in** operator is a set constant, the compiler generates the inclusion test using inline CMP instructions. Such inlined tests are more efficient than the code that would be generated by a corresponding Boolean expression using relational operators. For example, the statement

```
if ((Ch >= 'A') and (Ch <= 'Z')) or
  ((Ch >= 'a') and (Ch <= 'z')) then ... ;
```

is less readable and also less efficient than

```
if Ch in ['A'..'Z', 'a'..'z'] then ... ;
```

Because constant folding applies to set constants as well as to constants of other types, it's possible to use **const** declarations without any loss of efficiency:

```
const
  Upper = ['A'..'Z'];
  Lower = ['a'..'z'];
  Alpha = Upper + Lower;
```

Given these declarations, this **if** statement generates the same code as the previous **if** statement:

```
if Ch in Alpha then ... ;
```

Small sets

The compiler generates very efficient code for operations on *small sets*. A small set is a set with a lower bound ordinal value in the range 0..7 and an upper bound ordinal value in the range 0..15. For example, the following *TByteSet* and *TWordSet* are both small sets.

```
type
  TByteSet = set of 0..7;
  TWordSet = set of 0..15;
```

Small set operations, such as union (**+**), difference (**-**), intersection (*), and inclusion tests (**in**) are generated inline using AND, OR, NOT, and TEST machine code instructions instead of calls to run-time library routines. Likewise, the *Include* and *Exclude* standard procedures generate inline code when applied to small sets.

Order of evaluation

As permitted by the Pascal standards, operands of an expression are frequently evaluated differently from the left to right order in which they are written. For example, the statement

```
I := F(J) div G(J);
```

where *F* and *G* are functions of type *Integer*, causes *G* to be evaluated before *F*, because this enables the compiler to produce better code. For this reason, it's important that an expression never depend on any specific order of evaluation of the embedded functions. Referring to the previous example, if *F* must be called before *G*, use a temporary variable:

```
T := F(J);
I := T div G(J);
```

As an exception to this rule, when short-circuit evaluation is enabled (the {**$B-**} state), Boolean operands grouped with **and** or **or** are *always* evaluated from left to right.

Range checking

Assignment of a constant to a variable and use of a constant as a value parameter is range-checked at compile time; no run-time range-check code is generated. For example, X := 999, where *X* is of type *Byte*, causes a compile-time error.

Shift instead of multiply or divide

The operation *X* * *C*, where *C* is a constant and a power of 2, is coded using a SHL instruction. The operation *X* **div** *C*, where *X* is an unsigned integer (*Byte* or *Word*) and *C* is a constant and a power of 2, is coded using a SHR instruction.

Likewise, when the size of an array's components is a power of 2, a SHL instruction (not a MUL instruction) is used to scale the index expression.

Automatic word alignment

By default, Delphi aligns all variables and typed constants larger than 1 byte on a machine-word boundary. On all 16-bit 80x86 CPUs, word alignment means faster execution, because word-sized items on even addresses are accessed faster than words on odd addresses.

Data alignment is controlled through the **$A** compiler directive. In the default {**$A+**} state, variables and typed constants are aligned as described above. In the {**$A-**} state, no alignment measures are taken.

Eliminating dead code

Statements that never execute don't generate any code. For example, these constructs don't generate code:

```
if False then
  statement
while False do
```

```
    statement
```

Smart linking

Delphi’s built-in linker automatically removes unused code and data when building an .EXE file. Procedures, functions, variables, and typed constants that are part of the compilation, but are never referenced, are removed from the .EXE file. The removal of unused code takes place on a per procedure basis; the removal of unused data takes place on a per declaration section basis.

Consider the following program:

```
program SmartLink;

const
  H: array [0..15] of Char = '0123456789ABCDEF';

var
  I, J: Integer;
  X, Y: Real;

var
  S: string [79];

var
  A: array [1..10000] of Integer;

procedure P1;
begin
  A[1] := 1;
end;

procedure P2;
begin
  I := 1;
end;

procedure P3;
begin
  S := 'Borland Pascal';
  P2;
end;

begin
  P3;
end.
```

The main program calls *P3*, which calls *P2*, so both *P2* and *P3* are included in the .EXE file. Because *P2* references the first **var** declaration section, and *P3* references the second **var** declaration, *I*, *J*, *X*, *Y*, and *S* are also included in the .EXE file. No references are made to *P1*, however, and none of the included procedures reference *H* and *A*, so these objects are removed.

An example of such a unit is the *SysUtils* standard unit: It contains a number of procedures and functions, all of which are seldom used by the same program. If a program uses only one or two procedures from *SysUtils*, then only these procedures are included in the final .EXE file, and the remaining ones are removed, greatly reducing the size of the .EXE file.

Chapter

19

The built-in assembler

Delphi's built-in assembler allows you to write 8086/8087 and 80286/80287 assembler code directly inside your Object Pascal programs. Of course, you can still convert assembler instructions to machine code manually for use in **inline** statements, or link in .OBJ files that contain **external** procedures and functions when you want to mix Object Pascal and assembler.

The built-in assembler implements a large subset of the syntax supported by Turbo Assembler and Microsoft's Macro Assembler. The built-in assembler supports all 8086/8087 and 80286/80287 opcodes, and all but a few of Turbo Assembler's expression operators.

Except for DB, DW, and DD (define byte, word, and double word), none of Turbo Assembler's directives, such as EQU, PROC, STRUC, SEGMENT, and MACRO, are supported by the built-in assembler. Operations implemented through Turbo Assembler directives, however, are largely matched by corresponding Delphi constructs. For example, most EQU directives correspond to **const**, **var**, and **type** declarations in Delphi, the PROC directive corresponds to **procedure** and **function** declarations, and the STRUC directive corresponds to Delphi **record** types. In fact, Delphi's built-in assembler can be thought of as an assembler language compiler that uses Object Pascal syntax for all declarations.

The asm statement

The built-in assembler is accessed through **asm** statements. This is the syntax of an **asm** statement:

```
asm AsmStatement [ Separator AsmStatement ] end
```

where *AsmStatement* is an assembler statement, and *Separator* is a semicolon, a new-line, or a Object Pascal comment.

Multiple assembler statements can be placed on one line if they are separated by semicolons. A semicolon isn't required between two assembler statements if the statements are on separate lines. A semicolon doesn't indicate that the rest of the line is a comment--comments must be written in Object Pascal style using **{** and **}** or **(***and ***)**.

Register use

In general, the rules of register use in an **asm** statement are the same as those of an **external** procedure or function. An **asm** statement must preserve the BP, SP, SS, and DS registers, but can freely modify the AX, BX, CX, DX, SI, DI, ES, and Flags registers. On entry to an **asm** statement, BP points to the current stack frame, SP points to the top of the stack, SS contains the segment address of the stack segment, and DS contains the segment address of the data segment. Except for BP, SP, SS, and DS, an **asm** statement can assume nothing about register contents on entry to the statement.

Assembler statement syntax

This is the syntax of an assembler statement:

```
[ Label ":" ] < Prefix > [ Opcode [ Operand < "," Operand > ] ]
```

Label is a label identifier, *Prefix* is an assembler prefix opcode (operation code), *Opcode* is an assembler instruction opcode or directive, and *Operand* is an assembler expression.

Comments are allowed between assembler statements, but not within them. For example, this is allowed:

```
asm
  MOV AX,1 {Initial value}
  MOV CX,100 {Count}
end;
```

but this is an error:

```
asm
  MOV {Initial value} AX,1;
  MOV CX, {Count} 100
end;
```

Labels

Labels are defined in assembler as they are in Object Pascal--by writing a label identifier and a colon before a statement. There is no limit to a label length, except only the first 32 characters of an identifier are significant in the built-in assembler. And as they are in Object Pascal, labels defined in assembler must be declared in a **label** declaration part in the block containing the **asm** statement. There is one exception to this rule: *local labels*.

Local labels are labels that start with an at-sign (@). Because an at-sign can't be part of a Object Pascal identifier, such local labels are automatically restricted to use within **asm** statements. A local label is known only within the **asm** statement that defines it (that is, the scope of a local label extends from the **asm** keyword to the **end** keyword of the **asm** statement that contains it).

Unlike a normal label, a local label doesn't have to be declared in a **label** declaration part before it's used.

The exact composition of a local label identifier is an at-sign (@) followed by one or more letters (*A*.. *Z*), digits (0..9), underscores (_), or at-signs. As with all labels, the identifier is followed by a colon (:).

Instruction opcodes

The built-in assembler supports all 8086/8087 and 80286/80287 instruction opcodes. 8087 opcodes are available only in the {**$N+**} state (numeric processor enabled), 80286 opcodes are available only in the {**$G+**} state (80286 code generation enabled), and 80287 opcodes are available only in the {**$G+**,**N+**} state.

For a complete description of each instruction, refer to your 80x86 and 80x87 reference manuals.

RET instruction sizing

The RET instruction opcode generates a near return or a far return machine code instruction depending on the call model of the current procedure or function.

```
procedure NearProc; near;
begin
  asm
    RET    { Generates a near return }
  end;
end;

procedure FarProc; far;
begin
  asm
    RET    { Generates a far return }
  end;
end;
```

On the other hand, the RETN and RETF instructions always generate a near return and a far return, regardless of the call model of the current procedure or function.

Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and all conditional jump instructions, when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one byte opcode followed by a one byte displacement) if the distance to the target label is within -128 to 127 bytes; otherwise a near jump (one byte opcode followed by a two byte displacement) is generated.

For a conditional jump instruction, a short jump (1 byte opcode followed by a 1 byte displacement) is generated if the distance to the target label is within -128 to 127 bytes; otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (5 bytes in total). For example, the assembler statement

```
JC      Stop
```

where *Stop* isn't within reach of a short jump is converted to a machine code sequence that corresponds to this:

```
JNC     Skip
JMP     Stop
Skip:
```

Jumps to the entry points of procedures and functions are always either near or far, but never short, and conditional jumps to procedures and functions are not allowed. You can force the built-in assembler to generate an unconditional near jump or far jump to a label by using a NEAR PTR or FAR PTR construct. For example, the assembler statements

```
JMP     NEAR PTR Stop
JMP     FAR PTR Stop
```

always generates a near jump and a far jump, respectively, even if *Stop* is a label within reach of a short jump.

Assembler directives

Delphi's built-in assembler supports three assembler directives: DB (define byte), DW (define word), and DD (define double word). They each generate data corresponding to the comma-separated operands that follow the directive.

The DB directive generates a sequence of bytes. Each operand can be a constant expression with a value between -128 and 255, or a character string of any length. Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The DW directive generates a sequence of words. Each operand can be a constant expression with a value between -32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer, that is, a word that contains the offset part of the address.

The DD directive generates a sequence of double words. Each operand can be a constant expression with a value between -2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer, that is, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembler statements. To generate uninitialized or initialized data in the data segment, you should use Object Pascal **var** or **const** declarations.

Some examples of DB, DW, and DD directives follow:

```
asm
  DB        0FFH                      { One byte }
  DB        0,99                      { Two bytes }
  DB        'A'                       { Ord('A') }
  DB        'Hello world...',0DH,0AH  { String followed by CR/LF }
  DB        12,"Delphi"               { Object Pascal style string }
  DW        0FFFFH                    { One word }
  DW        0,9999                    { Two words }
  DW        'A'                       { Same as DB 'A',0 }
  DW        'BA'                      { Same as DB 'A','B' }
  DW        MyVar                     { Offset of MyVar }
  DW        MyProc                    { Offset of MyProc }
  DD        0FFFFFFFFH                { One double-word }
  DD        0,999999999               { Two double-words }
  DD        'A'                       { Same as DB 'A',0,0,0 }
  DD        'DCBA'                    { Same as DB 'A','B','C','D' }
  DD        MyVar                     { Pointer to MyVar }
  DD        MyProc                    { Pointer to MyProc }
end;
```

In Turbo Assembler, when an identifier precedes a DB, DW, or DD directive, it causes the declaration of a byte, word, or double-word sized variable at the location of the directive. For example, Turbo Assembler allows the following:

```
ByteVar      DB      ?
WordVar      DW      ?
  .
  .
  .
             MOV     AL,ByteVar
             MOV     BX,WordVar
```

The built-in assembler doesn't support such variable declarations. In Delphi, the only kind of symbol that can be defined in an built-in assembler statement is a label. All variables must be declared using Object Pascal syntax, and the preceding construct corresponds to this:

```
var
  ByteVar: Byte;
  WordVar: Word;
  .
  .
  .
asm
  MOV     AL,ByteVar
  MOV     BX,WordVar
```

```
end;
```

Operands

Built-in assembler operands are expressions that consist of a combination of constants, registers, symbols, and operators. Although built-in assembler expressions are built using the same basic principles as Object Pascal expressions, there are a number of important differences, as will be explained later in this chapter.

Within operands, the following reserved words have a predefined meaning to the built-in assembler:

Table 19-1 Built-in assembler reserved words

AH
AL
AND
AX
BH
BL
BP
BX
BYTE
CH
CL
CS
CX
DH
DI
DL
DS
DWORD
DX
ES
FAR
HIGH
LOW
MOD
NEAR
NOT
OFFSET
OR
PTR
QWORD
SEG
SHL
SHR
SI
SP
SS
ST
TBYTE
TYPE
WORD
XOR

The reserved words always take precedence over user-defined identifiers. For example, the code fragment,

```
var
  ch: Char;
  .
  .
  .
asm
  MOV     CH, 1
end;
```

loads 1 into the CH register, *not* into the *CH* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (&) identifier override operator:

```
asm
  MOV     &ch, 1
end;
```

It's strongly suggested that you avoid user-defined identifiers with the same names as built-in assembler reserved words, because such name confusion can easily lead to obscure and hard-to-find bugs.

Expressions

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants.

Built-in assembler expressions are built from *expression elements* and *operators*, and each expression has an associated *expression class* and *expression type*. These concepts are explained in the following sections.

Differences between Object Pascal and Assembler expressions

The most important difference between Object Pascal expressions and built-in assembler expressions is that all built-in assembler expressions must resolve to a *constant value*, a value that can be computed at compile time. For example, given these declarations:

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid built-in assembler statement:

```
asm
  MOV     Z,X+Y
end;
```

Because both *X* and *Y* are constants, the expression *X* + *Y* is merely a more convenient way of writing the constant 30, and the resulting instruction becomes a move immediate of the value 30 into the word-sized variable *Z*. But if you change *X* and *Y* to be variables,

```
var
  X, Y: Integer;
```

the built-in assembler can no longer compute the value of *X* + *Y* at compile time. The correct built-in assembler construct to move the sum of *X* and *Y* into *Z* is this:

```
asm
  MOV     AX,X
  ADD     AX,Y
  MOV     Z,AX
end;
```

Another important difference between Object Pascal and built-in assembler expressions is the way variables are interpreted. In a Object Pascal expression, a reference to a variable is interpreted as the *contents* of the variable, but in an built-in assembler expression, a variable reference denotes the *address* of the variable. For example, in Object Pascal, the expression *X* **+** 4, where *X* is a variable, means the contents of *X plus* 4, whereas in the built-in assembler, it means the contents of the word at an address four bytes higher than the address of *X*. So, even though you're allowed to write

```
asm
  MOV     AX,X+4
end;
```

the code doesn't load the value of *X* plus 4 into AX, but it loads the value of a word stored four bytes beyond *X* instead. The correct way to add 4 to the contents of *X* is like this:

```
asm
  MOV     AX,X
  ADD     AX,4
end;
```

Expression elements

The basic elements of an expression are *constants*, *registers*, and *symbols*.

Constants

The built-in assembler supports two types of constants: *numeric constants* and *string constants*.

Numeric constants

Numeric constants must be integers, and their values must be between -2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal (base 10) notation, but the built-in assembler supports binary (base 2), octal (base 8), and hexadecimal (base 16) notations as well. Binary notation is selected by writing a *B* after the number, octal notation is selected by writing a letter *O* after the number, and hexadecimal notation is selected by writing an *H* after the number or a $ before the number.

The *B*, *O*, and *H* suffixes aren't supported in Object Pascal expressions. Object Pascal expressions allow only decimal notation (the default) and hexadecimal notation (using a $ prefix).

Numeric constants must start with one of the digits 0 through 9 or a $ character; therefore when you write a hexadecimal constant using the *H* suffix, an extra zero in front of the number is required if the first significant digit is one of the hexadecimal digits *A* through *F*. For example, 0BAD4H and $BAD4 are hexadecimal constants, but BAD4H is an identifier because it starts with a letter and not a digit.

String constants

String constants must be enclosed in single or double quotes. Two consecutive quotes of the same type as the enclosing quotes count as only one character. Here are some examples of string constants:

```
'Z'
'Delphi'
"That's all folks"
'"That''s all folks," he said.'
```

```
'100'
'"'
"'"
```

Notice in the fourth string the use of two consecutive single quotes to denote one single quote character.

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters, and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where *Ch1* is the rightmost (last) character and *Ch4* is the leftmost (first) character. If the string is shorter than four characters, the leftmost (first) character(s) are assumed to be zero. Here are some examples of string constants and their corresponding numeric values:

Table 19-2 String examples and their values

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
‘ a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers:

Table 19-3 CPU registers

16-bit general purpose	AX BX CX DX
8-bit low registers	AL BL CL DL
8-bit high registers	AH BH CH DH
16-bit pointer or index	SP BP SI DI
16-bit segment registers	CS DS SS ES
8087 register stack	ST

When an operand consists solely of a register name, it's called a register operand. All registers can be used as register operands. In addition, some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI].

The segment registers (ES, CS, SS, and DS) can be used in conjunction with the colon (**:**) segment override operator to indicate a different segment than the one the processor selects by default.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(*x*), where *x* is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The built-in assembler allows you to access almost all Object Pascal symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the following special symbols:

@Code @Data @Result

The *@Code* and *@Data* symbols represent the current code and data segments. They should only be used in conjunction with the SEG operator:

```
asm
  MOV     AX,SEG @Data
  MOV     DS,AX
end;
```

The *@Result* symbol represents the function result variable within the statement part of a function. For example, in this function,

```
function Sum(X, Y: Integer): Integer;
begin
  Sum := X + Y;
end;
```

the statement that assigns a function result value to *Sum* uses the *@Result* variable if it is written in built-in assembler:

```
function Sum(X, Y: Integer): Integer;
begin
  asm
    MOV     AX,X
    ADD     AX,Y
    MOV     @Result,AX
  end;
end;
```

The following symbols can't be used in built-in assembler expressions:

- Standard procedures and functions (for example, *WriteLn*, *Chr*)
- The *Mem*, *MemW*, *MemL*, *Port*, and *PortW* special arrays
- String, floating-point, and set constants
- Procedures and functions declared with the **inline** directive
- Labels that aren't declared in the current block

- The *@Result* symbol outside a function

The following table summarizes the value, class, and type of the different kinds of symbols that can be used in built-in assembler expressions. (Expression classes and types are described in a following section.)

Table 19-4 Values, classes, and types of symbols

Symbol	Value	Class	Type
Label	Address of label	Memory	SHORT
Constant	Value of constant	Immediate	0
Type	0	Memory	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory	Size of type
Procedure	Address of procedure	Memory	NEAR or FAR
Function	Address of function	Memory	NEAR or FAR
Unit	0	Immediate	0
@Code	Code segment address	Memory	0FFF0H
@Data	Data segment address	Memory	0FFF0H
@Result	Result var offset	Memory	Size of type

Local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to SS:BP, and the value of a local variable symbol is its signed offset from SS:BP. The assembler automatically adds [BP] in references to local variables. For example, given these declarations,

```
procedure Test;
var
  Count: Integer;
```

the instruction

```
asm
  MOV     AX,Count
end;
```

assembles into MOV AX,[BP-2].

The built-in assembler always treats a **var** parameter as a 32-bit pointer, and the size of a **var** parameter is always 4 (the size of a 32-bit pointer). In Object Pascal, the syntax for accessing a **var** parameter and a value parameter is the same--this isn't the case in code you write for the built-in assembler. Because **var** parameters are really pointers, you have to treat them as such. So, to access the contents of a **var** parameter, you first have to load the 32-bit pointer and then access the location it points to. For example, if the X and Y parameters of the above function *Sum* were **var** parameters, the code would look like this:

```
function Sum(var X, Y: Integer): Integer;
begin
  asm
    LES     BX,X
    MOV     AX,ES:[BX]
    LES     BX,Y
    ADD     AX,ES:[BX]
    MOV     @Result,AX
  end;
end;
```

Some symbols, such as record types and variables, have a scope that can be accessed using the period (.) structure member selector operator. For example, given these declarations,

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

the following constructs can be used to access fields in the *P* and *R* variables:

```
asm
  MOV     AX,P.X
  MOV     DX,P.Y
  MOV     CX,R.A.X
  MOV     BX,R.B.Y
end;
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of ES:[DI+4] into AX:

```
asm
  MOV     AX,(TRect PTR ES:[DI]).B.X
  MOV     AX,TRect(ES:[DI]).B.X
  MOV     AX,ES:TRect[DI].B.X
  MOV     AX,TRect[ES:DI].B.X
  MOV     AX,ES:[DI].TRect.B.X
end;
```

A scope is provided by type, field, and variable symbols of a record or object type. In addition, a unit identifier opens the scope of a particular unit, just like a fully qualified identifier in Object Pascal.

Expression classes

The built-in assembler divides expressions into three classes: *registers*, *memory references*, and *immediate values*.

An expression that consists solely of a register name is a register expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references; Object Pascal's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values; this group includes Object Pascal's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
  Start = 10;
var
  Count: Integer;
  .
  .
  .
asm
  MOV     AX,Start            { MOV AX,xxxx }
  MOV     BX,Count            { MOV BX,[xxxx] }
  MOV     CX,[Start]          { MOV CX,[xxxx] }
  MOV     DX,OFFSET Count     { MOV DX,xxxx }
end;
```

Because *Start* is an immediate value, the first MOV is assembled into a move immediate instruction. The second MOV, however, is translated into a move memory instruction, as *Count* is a memory reference. In the third MOV, the square brackets operator is used to convert *Start* into a memory reference (in this case, the word at offset 10 in the data segment), and in the fourth MOV, the OFFSET operator is used to convert *Count* into an immediate value (the offset of *Count* in the data segment).

As you can see, the square brackets and the OFFSET operators complement each other. In terms of the resulting machine code, the following **asm** statement is identical to the first two lines of the previous **asm** statement:

```
asm
  MOV     AX,OFFSET [Start]
  MOV     BX,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either *relocatable expressions* or *absolute expressions.* A relocatable expression denotes a value that requires *relocation* at link time, and an absolute expression denotes a value that requires no such relocation. Typically, an expression that refers to a label, variable, procedure, or function is relocatable, and an expression that operates solely on constants is absolute.

Relocation is the process by which the linker assigns absolute addresses to symbols. At compile time, the compiler doesn't know the final address of a label, variable, procedure, or function; it doesn't become known until link time, when the linker assigns a specific absolute address to the symbol.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression types

Every built-in assembler expression has an associated type--or more correctly, an associated size, because the built-in assembler regards the type of an expression simply as the size of its memory location. For example, the type (size) of an *Integer* variable is two, because it occupies 2 bytes.

The built-in assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  .
  .
  .
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

the built-in assembler checks that the size of *QuitFlag* is one (a byte), and that the size of *OutBufPtr* is two (a word). An error results if the type check fails; for example, this isn't allowed:

```
asm
  MOV     DL,OutBufPtr
end;
```

The problem is DL is a byte-sized register and *OutBufPtr* is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
asm
  MOV     DL,BYTE PTR OutBufPtr
  MOV     DL,Byte(OutBufPtr)
  MOV     DL,OutBufPtr.Byte
end;
```

These MOV instructions all refer to the first (least significant) byte of the *OutBufPtr* variable.

In some cases, a memory reference is untyped; that is, it has no associated type. One example is an immediate value enclosed in square brackets:

```
asm
  MOV     AL,[100H]
  MOV     BX,[100H]
end;
```

The built-in assembler permits both of these instructions, because the expression [100H] has no associated type--it just means "the contents of address 100H in the data segment," and the type can be determined from the first operand (byte for AL, word for BX). In cases where the type can't be determined from another operand, the built-in assembler requires an explicit typecast:

```
asm
  INC     BYTE PTR [100H]
  IMUL    WORD PTR [100H]
end;
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Object Pascal types.

Table 19-5 Predefined type symbols

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFEH
FAR	0FFFFH

Notice in particular the NEAR and FAR pseudotypes, which are used by procedure and function symbols to indicate their call model. You can use NEAR and FAR in typecasts just like other symbols. For example, if *FarProc* is a FAR procedure,

```
procedure FarProc; far;
```

and if you are writing built-in assembler code in the same module as *FarProc*, you can use the more efficient NEAR call instruction to call it:

```
asm
  PUSH    CS
  CALL    NEAR PTR FarProc
end;
```

Expression operators

The built-in assembler provides a variety of operators, divided into 12 classes of precedence. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

Built-in assembler operator precedence is different from Object Pascal. For example, in a built-in assembler expression, the AND operator has lower precedence than the plus (+) and minus (-) operators, whereas in a Object Pascal expression, it has higher precedence.

Table 19-6 Summary of built-in asssembler expression operators

Operator(s)	Comments
&	Identifier override operator

(), [],HIGH, LOW	Structure member selector
+, -	Unary operators
:	Segment override operator
OFFSET, SEG, TYPE, PTR, *, /, MOD, SHL, SHR,+, -	Binary addition/ subtraction operators
NOT, AND, OR, XOR	Bitwise operators

Table 19-7 Definitions of built-in assembler expression operators

Operator	Description
&	**Identifier override.** The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	**Subexpression.** Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can optionally precede the expression within the parentheses; the result in this case becomes the sum of the values of the two expressions, with the type of the first expression.
[...]	**Memory reference.** The expression within brackets is evaluated completely prior to being treated as a single expression element. The expression within brackets can be combined with the BX, BP, SI, or DI registers using the plus (+) operator, to indicate CPU register indexing. Another expression can optionally precede the expression within the brackets; the result in this case becomes the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector.** The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	**Unary plus.** Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
-	**Unary minus.** Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
:	**Segment override.** Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction will be prefixed by an appropriate segment override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (low-order word) of the expression following the operator. The result is an immediate value.
SEG	Returns the segment part (high-order word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	**Typecast operator.** The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*****	**Multiplication.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right.** Both expressions must be absolute immediate values, and the result

is an absolute immediate value.

+ **Addition.** The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions are memory references, the result is also a memory reference.

- **Subtraction.** The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.

NOT **Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.

AND **Bitwise AND.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.

OR **Bitwise OR.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.

XOR **Bitwise exclusive OR.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembler procedures and functions

So far, every **asm**... **end** construct you've seen has been a statement within a normal **begin**... **end** statement part. Delphi's assembler directive allows you to write complete procedures and functions in built-in assembler, without the need for a **begin**... **end** statement part. Here's an example of an assembler function:

```
function LongMul(X, Y: Integer): Longint; assembler;
asm
  MOV     AX,X
  IMUL    Y
end;
```

The **assembler** directive causes Delphi to perform a number of code generation optimizations:

- The compiler doesn't generate code to copy value parameters into local variables. This affects all string-type value parameters, and other value parameters whose size isn't 1, 2, or 4 bytes. Within the procedure or function, such parameters must be treated as if they were **var** parameters.
- The compiler doesn't allocate a function result variable, and a reference to the *@Result* symbol is an error. String functions, however, are an exception to this rule--they always have a *@Result* pointer that is allocated by the caller.
- The compiler generates no stack frame for procedures and functions that aren't nested and have no parameters and no local variables.
- The automatically generated entry and exit code for an assembler procedure or function looks like this:

```
PUSH    BP              ;Present if Locals <> 0 or Params <> 0
MOV     BP,SP           ;Present if Locals <> 0 or Params <> 0
SUB     SP,Locals       ;Present if Locals <> 0
  .
  .
  .
MOV     SP,BP           ;Present if Locals <> 0
```

```
POP     BP              ;Present if Locals <> 0 or Params <> 0
RET     Params          ;Always present
```

- *Locals* is the size of the local variables, and *Params* is the size of the parameters. If both *Locals* and *Params* are zero, there is no entry code, and the exit code consists simply of a RET instruction.

Functions using the **assembler** directive must return their results as follows:

- Ordinal-type function results (integer, boolean, enumerated types, and *Char*) are returned in AL (8-bit values), AX (16-bit values), or DX:AX (32-bit values).
- Real-type function results (type *Real*) are returned in DX:BX:AX.
- 8087-type function results (type *Single*, *Double*, *Extended*, and *Comp*) are returned in ST(0) on the 8087 coprocessor's register stack.
- Pointer-type function results are returned in DX:AX.
- String-type function results are returned in the temporary location pointed to by the *@Result* function result symbol.

The **assembler** directive is comparable to the **external** directive, and **assembler** procedures and functions must obey the same rules as **external** procedures and functions. The following examples demonstrate some of the differences between **asm** statements in Object Pascal functions and assembler functions. The first example uses an **asm** statement in a Object Pascal function to convert a string to upper case. Notice that the value parameter *Str* in this case refers to a local variable, because the compiler automatically generates entry code that copies the actual parameter into local storage.

```
function UpperCase(Str: String): String;
begin
  asm
    CLD
    LEA     SI,Str
    LES     DI,@Result
    SEGSS   LODSB
    STOSB
    XOR     AH,AH
    XCHG    AX,CX
    JCXZ    @3
  @1:
    SEGSS   LODSB
    CMP     AL,'a'
    JB      @2
    CMP     AL,'z'
    JA      @2
    SUB     AL,20H

  @2:
    STOSB
    LOOP    @1
  @3:
```

```
  end;
end;
```

The second example is an assembler version of the *UpperCase* function. In this case, *Str* isn't copied into local storage, and the function must treat *Str* as a **var** parameter.

```
function UpperCase(Str: String): String; assembler;
asm
  PUSH    DS
  CLD
  LDS     SI,Str
  LES     DI,@Result
  LODSB
  STOSB
  XOR     AH,AH
  XCHG    AX,CX
  JCXZ    @3
@1:
  LODSB
  CMP     AL,'a'
  JB      @2
  CMP     AL,'z'
  JA      @2
  SUB     AL,20H
@2:
  STOSB
  LOOP    @1
@3:
  POP     DS
end;
```

Chapter

20

Linking assembler code

Procedures and functions written in assembly language can be linked with Delphi programs or units using the **$L** compiler directive. The assembly language source file must be assembled into an object file (extension .OBJ) using an assembler like Turbo Assembler. Multiple object files can be linked with a program or unit through multiple **$L** directives.

Procedures and functions written in assembly language must be declared as **external** in the Object Pascal program or unit. For example,

```
function LoCase(Ch: Char): Char; external;
```

In the corresponding assembly language source file, all procedures and functions must be placed in a segment named CODE or CSEG, or in a segment whose name ends in _TEXT. The names of the external procedures and functions must appear in PUBLIC directives.

You must ensure that an assembly language procedure or function matches its Object Pascal definition with respect to call model (near or far), number of parameters, types of parameters, and result type.

An assembly language source file can declare initialized variables in a segment named CONST or in a segment whose name ends in _DATA. It can declare uninitialized variables in a segment named DATA or DSEG, or in a segment whose name ends in _BSS. Such variables are private to the assembly language source file and can't be referenced from the Object Pascal program or unit. However, they reside in the same segment as the Object Pascal globals, and can be accessed through the DS segment register.

All procedures, functions, and variables declared in the Object Pascal program or unit, and the ones declared in the **interface** section of the used units, can be referenced from the assembly language source file through EXTRN directives. Again, it's up to you to supply the correct type in the EXTRN definition.

When an object file appears in a **$L** directive, Delphi converts the file from the Intel relocatable object module format (.OBJ) to its own internal relocatable format. This conversion is possible only if certain rules are observed:

- All procedures and functions must be placed in a segment named CODE or CSEG, or in a segment with a name that ends in _TEXT. All initialized private variables must be placed in a segment named CONST, or in a segment with a name that ends in _DATA. All uninitialized private variables must be placed in a segment named DATA or DSEG, or in a segment with a name that ends in _BSS. All other segments are ignored, and so are GROUP directives. The segment definitions can specify BYTE or WORD alignment, but when linked, code segments are always byte aligned, and data segments are always word aligned. The segment definitions can optionally specify PUBLIC and a class name, both of which are ignored.
- Delphi ignores any data for segments other than the code segment (CODE, CSEG, or *xxxx*_TEXT) and the initialized data segment (CONST or *xxxx*_DATA). So, when declaring variables in the uninitialized data segment (DATA, DSEG, or *xxxx*_BSS), always use a question mark (?) to specify the value, for instance:

```
Count   DW  ?
Buffer  DB  128 DUP(?)
```

- Byte-sized references to EXTRN symbols aren't allowed. For example, this means that the assembly language HIGH and LOW operators can't be used with EXTRN symbols.

Turbo Assembler and Delphi

Turbo Assembler (TASM) makes it easy to program routines in assembly language and interface them into your Delphi programs. Turbo Assembler provides simplified segmentation and language support for Object Pascal programmers.

The .MODEL directive specifies the memory model for an assembler module that uses simplified segmentation. For linking with Object Pascal programs, the .MODEL syntax looks like this:

```
.MODEL xxxx, PASCAL
```

xxxx is the memory model (usually this is large).

Specifying the language PASCAL in the .MODEL directive tells Turbo Assembler that the arguments were pushed onto the stack from left to right, in the order they were encountered in the source statement that called the procedure.

The PROC directive lets you define your parameters in the same order as they are defined in your Object Pascal program. If you are defining a function that returns a string, notice that the PROC directive has a RETURNS option that lets you access the temporary string pointer on the stack without affecting the number of parameter bytes added to the RET statement.

Here's an example coded to use the .MODEL and PROC directives:

```
        .MODEL LARGE, PASCAL
        .CODE
MyProc  PROC   FAR I : BYTE, J : BYTE RETURNS Result : DWORD
        PUBLIC MyProc
        LES    DI, Result        ;get address of temporary string
        MOV    AL, I             ;get first parameter I
        MOV    BL, J             ;get second parameter J
          .
          .
          .
        RET
```

The Object Pascal function definition would look like this:

```
function MyProc(I, J: Char): string; external;
```

For more information about interfacing Turbo Assembler with Delphi, refer to the *Turbo Assembler User's Guide*.

Examples of assembly language routines

The following code is an example of a unit that implements two assembly language string-handling routines. The *UpperCase* function converts all characters in a string to uppercase, and the *StringOf* function returns a string of characters of a specified length.

```
unit Stringer;
interface
function UpperCase(S: String): String;
function StringOf(Ch: Char; Count: Byte): String;
implementation
{$L STRS}
function UpperCase; external;
function StringOf; external;
end.
```

The assembly language file that implements the *UpperCase* and *StringOf* routines is shown next. It must be assembled into a file called STRS.OBJ before the *Stringer* unit can be compiled. Note that the routines use the far call model because they are declared in the **interface** section of the unit. This example uses standard segmentation:

```
        CODE    SEGMENT BYTE PUBLIC

        ASSUME  CS:CODE
        PUBLIC  UpperCase, StringOf     ;Make them known

; function UpperCase(S: String): String

UpperRes        EQU     DWORD PTR [BP + 10]
UpperStr        EQU     DWORD PTR [BP + 6]
```

```
UpperCase       PROC FAR

        PUSH    BP                      ;Save BP
        MOV     BP, SP                  ;Set up stack frame
        PUSH    DS                      ;Save DS
        LDS     SI, Upperstr            ;Load string address
        LES     DI, Upperres            ;Load result address
        CLD                             ;Forward string-ops
        LODSB                           ;Load string length
        STOSB                           ;Copy to result
        MOV     CL, AL                  ;String length to CX
        XOR     CH, CH
        JCXZ    U3                      ;Skip if empty string
U1:     LODSB                           ;Load character
        CMP     AL, 'a'                 ;Skip if not 'a'..'z'
        JB      U2
        CMP     AL, 'z'
        JA      U2
        SUB     AL, 'a'-'A'             ;Convert to uppercase
U2:     STOSB                           ;Store in result
        LOOP    U1                      ;Loop for all characters
U3:     POP     DS                      ;Restore DS
        POP     BP                      ;Restore BP
        RET     4                       ;Remove parameter and return

UpperCase       ENDP

; procedure StringOf(var S: String; Ch: Char; Count: Byte)

StrOfS          EQU     DWORD PTR [BP + 10]
StrOfChar       EQU     BYTE PTR [BP + 8]
StrOfCount      EQU     BYTE PTR [BP + 6]

StringOf        PROC FAR

        PUSH    BP                      ;Save BP
        MOV     BP, SP                  ;Set up stack frame
        LES     DI, StrOfRes            ;Load result address
        MOV     AL, StrOfCount          ;Load count
        CLD                             ;Forward string-ops
        STOSB                           ;Store length
        MOV     CL, AL                  ;Count to CX
        XOR     CH, CH
        MOV     AL, StrOfChar           ;Load character
        REP     STOSB                   ;Store string of characters
        POP     BP                      ;Restore BP
        RET     8                       ;Remove parameters and return

StringOf        ENDP

CODE    ENDS
```

```
          END
```

To assemble the example and compile the unit, use the following commands:

```
TASM STRS
BPC stringer
```

Assembly language methods

Method implementations written in assembly language can be linked with Delphi programs using the **$L** compiler directive and the **external** reserved word. The declaration of an external method in an object type is no different than that of a normal method; however, the implementation of the method lists only the method header followed by the reserved word **external**.

In an assembly language source text, an @ is used instead of a period (.) to write qualified identifiers (the period already has a different meaning in assembly language and can't be part of an identifier). For example, the Object Pascal identifier *Rect*. *Init* is written as *Rect@Init* in assembly language. The @ syntax can be used to declare both PUBLIC and EXTRN identifiers.

Inline machine code

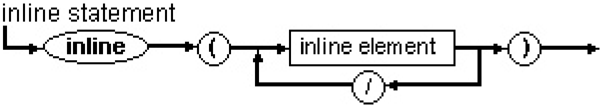
For very short assembly language subroutines, Delphi's **inline** statements and directives are very convenient. They let you insert machine code instructions directly into the program or unit text instead of through an object file.

Inline statements

An **inline** statement consists of the reserved word **inline** followed by one or more inline elements, separated by slashes and enclosed in parentheses:

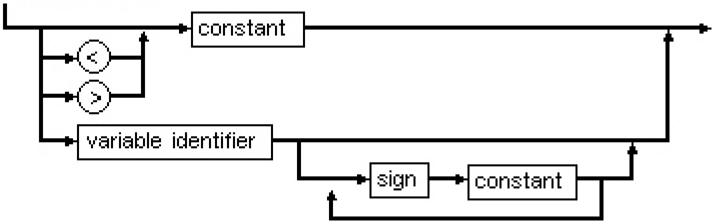
```
inline (10/$2345/Count + 1/Data - Offset);
```

Here's the syntax of an **inline** statement:



Each inline element consists of an optional size specifier, **<** or **>**, and a constant or a variable identifier, followed by zero or more offset specifiers (see the syntax that follows). An offset specifier consists of a **+** or a **-** followed by a constant.

inline element



Each inline element generates 1 byte or 1 word of code. The value is computed from the value of the first constant or the offset of the variable identifier, to which is added or subtracted the value of each of the constants that follow it.

An inline element generates 1 byte of code if it consists of constants only and if its value is within the 8-bit range (0..255). If the value is outside the 8-bit range or if the inline element refers to a variable, 1 word of code is generated (least-significant byte first).

The < and > operators can be used to override the automatic size selection we described earlier. If an inline element starts with a < operator, only the least-significant byte of the value is coded, even if it's a 16-bit value. If an inline element starts with a > operator, a word is always coded, even though the most-significant byte is 0. For example, the statement

```
inline (<$1234/>$44);
```

generates 3 bytes of code: $34, $44, $00.

The value of a variable identifier in an inline element is the offset address of the variable within its base segment. The base segment of global variables--variables declared at the outermost level in a program or a unit--and typed constants is the data segment, which is accessible through the DS register. The base segment of local variables--variables declared within the current subprogram--is the stack segment. In this case the variable offset is relative to the BP register, which automatically causes the stack segment to be selected. Registers BP, SP, SS, and DS must be preserved by **inline** statements; all other registers can be modified.

The following example of an **inline** statement generates machine code for storing a specified number of words of data in a specified variable. When called, procedure *FillWord* stores *Count* words of the value *Data* in memory, starting at the first byte occupied by *Dest*.

```
procedure FillWord(var Dest; Count, Data: Word);
begin
  inline (
    $C4/$BE/Dest/                   { LES DI,Dest[BP] }
    $8B/$8E/Count/                  { MOV CX,Count[BP] }
    $8B/$86/Data/                   { MOV AX,Data[BP] }
    $FC/                            { CLD }
    $F3/$AB);                       { REP STOSW }
end;
```

Inline statements can be freely mixed with other statements throughout the statement part of a block.

Inline directives

With **inline** directives, you can write procedures and functions that expand into a given sequence of machine code instructions whenever they are called. These are comparable to macros in assembly language. The syntax for an **inline** directive is the same as that of an **inline** statement:

inline directive
inline statement

When a normal procedure or function is called (including one that contains **inline** statements), the compiler generates code that pushes the parameters (if any) onto the stack, and then generates a **CALL** instruction to call the procedure or function. However, when you call an inline procedure or function, the compiler generates code from the inline directive instead of the **CALL**. Here's a short example of two inline procedures:

```
procedure DisableInterrupts; inline ($FA);      { CLI }
procedure EnableInterrupts; inline ($FB);       { STI }
```

When *DisableInterrupts* is called, it generates 1 byte of code--a **CLI** instruction.

Procedures and functions declared with inline directives can have parameters; however, the parameters can't be referred to symbolically in the inline directive (other variables can, though). Also, because such procedures and functions are in fact macros, there is no automatic entry and exit code, nor should there be any return instruction.

The following function multiplies two *Integer* values, producing a *Longint* result:

```
function LongMul(X, Y: Integer): Longint;
inline (
  $5A/                            { POP AX ;Pop X }
  $58/                            { POP DX ;Pop Y }
  $F7/$EA);                       { IMUL DX ;DX : AX = X * Y }
```

Note the lack of entry and exit code and the missing return instruction. These aren't required, because the 4 bytes are inserted into the instruction stream when *LongMul* is called.

Inline directives are intended for very short procedures and functions only (less than 10 bytes).

Because of the macro-like nature of **inline** procedures and functions, they can't be used as arguments to the @ operator and the *Addr*, *Ofs*, and *Seg* functions.

Appendix

A

The command-line compiler

Delphi's command-line compiler (DCC.EXE) lets you invoke all the functions of the IDE compiler (DELPHI.EXE) from the DOS command line. You run the command-line compiler from the DOS prompt using the following syntax:

DCC [*options*] *filename* [*options*]

options are zero or more optional parameters that provide additional information to the compiler. *filename* is the name of the source file to compile. If you type DCC alone, it displays a help screen of command-line options and syntax.

If *filename* does not have an extension, the command-line compiler assumes .PAS. If you don't want the file you're compiling to have an extension, you must append a period (.) to the end of the *filename*.

If the source text contained in *filename* is a program, the compiler creates an executable file named *filename*.EXE, and if *filename* contains a library, the compiler creates a file named *filename*.DLL. If *filename* contains a unit, the compiler creates a unit file named *filename*.DCU.

You can specify a number of options for the command-line compiler. An option consists of a slash (/) immediately followed by an option letter. In some cases, the option letter is followed by additional information, such as a number, a symbol, or a directory name. Options can be given in any order and can come before and/or after the file name.

Command-line compiler options

The IDE lets you set various options through the menus; the command-line compiler gives you access to these options using the slash (/) delimiter. You can also precede options with a hyphen (-) instead of a slash (/), but those options that start with a hyphen must be separated by blanks. For example, the following two command lines are equivalent and legal:

```
DCC -IC:\DELPHI -DDEBUG SORTNAME -$S- -$F+
DCC /IC:\DELPHI/DDEBUG SORTNAME /$S-/$F+
```

The first command line uses hyphens with at least one blank separating options; the second uses slashes and no separation is needed.

The following table lists the command-line options.

Table A-1 Command-line options

Option	Description
/$A+	Align data on word boundaries
/$A-	Align data on byte boundaries
/$B+	Complete Boolean evaluation
/$B-	Short circuit Boolean evaluation
/$D+	Debugging information on
/$D-	Debugging information off
/$E+	80x87 emulation on
/$E-	80x87 emulation off
/$F+	Force far calls on
/$F-	Force far calls off
/$G+	286 code generation on
/$G-	286 code generation off
/$I+	I/O checking on
/$I-	I/O checking off
/$K+	Smart callbacks on
/$K-	Smart callbacks off
/$L+	Local debug symbols on
/$L-	Local debug symbols off
/$M	Memory sizes
/$M+	Run-time type information on
/$M-	Run-time type information off
/$N+	Numeric coprocessor on
/$N-	Numeric coprocessor off
/$P+	Open parameters on
/$P-	Open parameters off
/$Q+	Overflow checking on
/$Q-	Overflow checking off
/$R+	Range checking on
/$R-	Range checking off
/$S+	Stack checking on
/$S-	Stack checking off
/$T+	Type-checked pointers on
/$T-	Type-checked pointers off
/$U+	Pentium safe FDIV on
/$U-	Pentium safe FDIV off
/$V+	Strict var-string checking
/$V-	Relaxed var-string checking
/$W+	Windows stack frames on
/$W-	Windows stack frames off
/$X+	Extended syntax support on

/$X-	Extended syntax support off
/$Y+	Symbol reference information on
/$Y-	Symbol reference information off
/$Z+	Word-sized enumerations
/$Z-	Byte-sized enumerations
/B	Build all units
/D*defines*	Define conditional symbol
/E*path*	EXE and DCU directory
/F*segment*: *offset*	Find run-time error
/GS	Map file with segments
/GP	Map file with publics
/GD	Detailed map file
/I*path*	Include directories
/L	Link buffer on disk
/M	Make modified units
/O*path*	Object directories
/Q	Quiet compile
/R*path*	Resource directories
/T*path*	DSL and CFG directory
/U*path*	Unit directories
/V	EXE debug information

If you type `DCC` alone at the command line, a list of command-line compiler options appears on your screen.

Compiler directive options

Delphi supports several compiler directives, all described in Appendix B, "Compiler directives." The **/$** and **/D** command-line options allow you to change the default states of most compiler directives. Using **/$** and **/D** on the command line is equivalent to inserting the corresponding compiler directive at the beginning of each source file compiled.

The switch directive option

The **/$** option lets you change the default state of all of the switch directives. The syntax of a switch directive option is **/$** followed by the directive letter, followed by a plus (**+**) or a minus (**-**). For example,

```
DCC MYSTUFF /$R-
```

compiles MYSTUFF.PAS with range-checking turned off, while

```
DCC MYSTUFF /$R+
```

compiles it with range checking turned on. Note that if a {**$R+**} or {**$R-**} compiler directive appears in the source text, it overrides the **/$R** command-line option.

You can repeat the **/$** option in order to specify multiple compiler directives:

```
DCC MYSTUFF /$R-/$I-/$V-/$F+
```

Alternately, the command-line compiler lets you write a list of directives (except for **$M**), separated by commas:

```
DCC MYSTUFF /$R-,I-,V-,F+
```

Note that only one dollar sign (**$**) is needed.

In addition to changing switch directives, **/$** also lets you specify a program's memory allocation parameters using the following format:

```
/$Mstacksize,heapsize
```

where *stacksize* is the stack size and *heapsize* is the size of the Windows local heap area in the data segment. The values are in bytes, and each is a decimal number unless it is preceded by a dollar sign (**$**), in which case it is assumed to be hexadecimal. So, for example, the following command lines are equivalent:

```
DCC MYSTUFF /$M16384,4096
DCC MYSTUFF /$M$4000,$1000
```

Note that, because of its format, you cannot use the **$M** option in a list of directives separated by commas.

The conditional defines option

The **/D** option lets you define conditional symbols, corresponding to the {**$DEFINE** *symbol*} compiler directive. The **/D** option must be followed by one or more conditional symbols separated by semicolons (;). For example, the following command line

```
DCC MYSTUFF /DIOCHECK;DEBUG;LIST
```

defines three conditional symbols, *iocheck*, *debug*, and *list*, for the compilation of MYSTUFF.PAS. This is equivalent to inserting

```
{$DEFINE IOCHECK}
{$DEFINE DEBUG}
{$DEFINE LIST}
```

at the beginning of MYSTUFF.PAS. If you specify multiple **/D** directives, you can concatenate the symbol lists. Therefore,

```
DCC MYSTUFF /DIOCHECK/DDEBUG/DLIST
```

is equivalent to the first example.

Compiler mode options

A few options affect how the compiler itself functions. These are **/M** (Make), **/B** (Build), **/F** (Find Error), **/L** (Link Buffer) and **/Q** (Quiet). As with the other options, you can use the hyphen format. Remember to separate the options with at least one blank.

The make (/M) option

The command-line compiler has a built-in MAKE utility to aid in project maintenance. The **/M** option instructs command-line compiler to check all units upon which the file being compiled depends.

A unit will be recompiled if

- The source file for that unit has been modified since the unit file was created.
- Any file included with the **$I** directive, any .OBJ file linked in by the **$L** directive, or any .RES file referenced by the **$R** directive, is newer than the unit file.
- The interface section of a unit referenced in a **uses** statement has changed.

Units in DELPHI.DSL are excluded from this process.

If you were applying this option to the previous example, the command would be

```
DCC MYSTUFF /M
```

The build all (/B) option

Instead of relying on the **/M** option to determine what needs to be updated, you can tell command-line compiler to update *all* units upon which your program depends using the **/B** option. You can't use **/M** and **/B** at the same time.

If you were using this option in the previous example, the command would be

```
DCC MYSTUFF /B
```

The find error (/F) option

When a program terminates due to a run-time error, it displays an error code and the logical segment address, (*segment*: *offset*) at which the error occurred. By specifying that address in a /**F***segment*: *offset* option, you can locate the statement in the source text that caused the error, provided your program and units were compiled with debug information enabled (via the **$D** compiler directive).

In order for the command-line compiler to find the run-time error with **/F**, you must compile the program with all the same command-line parameters you used the first time you compiled it.

As mentioned previously, you *must* compile your program and units with debug information enabled for the command-line compiler to be able to find run-time errors. By default, all programs and units are compiled with debug information enabled, but if you turn it off, using a {**$D-**} compiler directive or a **/$D-** option, the command-line compiler will not be able to locate run-time errors.

The link buffer (/L) option

The **/L** option disables buffering in memory when unit files are linked to create an .EXE file. Delphi's built-in linker makes two passes. In the first pass through the unit files, the linker marks every procedure called by other procedures. In the second pass, it generates an .EXE file by extracting the marked procedures from the unit files.

By default, the unit files are kept in memory between the two passes; however, if the **/L** option is specified, they are read again from disk during the second pass. The default method is faster but requires more memory; for very large programs, you may have to specify **/L** to link successfully.

The quiet (/Q) option

The quiet mode option suppresses the printing of file names and line numbers during compilation. When the command-line compiler is invoked with the quiet mode option

```
DCC MYSTUFF /Q
```

its output is limited to the sign-on message and the usual statistics at the end of compilation. If an error occurs, it will be reported.

Directory options

The command-line compiler supports several options that allow you to specify the six directory lists used by the command-line compiler: DSL & CFG, EXE & DCU, Include, Unit, Resource, and Object.

Excluding the EXE and DCU directory option, you can specify one or multiple directories for each command-line directory option. If you specify multiple directories, separate them with semicolons (;). For example, this command line tells the command-line compiler to search for Include files in C:\DELPHI\INCLUDE and D:\INC *after* searching the current directory:

```
DCC MYSTUFF /IC:\DELPHI\INCLUDE;D:\INC
```

If you specify multiple directives, the directory lists are concatenated. Therefore,

```
DCC MYSTUFF /IC:\DELPHI\INCLUDE /ID:\INC
```

is equivalent to the first example.

The DSL & CFG Directory (/T) option

DCC looks for two files when it is executed: DCC.CFG, the configuration file, and DELPHI.DSL, the resident library file. The command-line compiler automatically searches the current directory and the directory containing .EXE file. The **/T** option lets you specify other directories in which to search. For example, you could say

```
DCC /TC:\DELPHI\BIN MYSTUFF
```

If you want the **/T** option to affect the search for the .CFG file, it must be the very first command-line argument, as in the previous example.

The EXE & DCU directory (/E) option

This option lets you tell the command-line compiler where to put the .EXE and unit files it creates. It takes a directory path as its argument:

```
DCC MYSTUFF /EC:\DELPHI\BIN
```

You can specify only one EXE and DCU directory.

If no such option is given, the command-line compiler creates the .EXE and unit files in the same directories as their corresponding source files.

The include directories (/I) option

Delphi supports include files through the {**$I** *filename*} compiler directive. The **/I** option lets you specify a list of directories in which to search for include files.

The unit directories (/U) option

When you compile a program that uses units, the command-line compiler first attempts to find the units in the DELPHI.DSL file. If they cannot be found there, the command-line compiler searches for unit files in the current directory. The **/U** option lets you specify additional directories in which to search for units.

The resource directories (/R) option

DCC searches for resource files in the current directory. The **/R** option lets you indicate additional directories where DCC should look for resource files.

The object files directories (/O) option

Using {**$L** *filename*} compiler directives, Delphi lets you link in .OBJ files containing external assembly language routines, as explained in Chapter 20, "Linking assembler code." The **/O** option lets you specify a list of directories in which to search for such .OBJ files.

Debug options

The command-line compiler has two command-line options that enable you to generate debugging information: the map file option and the debugging option.

The map file (/G) option

The **/G** option instructs the command-line compiler to generate a .MAP file that shows the layout of the .EXE file. Unlike the binary format of .EXE and .DCU files, a .MAP file is a legible text file that can be output on a printer or loaded into the editor. The **/G** option must be followed by the letter *S*, *P*, or *D* to indicate the desired level of information in the .MAP file. A .MAP file is divided into three sections:

- Segment

- Publics
- Line Numbers

The **/GS** option outputs only the Segment section, **/GP** outputs the Segment and Publics section, and **/GD** outputs all three sections.

For modules (program and units) compiled in the {**$D+**,**L+**} state (the default), the Publics section shows all global variables, procedures, and functions, and the Line Numbers section shows line numbers for all procedures and functions in the module. In the {**$D+**,**L-**} state, only symbols defined in a unit's **interface** part are listed in the Publics section. For modules compiled in the {**$D-**} state, there are no entries in the Line Numbers section.

The debugging (/V) option

When you specify the **/V** option on the command line, the command-line compiler appends Turbo Debugger-compatible debug information at the end of the .EXE file. Turbo Debugger includes both source- and machine-level debugging and powerful breakpoints including breakpoints with conditionals or expressions attached to them.

Even though the debug information generated by **/V** makes the resulting .EXE file larger, it does not affect the actual code in the .EXE file, and if it is included, the .EXE file does not require additional memory.

The extent of debug information appended to the .EXE file depends on the setting of the **$D** and **$L** compiler directives in each of the modules (program and units) that make up the application. For modules compiled in the {**$D+**,**L+**} state, which is the default, *all* constant, variable, type, procedure, and function symbols become known to the debugger. In the {**$D+**,**L-**} state, only symbols defined in a unit's **interface** section become known to the debugger. In the {**$D-**} state, no line-number records are generated, so the debugger cannot display source lines when you debug the application.

The DCC.CFG file

You can set up a list of options in a configuration file called DCC.CFG, which will then be used in addition to the options entered on the command line. Each line in configuration file corresponds to an extra command-line argument inserted before the actual command-line arguments. Thus, by creating a configuration file, you can change the default setting of any command-line option.

The command-line compiler lets you enter the same command-line option several times, ignoring all but the last occurrence. This way, even though you've changed some settings with a configuration file, you can still override them on the command line.

When DCC starts, it looks for DCC.CFG in the current directory. If the file isn't found there, DCC looks in the directory where DCC.EXE resides. To force DCC to look in a specific list of directories (in addition to the current directory), specify a **/T** option as the first option on the command-line.

If DCC.CFG contains a line that does not start with a slash (**/**) or a hyphen (**-**), that line defines a default file name to compile. In that case, starting DCC with an empty command line (or with a command line consisting of command-line options only and no file name) will cause it to compile the default file name, instead of displaying a syntax summary.

Here's an example DCC.CFG file, defining some default directories for include, object, and unit files, and changing the default states of the **$F** and **$S** compiler directives:

```
/IC:\DELPHI\INC;C:\DELPHI\SRC
/OC:\DELPHI\ASM
/UC:\DELPHI\UNITS
/$F+
/$S-
```

Now, if you type

```
DCC MYSTUFF
```

at the system prompt, DCC acts as if you had typed in the following:

```
DCC /IC:\DELPHI\INC;C:\DELPHI\SRC /OC:\DELPHI\ASM /UC:\DELPHI\UNITS /$F+ /$S- MYSTUFF
```

Appendix B

Compiler directives

This appendix describes the compiler directives you can use to control the features of the Delphi compiler. They are listed alphabetically. Each compiler directive is classified as either a switch, parameter, or conditional compilation directive. Following the list of compiler directives is a brief discussion of how to use the conditional compilation directives. This reference section describes how to use conditional constructs and symbols to produce different code from the same source text.

A compiler directive is a comment with a special syntax. Delphi allows compiler directives wherever comments are allowed. A compiler directive starts with a **$** as the first character after the opening comment delimiter, immediately followed by a name (one or more letters) that designates the particular directive. You can include comments after the directive and any necessary parameters.

Three types of directives are described in this appendix:

- **Switch directives** turn particular compiler features on or off by specifying **+** or **-** immediately after the directive name. Switch directives are either global or local.

  Global directives affect the entire compilation and must appear before the declaration part of the program or the unit being compiled.

  Local directives affect only the part of the compilation that extends from the directive until the next occurrence of the same directive. They can appear anywhere.

  Switch directives can be grouped in a single compiler directive comment by separating them with commas with no intervening spaces. For example,

  ```
  {$B+,R-,S-}
  ```

- **Parameter directives.** These directives specify parameters that affect the compilation, such as file names and memory sizes.

- **Conditional directives.** These directives control conditional compilation of parts of the source text, based on user-definable conditional symbols. See page 242 for information about using conditional directives.

All directives, except switch directives, must have at least one blank between the directive name and the parameters. Here are some examples of compiler directives:

```
{$B+}
{$R- Turn off range checking}
{$I TYPES.INC}
{$M 32768,4096}
{$DEFINE Debug}
{$IFDEF Debug}
{$ENDIF}
```

You can insert compiler directives directly into your source code. You can also change the default directives for both the command-line compiler (DCC.EXE) and the IDE (DELPHI.EXE). The Options|Project|Compiler dialog box contains many of the compiler directives; any changes you make to the settings there will affect all subsequent compilations.

When using the command-line compiler, you can specify compiler directives on the command line (for example, `DCC /$R+ MYPROG`), or you can place directives in a configuration file (see page 222). Compiler directives in the source code always override the default values in both the command-line compiler and the IDE.

If you are working in the Delphi editor and want a quick way to see what compiler directives are in effect, press Ctrl+O O. Delphi will insert the current settings in the edit window at the top of your file.

Align data

Type	Switch
Syntax	`{$A+}` or `{$A-}`
Default	`{$A+}`
Scope	Global

Remarks

The **$A** directive switches between byte and word alignment of variables and typed constants. On all 80x86 CPUs, word alignment means faster execution because word-sized items on even addresses are accessed in one memory cycle rather than two memory cycles for words on odd addresses.

In the {**$A+**} state, all variables and typed constants larger than one byte are aligned on a machine-word boundary (an even-numbered address). If required, unused bytes are inserted between variables to achieve word alignment. The {**$A+**} directive does not affect byte-sized variables, nor does it affect fields of record structures and elements of arrays. A field in a record will align on a word boundary only if the total size of all fields before it is even. For every element of an array to align on a word boundary, the size of the elements must be even.

In the {**$A-**} state, no alignment measures are taken. Variables and typed constants are simply placed at the next available address, regardless of their size.

Regardless of the state of the **$A** directive, each global **var** and **const** declaration section always starts at a word boundary. The compiler always keeps the stack pointer (SP) word aligned by allocating an extra unused byte in a procedure's stack frame if required.

Boolean evaluation

Type	Switch
Syntax	`{$B+}` or `{$B-}`
Default	`{$B-}`
Scope	Local

Remarks

The **$B** directive switches between the two different models of code generation for the **and** and **or** Boolean operators.

In the {**$B+**} state, the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression built from the **and** and **or** operators is guaranteed to be evaluated, even when the result of the entire expression is already known.

In the {**$B-**} state, the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident.

For further details, see the section "Boolean operators" in Chapter 5, "Expressions."

Code segment attribute

Type	Parameter
Syntax	`{$C attribute attribute ...}`
Default	`{$C MOVEABLE DEMANDLOAD DISCARDABLE}`
Scope	Global

Remarks

The **$C** directive is used to control the attributes of a code segment. Every code segment in an application or library has a set of attributes that determine the behavior of the code segment when it is loaded into memory. For example, you can specify that a code segment is *moveable*, which means Windows can move the code segment around in memory as needed, or you can specify that a code segment is *fixed*, which means the location of the code segment in memory cannot change.

A **$C** directive affects only the code segment of the module (unit, program, or library) in which it is placed. For more information, see "Code Segments" in Chapter 16. In the following table, the code-segment attribute options occur in groups of two; each option has an opposite toggle. Here are the grouped options:

MOVEABLE	The system can change location of code segment in linear memory.
FIXED	The system can't change location of code segment in linear memory.

DEMANDLOAD	Code segment loads only when it is needed.
PRELOAD	Code segment loads when the program begins executing.

DISCARDABLE	Code segment can be unloaded when it's no longer needed.
PERMANENT	Code segment remains in memory once it is loaded.

The first option of each group is the default. You may specify multiple code segment attributes using the **$C** directive. If both options of a group in a **$C** directive are specified, only the last one will take effect. For example,

```
{$C FIXED MOVEABLE DISCARDABLE }
```

will make the code segment moveable, and it can be discarded when it is no longer needed.

Debug information

Type	Switch
Syntax	`{$D+}` or `{$D-}`
Default	`{$D+}`
Scope	Global

Remarks

The **$D** directive enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object-code addresses into source text line numbers.

For units, the debug information is recorded in the unit file along with the unit's object code. Debug information increases the size of unit file and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the {**$D+**} state, Delphi's integrated debugger lets you single-step and set breakpoints in that module.

The TDW Debug Info (Options | Project) and Map File (Options | Project) options produce complete line information for a given module only if you've compiled that module in the {**$D+**} state.

The **$D** switch is usually used in conjunction with the **$L** switch, which enables and disables the generation of local symbol information for debugging.

If you want to use Turbo Debugger to debug your program, choose Options | Project, and check the TDW Debug Info option.

DEFINE directive

Type	Conditional compilation
Syntax	`{$DEFINE name}`

Remarks

Defines a conditional symbol with the given *name*. The symbol is recognized for the remainder of the compilation of the current module in which the symbol is declared, or until it appears in an {**$UNDEF** *name*} directive. The {**$DEFINE** *name*} directive has no effect if *name* is already defined.

Description

Type	Parameter
Syntax	`{$D text}`
Scope	Global

Remarks

The **$D** directive inserts the text you specify into the module description entry in the header of a .EXE or .DLL.Traditionally the text is a name and version number, but you may specify any text of your choosing. For example,

```
{$D My Application version 12.5}
```

ELSE directive

Type	Conditional compilation
Syntax	`{$ELSE}`

Remarks

Switches between compiling and ignoring the source text delimited by the last {**$IF***xxx*} and the next {**$ENDIF**}.

ENDIF directive

Type Conditional compilation

Syntax `{$ENDIF}`

Remarks

Ends the conditional compilation initiated by the last {**$IF***xxx*} directive.

Extended syntax

Type Switch

Syntax `{$X+}` or `{$X-}`

Default `{$X+}`

Scope Global

Remarks

The **$X** directive enables or disables Delphi's extended syntax:

- Function statements. In the **{$X+}** mode, function calls can be used as procedure calls; that is, the result of a function call can be discarded. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. However, in certain cases a function can carry out multiple operations based on its parameters and some of those cases might not produce a useful result. When that happens, the **{$X+}** extensions allow the function to be treated as a procedure.
- Null-terminated strings. A **{$X+}** compiler directive enables Delphi's support for null-terminated strings by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays. For more details about null-terminated strings, see Chapter 15, "Using null-terminated strings."

Note The **{$X+}** directive does not apply to built-in functions (those defined in the *System* unit).

Force Far calls

Type Switch

Syntax `{$F+}` or `{$F-}`

Default `{$F-}`

Scope Local

Remarks

The **$F** directive determines which call model to use for subsequently compiled procedures and functions. Procedures and functions compiled in the {**$F+**} state always use the far call model. In the {**$F-**} state, Delphi automatically selects the appropriate model: far if the procedure or function is declared in the **interface** section of a unit; otherwise it is near.

The near and far call models are described in full in Chapter 17, "Control issues."

Generate 80286 Code

Type Switch

Syntax `{$G+}` or `{$G-}`

Default `{$G+}`

Scope Global

Remarks

The **$G** directive enables or disables 80286 code generation. In the {**$G-**} state, only generic 8086 instructions are generated, and programs compiled in this state can run on any 80x86 family processor. You can specify {**$G-**} any place within your code.

In the {**$G+**} state, the compiler uses the additional instructions of the 80286 to improve code generation, but programs compiled in this state cannot run on 8088 and 8086 processors. Additional instructions used in the {**$G+**} state include **ENTER**, **LEAVE**, **PUSH** immediate, extended **IMUL**, and extended **SHL** and **SHR**.

Group unit segments

Type Parameter

Syntax `{$G unitname unitname ...}`

Scope Global

Remarks

The **$G** directive lets you specify groups of units you want the linker to place in the same segment. Grouping units in the same segment ensures that the units swap in and out of memory at the same time. The **$G** directive is primarily used to group units containing discardable code.

Each **$G** directive specifies a group of units. **$G** directives are valid only in a program or library, and must appear after the program or library's **uses** clause. The compiler reports an error if you attempt to add a unit to more than one group. In addition to any groups created with **$G**, the compiler maintains a default group that includes all units not explicitly grouped.

Code segment attributes are controlled by the $C directive. The preferred segment size is set with $S.

The linker minimizes the number of code segments in an executable file by combining all units that belong to the same group. Two or more units are put into the same code segment if they belong to the same group and have the same code segment attributes, and if the combined size does not exceed the preferred segment size.

The linker will *never* put units that belong to different groups in the same code segment.

IFDEF directive

Type	Conditional compilation
Syntax	`{$IFDEF name}`

Remarks

Compiles the source text that follows it if *name* is defined.

IFNDEF directive

Type	Conditional compilation
Syntax	`{$IFNDEF name}`

Remarks

Compiles the source text that follows it if *name* is not defined.

IFOPT directive

Type	Conditional compilation
Syntax	`{$IFOPT switch}`

Remarks

Compiles the source text that follows it if *switch* is currently in the specified state. *switch* consists of the name of a switch option, followed by a **+** or a **-** symbol. For example, the construct

```
{$IFOPT N+}
  type Real = Extended;
```

```
{$ENDIF}
```

will compile the type declaration if the **$N** option is currently active.

Include file

Type	Parameter
Syntax	`{$I filename}`
Scope	Local

Remarks

The **$I** parameter directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text right after the {**$I** *filename*} directive. The default extension for *filename* is .PAS. If *filename* does not specify a directory path, then, in addition to searching for the file in the same directory as the current module, Delphi searches in the directories specified in the Search path input box on the Directories/Conditionals page of the Options|Project dialog (or in the directories specified in a **/I** option on the DCC command line).

There is one restriction to the use of include files: An include file can't be specified in the middle of a statement part. In fact, all statements between the **begin** and **end** of a statement part must exist in the same source file.

Input/output checking

Type	Switch
Syntax	`{$I+}` or `{$I-}`
Default	`{$I+}`
Scope	Local

Remarks

The **$I** switch directive enables or disables the automatic code generation that checks the result of a call to an I/O procedure. I/O procedures are described in Chapter 13, "Input and output." If an I/O procedure returns a nonzero I/O result when this switch is on, an *EInOutError* exception is raised (or the program is terminated if exception handling is not enabled). When this switch is off, you must check for I/O errors by calling *IOResult*.

Link object file

Type	Parameter
Syntax	`{$L filename}`

Scope	Local

Remarks

The **$L** parameter instructs the compiler to link the named file with the program or unit being compiled. The **$L** directive is used to link with code written in for subprograms declared to be **external**. The named file must be an Intel relocatable object file (.OBJ file). The default extension for *filename* is .OBJ. If *filename* does not specify a directory path, then, in addition to searching for the file in the same directory as the current module, Delphi searches in the directories specified in the Search path input box on the Directories/Conditionals page of the Options|Project dialog (or in the directories specified in a **/O** option on the DCC command line). For further details about linking with assembly language, see Chapter 20, "Linking assembler code."

Local symbol information

Type	Switch
Syntax	`{$L+}` or `{$L-}`
Default	`{$L+}`
Scope	Global

Remarks

The **$L** switch directive enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module's implementation part and the symbols within the module's procedures and functions.

For units, the local symbol information is recorded in the unit file along with the unit's object code. Local symbol information increases the size of unit files and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the {**$L+**} state, Delphi's integrated debugger lets you examine and modify the module's local variables. Furthermore, calls to the module's procedures and functions can be examined via the View|Call Stack.

The Include TDW debug info and Map file options on the Linker page of the Options|Project dialog produce local symbol information for a given module only if that module was compiled in the {$**L+**} state.

The **$L** switch is usually used in conjunction with the **$D** switch, which enables and disables the generation of line-number tables for debugging. The **$L** directive is ignored if the compiler is in the {**$D-**} state.

Memory allocation sizes

Type	Parameter
Syntax	`{$M stacksize,heapsize}`
Default	`{$M 16384,8192}`
Scope	Global

Remarks

The **$M** directive specifies an application or library's memory allocation parameters. *stacksize* must be an integer number in the range 1,024 to 65,520 which specifies the size of the stack segment. *Heapsize* specifies the size of the local heap area in the data segment. *heapsize* must be an integer number in the range 0 to 65520.

The **$M** directive has no effect when used in a unit. Furthermore, the *stacksize* parameter in a **$M** directive is ignored in a library (a library always uses the stack of the applications that call it).

Numeric coprocessor

Type	Switch
Syntax	`{$N+}` or `{$N-}`
Default	`{$N+}`
Scope	Global

Remarks

The **$N** directive switches between the two different models of floating-point code generation supported by Delphi. In the {**$N-**} state, code is generated to perform all real-type calculations in software by calling run-time library routines. In the {**$N+**} state, code is generated to perform all real-type calculations using the 80x87 numeric coprocessor.

Open String Parameters

Type	Switch
Syntax	`{$P+}` or `{$P-}`
Default	`{$P+}`
Scope	Local

Remarks

The **$P** directive controls the meaning of variable parameters declared using the **string** keyword. In the **{$P-}** state, variable parameters declared using the **string** keyword are normal variable parameters, but in the **{$P+}** state, they are open string parameters. Regardless of the setting of the **$P** directive, the *OpenString* identifier can always be used to declare open string parameters. For more information about open parameters, see Chapter 8, "Procedures and functions."

Overflow checking

Type	Switch
Syntax	`{$Q+}` or `{$Q-}`
Default	`{$Q-}`
Scope	Local

Remarks

The **$Q** directive controls the generation of overflow checking code. In the {**$Q+**} state, certain integer arithmetic operations (+, -, *, *Abs*, *Sqr*, *Succ*, and *Pred*) are checked for overflow. The code for each of these integer arithmetic operations is followed by additional code that verifies that the result is within the supported range. If an overflow check fails, an *EIntOverflow* exception is raised (or the program is terminated if exception handling is not enabled).

The {**$Q+**} does not affect the *Inc* and *Dec* standard procedures. These procedures are never checked for overflow.

The **$Q** switch is usually used in conjunction with the **$R** switch, which enables and disables the generation of range-checking code. Enabling overflow checking slows down your program and makes it somewhat larger, so use {**$Q+**} only for debugging.

Pentium safe FDIV operations

Type:	Switch
Syntax:	{$U+} or {$U-}
Default:	{$U+}
Scope:	Local

The **$U** directive controls generation of floating-point code that guards against the flawed FDIV instruction exhibited by certain early Pentium processors.

In the {**$U+**} state, all floating-point divisions are performed using a run-time library routine. The first time the floating-point division routine is invoked, it checks whether the processor's FDIV instruction works correctly, and updates the *TestFDIV* variable (declared in the *System* unit) accordingly. For subsequent floating-point divide operations, the value stored in *TestFDIV* is used to determine what action to take.

Table B-1 TestFDIV values

Value	Meaning
-1	FDIV instruction has been tested and found to be flawed.
0	FDIV instruction has not yet been tested.
1	FDIV instruction has been tested and found to be correct.

For processors that do not exhibit the FDIV flaw, {**$U+**} results in only a slight performance degredation. For a flawed Pentium processor, floating-point divide operations may take up to three times longer in the {**$U+**} state, but they will always produce correct results.

In the {**$U-**} state, floating-point divide operations are performed using in-line FDIV instructions. This results in optimum speed and code size, but may produce incorrect results on flawed Pentium processors. You should use the {**$U-**} state only in cases where you are certain that the code is not running on a flawed Pentium processor.

Range checking

Type Switch

Syntax `{$R+}` or `{$R-}`

Default `{$R-}`

Scope Local

Remarks

The **$R** directive enables or disables the generation of range-checking code. In the **{$R+}** state, all array and string-indexing expressions are verified as being within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, an *ERangeError* exception is raised (or the program is terminated if exception handling is not enabled).

Enabling range checking slows down your program and makes it somewhat larger, so use the {**$R+**} only for debugging.

Resource file

Type Parameter

Syntax `{$R Filename}`

Scope Local

Remarks

The **$R** directive specifies the name of a resource file to be included in an application or library. The named file must be a Windows resource file and the default extension for *filename* is .RES.

When a {**$R** filename} directive is used in a unit, the specified filename is simply recorded in the resulting unit file. No checks are made at that point to ensure that the filename is correct and that it specifies an existing file.

When an application or library is linked (after compiling the program or library source file), the resource files specified in all used units as well as in the program or library itself are processed and, each resource in each resource file is copied to the .EXE or .DLL being produced. During the resource processing phase, Delphi's linker searches for .RES files in the same directory as the module containing the $**R** directive, and in the directories specified in the Search path input box on the Directories/Conditionals page of the Options|Project dialog (or in the directories specified in a **/R** option on the DCC command line).

Run-time type information

Type: Switch

Syntax: {$M+} or {$M-}

Default: {$M-}

Scope: Local

The **$M** switch directive controls generation of run-time type information. When a class is declared in the {**$M+**} state, or is derived from a class that was declared in the {**$M+**} state, the compiler generates run-time type information for fields, methods, and properties that are declared in a **published** section. If a class is declared in the {**$M–**} state, and is not derived from a class that was declared in the {**$M+**} state, **published** sections are not allowed in the class.

Note The *TPersistent* class defined in the *Classes* unit of the Delphi Visual Class Library was declared in the {**$M+**} state, so any class derived from *TPersistent* is allowed to contain **published** sections. The Delphi Visual Class Library uses the run-time type information generated for **published** sections to access the values of a component's properties when saving a loading form files. Furthermore, the Delphi IDE uses a component's run-time type information to determine the list of properties to show in the Object Inspector.

There is seldom, if ever, any need for an application to directly use the **$M** compiler switch.

Segment size preference

Type Parameter

Syntax `{$S segsize}`

Default	`{$S 16384}`
Scope	Global

Remarks

The **$S** parameter directive is valid only in a main program or library. The directive specifies the preferred size of code segments for grouped units. The specified size must be in the range 0..65,535. Units that exceed the specified size are placed in their own code segments.

When grouping units, the linker puts units with the same code segment attributes into the same code segment, up to the size specified. The limit also applies to groups specified by the **$G** directive. Grouping of units is explained under the **$G** directive.

The **$S** directive never produces warnings or error messages. If a unit can't fit into a code segment with other units, it automatically is placed into a separate segment.

Setting the preferred segment size to 0 guarantees that every unit goes in a separate code segment; this was the default behavior in previous versions of the compiler.

Smart callbacks

Type	Switch
Syntax	`{$K+}` or `{$K-}`
Default	`{$K+}`
Scope	Global

Remarks

The **$K** directive controls the generation of *smart callbacks* for procedures and functions that are exported by an application. When an application is compiled in the {**$K-**} state, it must use the *MakeProcInstance* and *FreeProcInstance* Windows API routines when it creates callback routines. In the default {**$K+**} state, the application itself can call exported entry points, and there is no need to use *MakeProcInstance* and *FreeProcInstance*.

For more details about smart callbacks, see "Entry and exit code" in Chapter 17.

Stack-overflow checking

Type	Switch
Syntax	`{$S+}` or `{$S-}`
Default	`{$S+}`
Scope	Local

Remarks

The **$S** directive enables or disables the generation of stack-overflow checking code. In the {**$S+**} state, the compiler generates code at the beginning of each procedure or function that checks whether there is sufficient stack space for the local variables and other temporary storage. When there is not enough stack space, a call to a procedure or function compiled with {**$S+**} an *EStackFault* exception to be raised (or it causes the program to be terminated if exception handling is not enabled). In the {**$S-**} state, such a call is likely to cause a system crash.

Symbol reference information

Type	Switch
Syntax	`{$Y+}` or `{$Y-}`
Default	`{$Y+}`
Scope	Global

Remarks

The **$Y** directive enables or disables generation of symbol reference information. This information consists of tables that provide the line numbers of all declarations of and references to symbols in a module.

For units, the symbol reference information is recorded in the .DCU file along with the unit's object code. Symbol reference information increases the size of the .DCU files, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the **{$Y+}** state, Delphi's integrated browser can display symbol definition and reference information for that module.

The **$Y** switch is usually used in conjunction with the **$D** and **$L** switches, which control generation of debug information and local symbol information. The **$Y** directive has no effect unless both **$D** and **$L** are enabled.

Type-checked pointers

Type	Switch
Syntax	`{$T+}` or `{$T-}`
Default	`{$T-}`
Scope	Global

Remarks

The **$T** directive controls the types of pointer values generated by the @ operator. In the {**$T-**} state, the result type of the @ operator is always an untyped pointer that is compatible with all other pointer types. When @ is applied to a variable reference in the {**$T+**} state, the type of the result is ^*T*, where *T* is compatible only with other pointers to the type of the variable.

UNDEF directive

Type Conditional compilation

Syntax `{$UNDEF name}`

Remarks

Undefines a previously defined conditional symbol. The symbol is forgotten for the remainder of the compilation or until it reappears in a {**$DEFINE** *name*} directive. The {**$UNDEF** *name*} directive has no effect if *name* is already undefined.

Var-string checking

Type Switch

Syntax `{$V+}` or `{$V-}`

Default `{$V+}`

Scope Local

Remarks

The **$V** directive controls type checking on strings passed as variable parameters. In the {**$V+**} state, strict type checking is performed, requiring the formal and actual parameters to be of *identical* string types. In the {**$V-**} (relaxed) state, any string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

The {**$V-**} state essentially provides an "unsafe" version of open string parameters. Although {**$V-**} is still supported, you should use open string parameters. For additional information, see "Open string parameters" in Chapter 8.

Windows stack frames

Type Switch

Syntax `{$W+}` or `{$W-}`

Default `{$W-}`

Scope Local

Remarks

The **$W** directive controls the generation of Windows-specific procedure entry and exit code for **far** procedures and functions. In the {**$W+**} state, special entry and exit code is generated for **far** procedures and functions. Some debugging tools require this special entry and exit code to correctly identify **far** call stack frames.

In the {**$W-**} state, no special entry and exit code is generated for **far** procedures and functions. This is the recommended state for final applications.

See Chapter 17 for additional information.

Word sized enumeration types

Type	Switch
Syntax	`{$Z+}` or `{$Z-}`
Default	`{$Z-}`
Scope	Local

Remarks

The **$Z** directive controls the storage size of enumerated types. An enumerated type declared in the the {**$Z+**} state is always stored as a word. An enumerated type declared in the {**$Z–**} state is stored as a byte if the type has no more than 256 values; otherwise it is stored as a word. The {**$Z+**} state is useful for interfacing with C and C++ libraries, which usually represent enumerated types as words.

Using conditional compilation directives

Two basic conditional compilation constructs closely resemble Pascal's **if** statement. The first construct

```
{$IFxxx}
   ⋮
{$ENDIF}
```

causes the source text between {**$IF***xxx*} and {**$ENDIF**} to be compiled only if the condition specified in {**$IF***xxx*} is *True*. If the condition is *False*, the source text between the two directives is ignored.

The second conditional compilation construct

```
{$IFxxx}
   ⋮
{$ELSE}
   ⋮
{$ENDIF}
```

causes either the source text between {**$IF***xxx*} and {**$ELSE**} or the source text between {**$ELSE**} and {**$ENDIF**} to be compiled, depending on the condition specified by the {**$IF***xxx*}.

Here are some examples of conditional compilation constructs:

```
{$IFDEF Debug}
  Writeln('X = ', X);
{$ENDIF}

{$IFDEF CPU87}
  {$N+}
  type
    Real = Double;
{$ELSE}
  {$N-}
  type
    Single = Real;
    Double = Real;
    Extended = Real;
    Comp = Real;
{$ENDIF}
```

You can nest conditional compilation constructs up to 16 levels deep. For every {**$IF***xxx*}, the corresponding {**$ENDIF**} must be found within the same source file--which means there must be an equal number of {**$IF***xxx*}'s and {**$ENDIF**}'s in every source file.

Conditional symbols

Conditional compilation is based on the evaluation of conditional symbols. Conditional symbols are defined and undefined using the directives

```
{$DEFINE name}
{$UNDEF name}
```

You can also use the **/D** switch in the command-line compiler to define a symbol (or add the symbol to the Conditional Defines input box on the Directories/Conditionals page of the **O**ptions|**P**roject dialog box in the IDE).

Conditional symbols are best compared to Boolean variables: They are either *True* (defined) or *False* (undefined). The {**$DEFINE**} directive sets a given symbol to *True*, and the {**$UNDEF**} directive sets it to *False*.

Conditional symbols follow the same rules as Pascal identifiers: They must start with a letter, followed by any combination of letters, digits, and underscores. They can be of any length, but only the first 63 characters are significant.

Conditional symbols and Pascal identifiers have no correlation whatsoever. Conditional symbols cannot be referenced in the actual program and the program's identifiers cannot be referenced in conditional directives. For example, the construct

```
const
```

```
  Debug = True;
begin
  {$IFDEF Debug}
  Writeln('Debug is on');
  {$ENDIF}
end;
```

will *not* compile the *Writeln* statement. Likewise, the construct

```
{$DEFINE Debug}
begin
  if Debug then
    Writeln('Debug is on');
end;
```

will result in an unknown identifier error in the **if** statement.

Delphi defines the following standard conditional symbols:

VER80 Always defined, indicating that this is version 8.0 of Delphi. Each version has corresponding predefined symbols; for example, version 9.0 would have *VER90* defined, version 9.5 would have *VER95* defined, and so on.

WINDOWS Indicates that the operating environment is MS-Windows.

CPU86 Always defined, indicating that the CPU belongs to the 80x86 family of processors. Versions of Delphi for other CPUs will instead define a symbolic name for that particular CPU.

CPU87 Defined if an 80x87 numeric coprocessor is present at compile time.

Other conditional symbols can be defined before a compilation by using the Conditional Defines input box, or the **/D** command-line option if you are using the command-line compiler.

Appendix

C

Error Messages

This chapter lists the possible error messages you can get from Delphi. The error messages are grouped into two categories, *compiler* errors and *run-time* errors.

Compiler error messages

Whenever possible, the compiler will display additional diagnostic information in the form of an identifier or a file name. For example,

```
Error 15: File not found (TEST.DCU)
```

When an error is detected, Delphi automatically loads the source file and places the cursor at the error. The command-line compiler displays the error message and number and places the cursor at the point in the source line where the error occurred. Note, however, that some errors are not detected until a little later in the source text. For example, a type mismatch in an assignment statement cannot be detected until the entire expression after the := has been evaluated. In such cases, look for the error to the left of or above the cursor.

1 Out of memory

This error occurs when the compiler has run out of memory.

Try these possible solutions:

- Increase the amount of available memory in Windows by closing other Windows applications or by increasing the swap file size.
- Set the Link Buffer option to Disk on the Linker page of the Options | Project dialog box.
- If you are using the command-line compiler, use the /L option to place the link buffer on disk.

If none of these suggestions help, your program or unit might be too large to compile in the amount of memory available; you might have to break it into two or more smaller units.

2 Identifier expected

An identifier was expected at this point.

You may be trying to redeclare a reserved word or standard directive as a variable.

3 Unknown identifier

This identifier has not been declared, or it may not be visible within the current scope.

4 Duplicate identifier

Within the current scope, the identifier you are declaring already represents a program's name, a constant, a variable, a type, a procedure or a function.

5 Syntax error

An illegal character was found in the source text.

You may have omitted the quotes around a string constant.

6 Error in real constant

The syntax of your real-type constant is invalid. Check to make sure that the assigned value is within the range of a real type. For more information, see Chapter 3.

Note Whole real numbers outside the maximum integer range must be followed by a decimal point and a zero, like this:

```
12345678912.0
```

7 Error in integer constant

The syntax of your integer-type constant is invalid. Check to make sure that the assigned value is within the range of a integer type. For more information, see Chapter 3.

Note Whole real numbers outside the maximum integer range must be followed by a decimal point and a zero, like this:

```
12345678912.0
```

8 String constant exceeds line

You have probably omitted the ending quote in a string constant or your quoted string runs past the line limit of 127 characters. Note that string constants must be declared on a single line.

10 Unexpected end of file

The most likely causes of this error message are as follows:

- Your source file ends before the final **end** of the main statement part. The begins and ends are probably unbalanced.
- An Include file ends in the middle of a statement part. (Every statement part must be entirely contained in one file.)
- You did not close a comment.

11 Line too long

The maximum source-code line length is 127 characters.

12 Type identifier expected

The identifier which you are declaring does not denote a type. For more information, "**Identifiers**" on page 5.

13 Too many open files

You have exceeded the maximum number of open files. Your CONFIG.SYS file does not include a FILES=xx entry, or the entry specifies too few files.

Increase the number of FILES or close some open files.

14 Invalid file name

The file name is invalid or specifies a nonexistent path.

15 File not found

The file could not be found in the current directory or in any of the search directories that apply to this type of file.

16 Disk full

The disk on which you are compiling this project is out of space. You must delete some files or use a different disk.

Note that the symbolic information used by Turbo Debugger and the Browser can be quite large. If you are not debugging or browsing your files, you can turn these options off to save disk space.

17 Invalid compiler directive

One of the following applies:

- The compiler directive letter is unknown.
- One of the compiler directive parameters is invalid.

- You are using a global compiler directive in the body of the source-code module.

For more information on compiler directives, see Appendix B.

18 Too many files

There are too many files involved in the compilation of the program or unit.

Try to reduce the number of files by merging Include files, by merging unit files, or by making the file names shorter. Alternately, you can move all the files into one directory and make it the current directory at compile time.

19 Undefined type in pointer definition

The type was referenced in a pointer-type declaration or a class reference, but never declared in that same block. The referenced type must be declared in the same type block as the pointer or class reference definition. For more information on pointer types, see page 21.

20 Variable identifier expected

The identifier does not denote a variable. For more information on identifiers, see page 5.

21 Error in type

This symbol cannot start a type definition.

22 Structure too large

The maximum allowable size of a structured type is 65520 bytes. In addition, the maximum array size is 65520. Multiply the size of the base type with the number of array elements to obtains the actual size of the array. To manage larger data structures, use *TList* objects or pointers.

23 Set base type out of range

The base type of a set must be a subrange between 0 and 255, or an enumerated type with no more than 256 possible values. For more information on set-type ranges, "**Set types**" on page20.

24 File components cannot be files or objects

The component type of a file type cannot be any of the following:

- an object type
- a file type
- a file of any structured type that includes a file-type component.

25 Invalid string length

The declared maximum length of a string must be between 1 and 255. For more information on strings, see "**String types**" on page 17.

26 Type mismatch

This error occurs due to one of the following:

- Incompatible types of the variable and expression in an assignment statement.
- Incompatible types of the actual and formal parameter in a call to a procedure or function.
- An expression type that is incompatible with the index type in array indexing.
- Incompatible operand types in an expression.
- Your typecast is incorrect.

For more information, see "Type Compatibility" on page 25.

27 Invalid subrange base type

Only ordinal types are valid base types. For a listing on the Object Pascal ordinal types, see page 12.

28 Lower bound greater than upper bound

The declaration of a subrange type must specify a lower bound less than the upper bound.

29 Ordinal type expected

Real types, string types, structured types, and pointer types are not allowed here. For a listing on the Object Pascal ordinal types, see page 12.

30 Integer constant expected

Only an integer constant is allowed here. For more information on constants, see Chapter 2.

31 Constant expected

Only a constant is allowed here. For more information on constants, see Chapter 2.

32 Integer or real constant expected

Only a numeric constant is allowed here. For more information on constants, see Chapter 2.

33 Pointer type identifier expected

The identifier does not denote a pointer type. For more information on pointer types, see “Pointer Types” on page 21.

34 Invalid function result type

File types are not valid function result types. For more information, see page 74.

35 Label identifier expected

The identifier does not denote a label. For more information about labels, see page 6.

36 BEGIN expected

A **BEGIN** was expected here, or there is an error in the block structure of the unit or program.

37 END expected

An **END** was expected here, or there is an error in the block structure of the unit or program.

38 Integer expression expected

The expression must be of an Integer type. For more information on integer types page 12.

39 Ordinal expression expected

The expression must be of an ordinal type. For more information on ordinal types, see page 12.

40 Boolean expression expected

The expression must be of type Boolean. For more information on Boolean expressions, see page 45.

The compiler can also generate this error if you forget to include an operator in the conditional expression of an **if** statement.

41 Operand types do not match operator

The operator cannot be applied to operands of this type; for example, 'A' **div** '2'.

42 Error in expression

This symbol cannot participate in an expression in the way it is written.

You may have forgotten to write an operator between two operands.

43 Illegal assignment

Files and untyped variables cannot be assigned values (you must typecast untyped variables to be able to assign values to them).

A function identifier can only be assigned values within the statement part of the function.

44 Field identifier expected

The identifier does not denote a field in the record variable. For more information on record types, see page 19.

45 Object file too large

Delphi cannot link in .OBJ files larger than 64K.

46 Undefined external

The **external** procedure or function does not have a matching **PUBLIC** definition in an object file.

Make sure you have specified all object files in **$L** filename directives, and checked the spelling of the procedure or function identifier in the .ASM file.

47 Invalid object file record

The .OBJ file contains an invalid object record; make sure the file is an .OBJ file.

There are many reasons why the compiler might generate this error message; however, all the reasons relate to an object file that does not conform to the Delphi object-file requirements. For example, Delphi does not support linking 32-bit flat memory model object files.

48 Code segment too large

The maximum code size of a program or unit is 65520 bytes.

- If you are compiling a program, move some procedures or functions into a unit.
- If you are compiling a unit, break it into two or more units.

49 Data segment too large

The maximum size of a program's data segment is 65520 bytes, including data declared by the units in the program. Note that in Windows programs, both the stack and the local heap "live" in the data segment.

If you need more global data than this, declare larger structures as pointers, and allocate them dynamically using the *New* procedure.

Note that old model objects (as supported by Borland Pascal 7) store their virtual method tables in the data segment. The new model classes do not. Also, *PChar* string constants are stored in the data segment. Pascal string constants are not.

If you are using the old model objects, try using the new method classes declaration to reduce the virtual methods used in your program. In addition, try moving *PChar* string constants into a string table resource.

50 DO expected

The reserved word **DO** does not appear where it should.

51 Invalid PUBLIC definition

Here are some possible sources of this error:

- Two or more **PUBLIC** directives in assembly language define the same identifier.
- The .OBJ file defines **PUBLIC** symbols that do not reside in the **CODE** segment.

52 Invalid EXTRN definition

Here are some possible sources of this error:

- The identifier was referred to through an **EXTRN** directive in assembly language, but it is not declared in the unit, nor in the **interface** part of any of the used units.
- The identifier denotes an absolute variable.
- The identifier denotes an inline procedure or function.

53 Too many EXTRN definitions

Delphi cannot handle .OBJ files with more than 256 **EXTRN** definitions per object file.

54 OF expected

The reserved word **OF** does not appear where it should in the **case** statement.

55 INTERFACE expected

The reserved word **INTERFACE** does not appear where it should; the reserved word **INTERFACE** is missing or declarations appear before the **INTERFACE** reserved word.

56 Invalid relocatable reference

A relocatable reference is usually a pointer to a procedure or function or another type of reference to the code segment.

Here are some possible sources of this error:

- The .OBJ file contains data and relocatable references in segments other than **CODE**. For example, you may be attempting to declare initialized variables in the **DATA** segment.
- The .OBJ file contains byte-sized references to relocatable symbols. This error occurs if you use the **HIGH** and **LOW** operators with relocatable symbols, or if you refer to relocatable symbols in DB directives.
- An operand refers to a relocatable symbol that was not defined in the **CODE** segment or in the **DATA** segment.
- An operand refers to an **EXTRN** procedure or function with an offset; for example:

```
CALL SortProc+8
```

57 THEN expected

The reserved word **THEN** does not appear where it should.

58 TO or DOWNTO expected

The reserved word **TO** or **DOWNTO** does not appear in the **for** loop.

59 Undefined FORWARD

Here are some possible sources of this error:

- The procedure or function was declared in the interface part of a unit, but its body never occurred in the implementation part.
- The procedure or function was declared forward but its definition was never found.

61 Invalid typecast

Here are some possible sources of this error:

- In a variable typecast, the sizes of the variable reference and the destination type differ.
- You are attempting to typecast an expression where only a variable reference is allowed.
- For more information on variable typecasting, see page 34.

62 Division by zero

You are attempting to divide using a constant expression that evaluates to zero. The compiler can generate this error only when you use a constant expression as the divisor of a divide operator, and that constant expression evaluates to zero.

63 Invalid file type

The file-handling procedure does not support the given file's type.

For example, you might have made a call to *Readln* with a typed file or *Seek* with a text file.

64 Cannot read or write variables of this type

Reading:

Read and *Readln* can input these variables:

- character
- integer
- real
- string

Writing:

Write and *Writeln* can output these variables:

- character
- integer
- real
- string
- Boolean

65 Pointer variable expected

This variable must be of a pointer type. For more information on pointer types, see page 21.

66 String variable expected

This variable must be of a string type. For more information about string types, see page 17.

67 String expression expected

This expression must be of a string type. For more information about string types, see page 17.

68 Circular unit reference

Two units reference each other in their interface parts. Move the cross reference of at least one of the units into the implementation section of that unit (rearrange your **uses** clauses so that at least one of the circular references occurs in the implementation part of the unit). Note that it is legal for two units to use each other in their implementation parts.

69 Unit name mismatch

The name of the disk file does not match the name declared inside that unit. Check the spelling of the unit name.

70 Unit version mismatch

A unit used by this project was compiled with an earlier version of the compiler. Make sure the source file for the specified unit is on the compiler search path, then use Compile | Build All to automatically recompile the units that have changed.

If you don't have the source file for the offending DCU unit, contact the author of the unit for an update.

71 Internal stack overflow

The compiler's internal stack is exhausted due to too many levels of nested statements.

You must rearrange your code to remove some levels of nesting.

For example, move the inner levels of nested statements into a separate procedure.

72 Unit file format error

The .DCU file is invalid; make sure it is a .DCU file.

The .DCU file may have been created with a previous version of Delphi. In this case, you must recompile the corresponding .PAS file to create a new .DCU file.

For more information about units, see Chapter 11.

73 IMPLEMENTATION expected

The reserved word **IMPLEMENTATION** must appear between the interface part and the actual procedure definitions.

74 Constant and case types don't match

The type of the **case** constant is incompatible with the **case** statement's selector expression. For more information on type compatibility, see page 25.

75 Record or object variable expected

This variable must be of a record or object type. For more information, see page 17.

76 Constant out of range

You are trying to do one of the following:

- Index an array with an out-of-range constant.
- Assign an out-of-range constant to a variable.

- Pass an out-of-range constant as a parameter to a procedure or function.

77 File variable expected

This variable must be of a file type. For more information on file types, see page 21.

78 Pointer expression expected

This expression must be of a pointer type. For more information on pointer types, page 21.

79 Integer or real expression expected

This expression must be of an integer or a real type. For more information on integer or real types, see page 11.

80 Label not within current block

A **goto** statement cannot reference a label outside the current block. For more information on **goto** statements, see page 56.

81 Label already defined

The label already marks a statement. You have declared a label and then are trying to reassign the label identifier within the same block. For more information on labels, see page 6.

82 Undefined label in preceding statement part

The label was declared and referenced in a statement part, but never defined. You must define some action to occur when control proceeds to this label. For more information on labels, see page 6.

83 Invalid @ argument

Valid arguments are variable references and procedure or function identifiers. For more information on the @ operator, see page 49.

84 UNIT expected

The reserved word **unit** should appear in the header for the module. If you have any other reference, such as **program** or **library**, you must replace it with **unit**. If you are trying to compile this unit as a .DLL you must replace **program** with **library** in the project file.

85 ";" expected

A semicolon does not appear where it should. The line above the highlighted line is missing a semicolon. All Object Pascal statements are separated by a semicolon.

86 ":" expected

A colon does not appear where it should. The offending line is missing a colon. When you are defining an identifier you must separate it from its type using a colon. For more information on type declarations, see page 11.

87 "," expected

A comma does not appear where it should.

88 "(" expected

An opening parenthesis is missing from the selected line.

This error might indicate that the compiler considers the identifier to the left of the insertion point a type identifier. In this case, the compiler is looking for an opening parenthesis to make a typecast expression.

89 ")" expected

A closing parenthesis is missing from the selected line.

90 "=" expected

An equal sign is missing from the selected line. The equals sign is a relational operator used to test equality.

91 ":=" expected

An assignment operator is missing from the selected line. The assignment operator is used to assign values to variables or properties.

For more information on assignment statements, see page 55.

92 "[" or "(." expected

A left bracket is missing from the selected line.

93 "]" or ".)" expected

A right bracket is missing from the selected line.

94 "." expected

A period is missing from the selected line. This indicates that a type is being used as a variable or that the name of the program itself overrides an important identifier from another unit.

95 ".." expected

A subrange is missing from the declaration for a subrange type. For more information on subrange types, see page 15.

96 Too many variables

Global

The total size of the global variables declared within a program or unit cannot exceed 64K.

Local

The total size of the local variables declared within a procedure or function cannot exceed 64K.

97 Invalid FOR control variable

The **FOR** statement control variable must be a simple variable defined in the declaration part of the current subroutine.

98 Integer variable expected

This variable must be of an integer type. For more information on integers, see page 12.

99 File types are not allowed here

A typed constant cannot be of a file type. For more information on typed constants, see page 35.

100 String length mismatch

The length of the string constant does not match the number of components in the character array. For more information on string constants, see page 37.

101 Invalid ordering of fields

The fields of a record or object type constant must be written in the order of declaration. For more information on record types see page 19.

102 String constant expected

A string constant is expected to appear in the selected statement. For more information on string constants, see page 37.

103 Integer or real variable expected

This variable must be of an integer or real type. For more information on numeric types, see page 11.

104 Ordinal variable expected

This variable must be of an ordinal type. For more information on ordinal types, see page 12.

105 INLINE error

The < operator is not allowed in conjunction with relocatable references to variables.

Such references are always word-sized.

For more information on relocatable expressions, see page 198.

106 Character expression expected

This expression must be of a character type. For more information on character types, see page 14.

107 Too many relocation items

The size of the relocation table in the .EXE file exceeds 64K, which is the limit supported by the Windows executable format.

If you encounter this error, your program is simply too big for the linker to handle. It is also probably too big for Windows to execute. Each executable segment has its own relocation table. Too many relocations in one code segment can cause this error.

If you use the old object model supported by Borland Pascal 7, virtual method tables could cause this error to occur in the data segment, since the virtual method tables are 100% relocation items. For more information on the new class model supported by Delphi, see Chapter 9.

One solution is to split the program into a "main" part that executes two or more "subprogram" parts.

108 Overflow in arithmetic operation

The result of the preceding arithmetic operation is not in the Longint range

```
(-2147483648..2147483647)
```

Correct the operation or use real-type values instead of integer-type values.

This is an error that the compiler catches at compile time; the expression in question must be a constant expression, and not an expression that contains variables that are determined at run-time.

109 No enclosing FOR, WHILE, or REPEAT statement

The **Break** and **Continue** standard procedures cannot be used outside a **for**, **while**, or **repeat** statement.

110 Debug information table overflow

This error indicates that an overflow occurred while generating Turbo Debugger symbol information in the .EXE file.

To avoid this error, turn debug information off (using a **{$D-}** compiler directive) in one or more of your units or by unchecking Debug Information on the Compiler page of the Project Options dialog box.

112 CASE constant out of range

For integer-type case statements, the constants must be within the range - 32768..65535. For more information on case statements, see page 58.

113 Error in statement

This symbol cannot start a statement.

This error, for example, can be caused by unbalanced **begin** and **end** statements. As an example, the following segment of code generates this error on the **else** statement:

```
if ( <expression> )
  begin
    <statement>
    <statement>
  else
    <statement>
```

114 Cannot call an interrupt procedure

You cannot directly call an interrupt procedure.

115 Duplicate CASE constant

The constant or range is already handled by a previous **case** statement entry. A **case** statement is not allowed to have overlapping constants or ranges.

116 Must be in 80x87 mode to compile

This construct can only be compiled in the **{$N+}** state.

Operations on the 80x87 real types (Single, Double, Extended, and Comp) are not allowed in the **{$N-}** state. Note that **{$N+}** is on by default.

117 Target address not found

The Search|Find Error command could not locate a statement that corresponds to the specified address.

The unit containing the target address must have **{$D+}** debug information enabled for the compiler to locate the statement.

118 Include files are not allowed here

Every statement part must be entirely contained in one file.

119 No inherited methods are accessible here

You are using the **inherited** keyword outside a method, or in a method of an object type that has no ancestor. For more information on inheriting methods or properties, see page 87.

121 Invalid qualifier

You are trying to do one of the following:

- Index a variable that is not an array.
- Specify fields in a variable that is not a record.
- Dereference a variable that is not a pointer.

For more information on qualifiers, see page 32.

122 Invalid variable reference

This construct follows the syntax of a variable reference, but it does not denote a memory location. Some of the possible causes for this error are:

- You are trying to pass a constant parameter to a variable parameter.
- You are calling a pointer function, but you are not dereferencing the result.
- You are trying to assign a value (or values) to a portion of a property that is of type record. When properties are of type record, you must assign values to the entire record; you cannot assign values to individual fields of the record. (although you can read individual field values of a record). For example, if you have a property *P* that is of type *TPoint*, the following statement will generate this error:

  ```
  P.X := 50;
  ```

  Instead, you must assign values to all the fields of the record property:

  ```
  XX := 50;
  YY := 25;
  P := Point (XX, YY);
  ```

For more information on variable references, see page 31.

123 Too many symbols

The program or unit declares more than 64K of symbols.

If you are compiling with **{$D+}**, try turning it off. You can also split the unit by moving some declaration into a separate unit.

Note This will disable debugging information from the module and will be unable to use the debugger with that module.

124 Statement part too large

Delphi limits the size of a statement part to about 24K of compiled machine code.

If you encounter this error, move sections of the statement part into one or more procedures.

For more information on statements, see Chapter 6.

125 Undefined class in preceding declaration

You declared a forward reference to a class, but the actual declaration of the class does not appear in the same type block (make sure you have not inserted a **const** block between the two blocks). For example,

```
type
  MyClass = class;
```

is a forward reference.

For more information on deriving classes, see Chapter 9.

126 Files must be var parameters

You are attempting to declare a file-type value parameter.

File-type parameters must be **var** parameters.

For more information on **var** parameters, see page 77.

127 Too many conditional symbols

There is not enough room to define further conditional symbols.

Try to eliminate some symbols, or shorten some of the symbolic names.

For more information on conditional symbols, see page 242.

128 Misplaced conditional directive

The compiler encountered an **{$ELSE}** or **{$ENDIF}** directive without a matching **{$IFDEF}**, **{$IFNDEF}**, or **{$IFOPT}** directive.

For more information on these directives, see page 242.

129 ENDIF directive missing

The source file ended within a conditional compilation construct.

There must be an equal number of **{$IFxxx}**s and **{$ENDIF}** in a source file.

130 Error in initial conditional defines

The initial conditional symbols specified on the Directories/Conditionals page of the Project Options dialog box (or in a **/D** directive) are invalid.

The compiler expects zero or more identifiers separated by blanks, commas, or semicolons.

131 Header does not match previous definition

The procedure or function header specified in an interface part or **FORWARD** declaration does not match this header.

132 Too many PUBLIC definitions

The .OBJ file contains more PUBLIC definitions than Delphi's linker can handle. Attempt to reduce the number of PUBLIC definitions, for example by breaking the .OBJ file into two or more .OBJ files.

133 Cannot evaluate this expression

You are attempting to use a non-supported feature in a constant expression or debug expression.

For example, you might be attempting to use the Sin function in a **const** declaration, or attempting to call a user-defined function in a debug expression.

For more information on constant expressions, see page 9.

134 Expression incorrectly terminated

The compiler expects either an operator or the end of the expression (a semicolon) at this point, but found neither.

135 Invalid format specifier

You are using an invalid debugger format specifier in an expression, or else the numeric argument of a format specifier is out of range.

136 Unit is missing from uses clause

The statement attempts to make an invalid indirect reference.

For example, you might be using an **absolute** variable whose base variable is not known in the current module, or using an **inline** routine that references a variable not known in the current module.

Or, you might have declared a new component type, inheriting from, for example, *MyType*. *MyType* has propertied of types defined in a third unit. Your new component's unit does not use that third unit. This results in the generation of this error message on those inherited property types.

In general, if unit A uses a type defined in unit B, and unit B uses a type defined in unit C, unit A must also use unit C to avoid indirect references.

For more information on indirect references, see page 126.

137 Structured variables are not allowed here

You are attempting to perform a non-supported operation on a structured variable.

For example, you might be trying to multiply two records.

For more information on structured variables, see page 17.

138 Cannot evaluate without System unit

Your DELPHI.DSL library file must contain the *System* unit for the debugger to be able to evaluate expressions.

139 Cannot access this symbol

A program's entire set of symbols is available to the debugger (or the Browser) as soon as you have compiled the program with symbolic debug information.

You may be trying to access a symbol that cannot be accessed until you actually run the program.

140 Invalid floating-point operation

An operation on two real-type values produced an overflow or a division by zero. This is a compiler error that results only from the evaluation of constant expressions.

For more information on floating point operations, see page 44.

142 Pointer or procedural variable expected

The *Assigned* standard function requires the argument to be a variable of a pointer or procedural type.

143 Invalid procedure or function reference

You are attempting to call a procedure in an expression.

A procedure or a function must be compiled in the **{$F+}** state, and cannot be declared with inline if it is to be assigned to a procedure variable.

146 File access denied

The file could not be opened or created. The compiler may be trying to write to a read-only file, read from a file that does not exist, or access a file that is already in use. You cannot compile (generate) an .EXE or a .DLL while that module is executing. Terminate the program, then recompile.

147 Object or class type expected

The identifier does not denote an object or class type.

For more information on object types, see Chapter 3.

148 Local object or class types are not allowed

Object and class types can be defined only in the outermost scope of a program or unit.

Object-type definitions within procedures and functions are not allowed.

For more information on object types, see Chapter 3.

149 VIRTUAL expected

The keyword **virtual** is missing.

150 Method identifier expected

The identifier does not denote a method. A method call was expected in the selected statement.

151 Virtual constructors are not allowed

A constructor method must be static when declared using the object model supported by Borland Pascal 7. For more information on constructors, see page 95.

Note that this construct is of concern only when you use the object model supported by Borland Pascal 7; virtual constructors are supported by the Delphi class model. For more information on the Delphi class model, see Chapter 9.

152 Constructor identifier expected

This error message is generated when you make a call similar to the following:

```
New(Type, Constructor);
```

The second parameter is not a constructor of the given object type. For more information on constructors, see page 95.

Note that this construct is of concern only when you use the object model supported by Borland Pascal 7. For more information on the Delphi class model, see Chapter 9.

153 Destructor identifier expected

This error message is generated when you make a call similar to the following:

```
Dispose (objvar, Destructor);
```

The second parameter is not a destructor of the given object type. For more information in destructors, see page 96.

154 Fail only allowed within constructors

The *Fail* standard procedure can be used only within constructors.

155 Invalid combination of opcode and operands

The assembler opcode does not accept this combination of operands.

Possible causes are:

- There are too many or too few operands for this assembler opcode; for example, INC AX,BX or MOV AX.
- The number of operands is correct, but their types or order do not match the opcode; for example, DEC 1, MOV AX,CL or MOV 1,AX.

156 Memory reference expected

The assembler operand is not a memory reference, which is required here.

You may have forgotten to put square brackets around an index register operand, for example MOV AX,BX+SI instead of MOV AX,[BX+SI].

For more information on memory references, see page 198.

157 Cannot add or subtract relocatable symbols

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant.

Variables, procedures, functions, and labels are relocatable symbols. Assuming that Var is variable and Const is a constant, then the instructions MOV AX,Const+Const and MOV AX,Var+Const are valid, but MOV AX,Var+Var is not.

158 Invalid register combination

Valid index-register combinations are

- [BX]
- [BP]
- [SI]
- [DI]
- [BX+SI]
- [BX+DI]
- [BP+SI]

- [BP+DI]

Other index-register combinations, such as [AX], [BP+BX], and [SI+DX], are not allowed.

Local variables are always allocated on the stack and accessed via the BP register. The assembler automatically adds [BP] in references to such variables, so that even though a construct like Local[BX] (where Local is a local variable) appears valid, it is not, since the final operand would become Local[BP+BX].

159 286/287 instructions are not enabled

This error is generated when you are using a 80286 (or greater) opcode in inline assembly statements. Use the **{$G+}** compiler directive to enable 286/287 opcodes, but be aware that the resulting code cannot be run on 8086 and 8088 based machines (the default for this directive is on).

160 Invalid symbol reference

This symbol cannot be accessed in an assembler operand.

Possible causes are that you are attempting to:

- access a standard procedure, a standard function, or the Mem, MemW, MemL, Port, or PortW special arrays in an assembler operand.
- access a string, floating-point, or set constant in an assembler operand.
- access an inline procedure or function in an assembler operand.
- access the @Result special symbol outside a function.
- generate a short JMP instruction that jumps to something other than a label.

161 Code generation error

The preceding statement part contains a LOOPNE, LOOPE, LOOP, or JCXZ instruction that cannot reach its target label. These short-jump assembly instructions can jump at the most 127 bytes (or -128 bytes). Reverse the logic of your jump instruction to do an unconditional far jump to the target label, preceded by conditional near jump (reverse logic) over the far jump.

162 ASM expected

The compiler expects an **ASM** reserved word at this location. (A procedure or function declared as **assembler** must start with **asm**, not **begin**.

163 Duplicate dynamic method index

This dynamic method index has already been used by another method.

For more information on dynamic methods, see page 92.

Note that this construct is of concern only when you use the object model supported by Borland Pascal 7, since you cannot specify an index for dynamic methods using the class model introduced with Delphi. For more information on the Delphi class model, see Chapter 9.

164 Duplicate resource identifier

This resource file contains a resource with a name or identifier that has already been used by another resource.

165 Duplicate or invalid export index

The ordinal number specified in the Index cause is not between 1 and 32767, or has already been used by another exported routine.

166 Procedure or function identifier expected

The exports clause only allows procedures and functions to be exported.

167 Cannot export this symbol

A procedure or function cannot be exported unless it was declared with the export directive.

168 Duplicate export name

The name specified in the name clause has already been used by another exported routine.

169 Executable file header too large

The size of the header of the .EXE file being generated exceeds the .EXE file format's upper limit of 64K bytes.

You may be importing or exporting too many procedures and functions by name, or you may have too many named resources.

Consider importing or exporting by ordinal numbers instead.

170 Too many segments

The executable file being generated contains more than 254 segments, which is the upper limit of the DOS protected mode and Windows .EXE file format.

Most likely, the preferred code segment size for the executable file is too small. Specify a larger preferred code segment size using a **{$S** *segsize***}** compiler directive.

172 READ or WRITE clause expected

The property you are declaring must obtain its values by reading and writing to methods or fields declared in the private part of the class declaration or are inherited from an ancestor.

For more information on declaring properties, see page 102.

173 Cannot read a write-only property

The property you are attempting to read from is write-only. It has no value to read.

174 Cannot assign to a read-only property

The property is your assignment statement is read only. It will not accept any values you try to assign.

175 Cannot exit a FINALLY block

The entire group of statements in a finally block must execute. It is illegal to prematurely break out of a finally block.

For more information on finally blocks, see page 120.

176 Label and GOTO not at same nesting level

Any label which is being accessed by a goto statement must be within the same block a the goto statement. You cannot jump into another routine.

For more information on goto statements, see page 56.

177 ON expected

To handle a raised exception within an excepts block you must preface the exception type identifier with the **on** standard directive. For more information on handling exceptions, see page 115.

178 Cannot mix class and object types

These types are incompatible. You cannot inherit one type from the other or assign from variables of one to the other. For more information on type compatibility, see page 25.

179 PROCEDURE or FUNCTION expected

The class method you are trying to declare requires that you follow the **class** reserved word with either **procedure** or **function**. For more information on class methods, see page 110.

180 Class type identifier expected

The instance of the exception you are trying to handle in your **on** clause must be a class type. For more information on handling exceptions, see page 115.

181 Class expression expected

The exception you are trying to raise must evaluate to a class instance. You can only raise exceptions on class instances. For more information on raising exceptions, see page 114.

182 Instance variable not accessible here

Inside the body of a class procedure or class function, you cannot refer to any instance data, nor can you refer to the **self** pointer or any fields of the class type.

183 Invalid method reference

You are trying to access a method of a class without a valid instance of that class. You cannot call non-class methods from within a class method. To solve this problem you must instantiate the class or use a class method. For more information on class methods, see page 110.

184 Default property must be an array property

You have declared a property to be the default property of a class that is not of type array or without assigning it a value. (You might have forgotten to define a value after the **default** directive.

Although you can assign a default value to a property of any type, Delphi will only accept default properties that are of type array.

For more information on specifying default properties, see page 104.

185 Class already has a default property

Each class can only have one default property. You cannot change the default if your ancestor defines a default array property. For more information on specifying default properties, see page 104.

186 Invalid message handler parameter list

The parameter list you are passing to the message handler does not match the list the message handler was expecting. A message method must have one variable parameter of any type. Any other parameters will cause this message to be generated.

187 Method does not exist in base class

You can only override methods that exist in an ancestor class. You are trying to override a method that does not exist in the ancestor class. To solve the problem, remove **override** from the method declaration and recompile your module (also, be sure you are using the correct spelling of the method you are typing to override.

For more information on overriding methods, see page 93.

188 Cannot override a static method

Static methods cannot be overridden using the **override** directive. A method must be dynamic or virtual in order to be overridden. For more information on static methods, see page 90.

189 Property does not exist in base class

You can only override properties that exist in an ancestor class. You can move an inherited property to a higher visibility level by "redeclaring" the property name with no type information in the descendant. For example:

```
Public
  property MyProp;
  end;
```

Also, when you are redeclaring (promoting) a property from protected to public or published, make sure you match the spelling of the property as it is shown the base class.

190 Unit has no register procedure

When you are declaring a new component, you must declare and define a Register procedure for the unit. This will tell Delphi what components to add to its component library and which page of the Component palette the components should appear. Note that you will only receive this message when you install the unit into the component library.

191 Type not supported in expression lists

In an **array of const** constructor you can only use the following types:

- float types
- scalar types
- pointer types
- string types

For example, you cannot pass a record to the *Format* function which uses an array of **const** for the second parameter.

For more information on type-variant open-array parameters, see page 82.

192 Property access method not found

The property you are trying to evaluate is a write only property. You cannot evaluate a property that does not have a read method. For more information on read methods, see page 102.

193 Expression too complex

The selected expression is too complex for the debugger's expression evaluator. Try breaking the expression into smaller statements. For more information on expressions, see Chapter 5.

194 Process faulted during evaluation

The expression you are evaluating in the integrated debugger caused a General Protection Fault during its execution.

195 Exception raised during evaluation

The process which you are evaluating with the integrated debugger threw an exception.

196 Evaluator already active

You can only have one instance of the debugger's evaluator open at any time.

197 Property access method removed by smart linker

You are trying to access or evaluate a run-time only property for a class that was not used in the source code. If a run-time only property is not used in the source code, the smart linker removes that property from the compiled code.

If you want to access this property, you must use it in the source code.

198 RAISE not allowed outside EXCEPT..END block

You can only reraise an exception within a **except** block. For more information on reraising exceptions, see page 117.

199 Resource file format error

The given file is not a valid .RES file. Note that bitmap (.BMP) and icon (.ICO) files cannot be linked directly into an executable. They must first be imported into a resource file.

200 PUBLISHED not allowed in this class

If a class is declared in the {**$M–**} state, and is not derived from a class that was declared in the {**$M+**} state, **published** sections are not allowed in the class. For further details, see "Component Visibility" on page 89, and the description of the **$M** compiler directive on page 181.

201 This field cannot be PUBLISHED

Fields defined in a **published** section must be of a class type. Fields of all other types are restricted to **public**, **protected**, and **private** sections.

202 This property cannot be PUBLISHED

The property does not meet the requirements of a **published** property. For further details, see "Published components" on page 89.

Run-time errors

Certain errors at run time cause the program to display an error message and terminate:

```
Run-time error nnn at xxxx:yyyy
```

where nnn is the run-time error number, and xxxx:yyyy is the run-time error address (segment and offset).

The run-time errors are divided into four categories: File errors, 1 through 99; I/O errors, 100 through 149; and fatal errors, 200 through 255.

File errors

1 Invalid function number.

You made a call to a nonexistent DOS function.

2 File not found.

Reported by *Reset*, *Append*, *Rename*, *Rewrite* if the filename is invalid, or *Erase* if the name assigned to the file variable does not specify an existing file.

3 Path not found.

- Reported by *Reset*, *Rewrite*, *Append*, *Rename*, or *Erase* if the nameassigned to the file variable is invalid or specifies a nonexistent subdirectory.
- Reported by *ChDir*, *MkDir*, or *RmDir* if the path is invalid or specifiesa nonexistent subdirectory.

4 Too many open files.

Reported by *Reset*, *Rewrite*, or *Append* if the program has too many open files. DOS never allows more than 15 open files per process. If you get this error with less than 15 open files, it might indicate that the CONFIG.SYS file does not include a FILES=xx entry or that the entry specifies too few files. Increase the number to some suitable value, such as 20.

5 File access denied.

- Reported by *Reset* or *Append* if *FileMode* allows writing and the name assigned to the file variable specifies a directory or a read-only file.
- Reported by *Rewrite* if the directory is full or if the name assigned to the file variable specifies a directory or an existing read-only file.
- Reported by *Rename* if the name assigned to the file variable specifies a directory or if the new name specifies an existing file.
- Reported by *Erase* if the name assigned to the file variable specifies a directory or a read-only file.
- Reported by *MkDir* if a file with the same name exists in the parent directory, if there is no room in the parent directory, or if the path specifies a device.
- Reported by *RmDir* if the directory isn't empty, if the path doesn't specify a directory, or if the path specifies the root directory.
- Reported by *Read* or *BlockRead* on a typed or untyped file if the file is not open for reading.
- Reported by *Write* or *BlockWrite* on a typed or untyped file if the file is not open for writing.

6 Invalid file handle.

This error is reported if an invalid file handle is passed to a DOS system call. It should never occur; if it does, it is an indication that the file variable is somehow trashed.

12 Invalid file access code.

Reported by *Reset* or *Append* on a typed or untyped file if the value of *FileMode* is invalid.

15 Invalid drive number.

Reported by *GetDir* or *ChDir* if the drive number is invalid.

16 Cannot remove current directory.

Reported by *RmDir* if the path specifies the current directory.

17 Cannot rename across drives.

Reported by *Rename* if both names are not on the same drive.

18 No more files.

A call to *FindFirst* or *FindNext* found no files matching the specified file name and set of attributes.

I/O errors

These errors cause termination if the particular statement was compiled in the {**$I+**} state. In the {**$I-**} state, the program continues to execute, and the error is reported by the *IOResult* function.

100 Disk read error.

Reported by *Read* on a typed file if you attempt to read past the end of the file.

101 Disk write error.

Reported by *CloseFile*, *Write*, *Writeln*, or *Flush* if the disk becomes full.

102 File not assigned.

Reported by *Reset*, *Rewrite*, *Append*, *Rename*, and *Erase* if the file variable has not been assigned a name through a call to *Assign*.

103 File not open.

Reported by *CloseFile, Read, Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead*, or *BlockWrite* if the file is not open.

104 File not open for input.

Reported by *Read, Readln, Eof, Eoln, SeekEof*, or *SeekEoln* on a text file if the file is not open for input.

105 File not open for output.

Reported by *Write* and *Writeln* on a text file if you fail to use the WinCrt unit.

106 Invalid numeric format.

Reported by *Read* or *Readln* if a numeric value read from a text file does not conform to the proper numeric format.

Fatal errors

These errors always immediately terminate the program.

200 Division by zero.

The program attempted to divide a number by zero during a **/**, **mod**, or **div** operation.

201 Range check error.

This error is reported by statements compiled in the {**$R+**} state when one of the following situations arises:

- The index expression of an array qualifier was out of range.
- You attempted to assign an out-of-range value to a variable.
- You attempted to assign an out-of-range value as a parameter to a procedure or function.

202 Stack OverFlow

There are too many local variables declared in a procedure or function compiled in the **{$S+}** state.

Increase the size of the stack with the **$M** compiler directive.

The Stack Overflow error can also be caused by infinite recursion, or by an assembly language procedure that does not maintain the stack properly.

203 Heap overflow error.

This error is reported by *New* or *GetMem* when there is not enough free space in the heap to allocate a block of the requested size.

For a complete discussion of the heap manager, see Chapter 16.

204 Invalid pointer operation.

This error is reported by *Dispose* or *FreeMem* if the pointer is **nil** or points to a location outside the heap.

205 Floating point overflow.

A floating-point operation produced a number too large for Delphi or the numeric coprocessor (if any) to handle.

206 Floating point underflow.

A floating-point operation produced an underflow. This error is only reported if you are using the 8087 numeric coprocessor with a control word that unmasks underflow exceptions. By default, an underflow causes a result of zero to be returned.

207 Invalid floating point operation.

- The real value passed to *Trunc* or *Round* could not be converted to an integer within the Longint range (-2,147,483,648 to 2,147,483,647).
- The argument passed to the *Sqrt* function was negative.
- The argument passed to the *Ln* function was zero or negative.

- An 8087 stack overflow occurred. For further details on correctly programming the 8087, see Chapter 14.

210 Call to an Abstract Function

You are trying to execute an abstract virtual method.

215 Arithmetic overflow error.

This error is reported by statements compiled in the {**$Q+**} state when an integer arithmetic operation caused an overflow, such as when the result of the operation was outside the supported range.

216 General Protection fault

This error results if you try to access memory that is not legal for your your application to access. The operating system halts your application and tells you that a general protection (GP) fault occurred, but your system does not crash. The following practices cause GP faults:

- Dereferencing **nil** pointers
- Using an object that has not been constructed
- Loading constant values into segment registers
- Performing arithmetic operations on segment registers of selectors
- Using segment registers for temporary storage
- Writing to code segments
- Accessing memory beyond the local address space given to your application

217 Unhandled exception

An exception has occurred for which an exception handler could not be located.

219 Invalid typecast

The object given on the left hand side of an **as** operator is not of the class given on the right hand side of the operator.

Index

Symbols

- # (pound) character, 7
- @ operator, 49
  - with a variable, 49
  - with procedures and functions, 50
- @@ operator, 53
- ^ (pointer) symbol, 21, 22, 33
- 80286 code generation compiler switch, 231
- 80x87
  - floating-point model, 16
  - software emulation, selecting, 16
- 80x87 code option, 235

A

- $A compiler directive, 184, 226
- Abs function, 181
- absolute
  - clause syntax, 30
  - expressions, built-in assembler, 199
  - variables, 30
- abstract directive, 92
- abstract methods, 92
- access specifiers
  - property definitions, 103
- accessing properties, 102
- actual parameters, 56
- address
  - factor, 42
- address-of (@) operator, 21, 34, 49, 52
- Align Data command, 226
- aligning data, 226
- allocating
  - Windows memory, 162
- ancestors
  - defined, 87
- and operator, 45
- apostrophes in character strings, 7
- Append procedure, 139, 140
- arguments
  - command-line compiler, 215
- arithmetic
  - operations precison rules, 13
  - operators, 44
- array
  - types, 18
  - variables, 32
- array properties, 104
  - default, 105
- arrays, 18, 32
  - accessing elements, 18
  - indexing
    - multidimensional, 32
  - number of elements, 18
  - of arrays, 18
  - types, 166
  - valid index types, 18
  - zero-based character, 19, 37, 153, 155
- array-type constant syntax, 37
- asm statement, 187
- .ASM files, 152
- assembler
  - code
    - in Delphi, 187
    - linked with Delphi, 207
  - declaration syntax, 73
- assembly language, 234
  - 80x87 emulation and, 152
  - call model, 207
  - inline
    - directives, 213
    - statements, 211
  - interfacing programs with, 208
  - statements
    - multiple, 188
    - syntax, 192
- Assign procedure, 139, 140, 145
- AssignCrt procedure, 144, 145
- assignment
  - compatibility, 26
  - statement syntax, 55
- assignment-compatibility
  - class-reference types, 109
- automatic
  - call model selection, overriding, 174
  - data segment, 160
  - jump sizing, built-in assembler, 189
  - word alignment, 184
- AutoTracking variable, 143, 145
- AX register, 173, 213

B

- $B compiler directive, 46, 182, 227
- base type, 21
- binary
  - arithmetic operators, 44
  - operands, 41
  - operators, 13
- binary format, 221
- bitwise operators, 45
- blanks, defined, 3
- block
  - defined, 65
  - scope, 66
  - syntax, 65
- BlockRead procedure, 139, 142
- BlockWrite procedure, 139, 142
- Boolean
  - data type, 14, 163
  - expression evaluation, 182
    - complete, 46
  - operators, 14
  - variables, 14
- Boolean evaluation
  - compiler switch, 227
  - complete, 227
  - short circuit, 227
- Boolean expression evaluation
  - short-circuit, 46
- Boolean operators, 45
- BP register, 178
- brackets, in expressions, 50
- BufEnd variable, 167

- buffer
- BufPtr pointer, 167
- BufSize variable, 167
- Build command, 219
- build command-line option, 219
- built-in assembler
  - directives, 187
  - expressions, 192
    - classes, 198
    - Object Pascal
      - expressions versus, 193
    - operators, 192
    - types, 201
  - instruction sizing, 189
  - opcodes, 189
  - operands, 192
  - procedures and functions, 203
  - reserved words, 192
- BX register, 173
- Byte data type, 12
- ByteBool data type, 163

C

- $C code segment attribute, 227
- call model, 207
- call models, 70
- callbacks
  - smart, 71
- calling
  - methods, 93
- calling conventions, 171
  - constructors and destructors, 176
- calls
  - near and far, 173
- case
  - sensitivity of Turbo Pascal, 4
  - statement syntax, 58
- Char data type, 14, 162
- character
  - arrays, 155
  - pair special symbols, 4
  - pointer operators, 46
  - pointers
    - character arrays and, 155
    - indexing, 155
    - string literals and, 154
    - strings, 7
- ChDir procedure, 139
- CheckBreak variable, 145
- CheckEOF variable, 144, 145
- Chr function, 14, 181
- class components
  - private, 90
  - protected, 90
  - public, 89
  - published, 89
  - scope, 88
  - visibility, 89
- class domain, 87
- class forward references, 88
- class instances, 86
- class methods, 87, 110
- class of reserved words, 108
- class reference types
  - constructors, 109
- class references, 86
- class reserved word, 88, 110
- class type compatibility, 88
- class types, 85
- classes
  - ancestors, 87
  - components, 85
  - descendants, 87
  - inheritance, 87
  - scope, 88
- class-reference types, 108
- ClassType method function, 109
- Close procedure, 139, 140, 145
- CloseFile procedure, 146
- ClrEol procedure, 144
- ClrScr procedure, 144
- CmdShow variable, 137
- Code generation
  - Windows, 241
- code segment, 208
  - attributes, 159
    - changing, 160
  - maximum size of, 159
  - procedures and functions in, 207
- Code segment attribute, 227
- codesegs2, 159
- command-line
  - compiler reference, 215
  - options, 215
    - /B, 219
    - /D, 218
    - debug, 221
    - /E, 221
    - /F, 219
    - /G, 221
    - /GD, 222
    - /GP, 222
    - /GS, 222
    - /I, 221
    - /L, 219
    - /M, 219
    - mode, 218
    - /O, 221
    - /Q, 220
    - switching directive defaults (/$), 217
    - /T, 220
    - /U, 221
    - /V, 222
- command-line compiler
  - arguments, 215
  - compiling and linking with, 215
  - extended syntax, 217
  - options, 215
    - 286 code generation, 216
    - align data, 216
    - append debug information to EXE, 222
    - Boolean evaluation, 216
    - build all, 218, 219
    - build all units, 217
    - debug, 221
    - debug information, 216, 222
      - in EXE, 217
    - define conditional symbol, 217
    - DOS real-mode .EXE, 217
    - emulation, 216
    - EXE & DCU directory, 221
    - EXE and TPU directory, 217
    - find error, 219
    - find run-time error, 217
    - force far calls, 216
    - I/O checking, 216
    - include directories, 217, 221
    - link buffer, 218, 219

link buffer on disk, 217
list of, 216
local symbols, 216, 222
make, 219
make modified units, 217
map file, 217, 221
memory sizes, 216
numeric coprocessor, 216
object directories, 217, 221
open parameters, 216
overflow checking, 216
quiet compile, 217, 218, 220
range checking, 216
resource directories, 221
Smart callbacks, 216
stack checking, 216
TPL & CFG directories, 217
TPL & CFG directory, 220
type-checked pointers, 216
unit directories, 217, 221
var-string checking, 216
Windows real-mode support, 216
symbol reference information, 217
comments, 7
built-in assembler, 188
common types of integer types, 13
Comp data type, 16, 150, 165
comparing
character pointers, 48
packed strings, 48
pointers, 48
sets, 49
simple types, 48
strings, 48
values of real types, 151
compilation
conditional, 242
compiler
directives
$A, 226
$B, 46, 227
$C, 227
change state of, 217
conditional, 226
$D, 219, 228, 229
$DEFINE, 218, 229, 243
defined, 8
$ELSE, 229
$ENDIF, 230
$F, 24, 71, 174, 230
$G, 189, 231
$I, 233
$IFDEF, 232
$IFNDEF, 232
$IFOPT, 232
$K, 239
$L, 207, 208, 211, 233, 234
$L filename, 73, 152
$M, 30, 218, 235
$N, 16, 44, 189, 235
$P, 78, 235
parameter, 225
$R, 237
$S, 30, 137, 238, 239
switch, 225
$T, 26, 49
$UNDEF, 241, 243
$V, 241
$W, 241
$X, 230
$X, 7, 19, 22, 31, 46
$Y, 240
compiling
to .EXE file, 219
Complete Boolean Eval option, 227
complete Boolean evaluation, 46
compound statement syntax, 57
concatenation, 46
conditional
defines (command-line option), 218
symbols, 243
conditional statement syntax, 57
configuration file
DCC.CFG, 222
CONST segment, 207
constant
address expressions, 36
declaration part syntax, 65
declarations, 9
expressions, 9
parameters, 76
constants, 9, 194
array-type, 37
folding, 181
merging, 181
numeric, built-in assembler, 194
pointer-type, 39
procedural-type, 40
record-type, 38
set-type, 39
simple-type, 9, 36
string, built-in assembler, 194
string-type, 36
structured-type, 37
typed, 35
constructor reserved word, 94
constructors, 95
calling conventions, 176
class references, 109
defined, 87
control
characters, 7, 143
defined, 3
embedding in strings, 7
string syntax diagram, 7
CPU symbols, 244
creatfile, 219
creating objects, 95
CRT window, 142
closing, 143
scrolling, 143
CSEG segment, 208
current
file position, 140
file size, 140
Cursor variable, 145
CursorTo procedure, 144

D

$D compiler directive, 219, 228
$D description, 229
/D command-line option, 218
data
alignment, 184
ports, 169

segment, 207, 208
DLL, 137
maximum size, 30
data alignment, 226
DATA segment, 207
DCC.CFG file, 220, 222
sample, 223
dead code eliminated, 184
Debug Information
command, 228
option, 228
debugging
command-line option, 222
information switch, 228
options, command-line, 221
range-checking switch, 237
stack overflow switch, 239
decimal notation, 6
default array properties, 105
default directive, 105, 106
$DEFINE compiler directive, 218, 229, 243
defining properties, 102
access specifiers, 103
storage specifiers, 106
DELPHI.DSL, 220, 221
DEMANDLOAD code segment attribute, 160
descendants
defined, 87
Description directive, 229
designators
field, 33
method, 33
destorying objects
nil values, 97
Destroy method, 97
Destroy method versus Free method, 97
destroying objects, 96
destructor reserved word, 94
destructors, 96
calling conventions, 176
defined, 87
devices
drivers, 145
digit syntax diagram, 3
digits, defined, 3
direct memory, 168
directives
assembler, defined, 190
built-in assembler, 187, 203, 204
compiler, defined, 8
forward, 72
inline, 73
list of Borland Pascal, 4
private, 5
public, 5
standard, 4
directories
command-line options, 220
DISCARDABLE code segment attribute, 160
div operator, 44
DLL
CmdShow variable, 137
contrasted with a unit, 129
data segment, 137
exit code, 135
files in a, 137
global memory in a, 137
global variables in, 137
gmem_DDEShare attribute, 137
HeapLimit variable, 138
HPrevInst variable, 137
initalization code, 135
KERNEL, 130
multi-language programming, 129
PrefixSeg variable, 137
run-time errors in a, 138
structure of, 133
syntax, 133
unloading a, 136
using a, 130
domain
classes, 87
DoneWinCrt procedure, 143, 144
double address-of (@@) operator, 53
Double data type, 16, 150, 164
drivers
text-file device, 145
DS register, 178
DSEG segment, 207
DSL & CFG directory command-line option, 220
DX register, 173
dynamic
importing, 132
linking, 129
variables, 30
dynamic directive, 92
dynamic methods, 92
dynamic variables, 21, 33
dynamic versus virtual methods, 92
dynamic-link libraries, 129. *See also* DLL

E

/E command-line option, 221
editing keys
in WinCrt unit, 144
eliminate dead code, 184
$ELSE compiler directive, 229
embedding control characters in strings, 7
empty set, 21
$ENDIF compiler directive, 230
end-of-line character, 3
entry code, 176, 203
enumerated
types, 14, 163
Eof function, 139
Eoln function, 139
Erase procedure, 139
error messages
searching, 219
ErrorAddr variable, 179
errors
range, 237
reporting, 178
examples
array type, 18
character strings, 7
constant expressions, 9, 10
control characters in strings, 7
enumerated type, 15
record type, 19
subrange type, 15
using subrange types, 15
variant part of a record, 20
EXE & DCU directory command-line option, 221
.EXE files

  building, 185
.EXE files, creating, 219
exit
  code, 203
    in DLL, 136
  functions, 176
  procedures, 136, 137, 176, 178
ExitCode variable, 136, 179
exiting a program, 178
ExitProc variable, 136, 137, 178
exponents, 163
export directive, 134
exports clause, 134
  syntax, 134
expression syntax, 41
expressions, 194, 199
  built-in assembler, 192, 199
    classes, 198
    elements of, 196
    versus Object Pascal, 193
  constant, 9, 10
    address, 36
  order of evaluation, 182
  types, built-in assembler, 200
Extended data type, 16, 150, 164
  range arithmetic, 150
extended syntax, 7, 19, 22, 230
Extended Syntax option, 230
external
  (reserved word), 211
  declaration, 130
  declarations, 234
  directive, 204
    in imported procedure and functions, 130
  procedures and functions, 152, 207
EXTRN directive, 208

F

$F compiler directive, 24, 71, 145, 174, 230
/F command-line option, 219
False predefined constant identifer, 14
far
  call, 173
    model
      in imported procedures and function, 130
    model, forcing use of, 178
far call model, forcing use of, 230
field
  designators syntax, 33
  list (of records), 19
  record, 33
fields
  object, 87
file. *See also* files
  buffer, 167
  handles, 166
  modes, 166
  types, 21
FilePos function, 139, 140
files
  .ASM, 152
  .OBJ, 207
  functions for, 139
  .MAP, 221
  .OBJ, 221
    linking with, 233
  procedures for, 139
  text, 141
  typed, 166
  types of, 166
  untyped, 142, 166
files,, 274
FileSize function, 139, 140
Find Error command, 219
find error command-line option, 219
finding the size of a given string, 17
FIXED code segment attribute, 159
fixed part of records, 19
floating-point
  calculations, type Real and, 150
  numbers, 16, 149
  numeric coprocessor (80x87), 16
  parameters, 172
  software, 16
  types, 150
floating-point,, 276
Flush function, 146
Flush procedure, 139
for statement syntax, 61
Force Far Calls
  command, 231
  compiler directive, 230
formal
  parameter list syntax, 75
  parameters, 50, 56
forward
  declarations, 72
  directive, 72
forward references, 88
Free method, 97
Free method versus Destroy method, 97
function
  calls, 50
    extended syntax and, 50
    syntax, 50
  declarations
    assembler, 72
    external, 72
  headings, 74
  results, 173
  syntax, 74
functions, 69, 203. *See also* procedures and
  calls, 171
  discarding results, 230
  entry/exit code, built-in assembler, 203
  extended syntax, 230
  far, 174
  near, 174
  nested, 174
  parameters, built-in assembler, 203
  private, 125
  SizeOf, 79
  stack frame for, built-in assembler, 203
  standard
    and constant expressions, 10

G

$G compiler directive, 189, 231
/G command-line option, 221

/GD command-line option, 222
GetDir procedure, 139
GetMem procedure, 33
global
  heap in Windows, 161
  memory in a DLL, 137
  variables in a DLL, 137
gmem_DDEShare attribute, 137, 138
gmem_Moveable attribute, 138
goto statement syntax, 56
GotoXY procedure, 144
/GP command-line option, 222
Group unit segments compiler directive, 231

H

Halt procedure, 178
handles
  file, 166
heap
  management
    Windows, 161
      fragmenting, 161
  management, sizes, 218
  manager
    Windows, 161
      allocating memory blocks, 162
  Windows global, 161
  Windows local, 161
    changing size of, 161
    size of, 161
heap management, sizes, 235
HeapAllocFlags variable, 138, 162
HeapBlock variable, 162
HeapLimit variable, 162
heapmanwin, 161
hex digits, 3
hexadecimal
  constants, 6
  numbers, 6
Hi function, 181
high bounds of index type of an array, 18
High function, 12, 18, 79
highest value in a range, 12
HInstance variable, 137
host type, 15

I

$I compiler directive, 140, 233
I/O, 139
  checking, 233
    command, 233
  devices, 145
  error-checking, 140, 233
/I command-line option, 221
identifiers
  as labels, 7
  defined, 5
  examples, 5
  how identified in manuals, 5
  length of, 5
  qualified, 5
  restrictions on naming, 5
  scope of, 11
if statement syntax, 57
$IFDEF compiler directive, 232
$IFNDEF compiler directive, 232
$IFOPT compiler directive, 232
immediate values, built-in assembler, 198
implementation part
  of a unit, 125, 174
  syntax, 125
import units, 131
importing procedures and functions
  dynamically, 132
  statically, 132
  with import unit, 131
in operator, 47, 49
InactiveTitle variable, 145
Include directories command-line option, 221, 233
include files, 221, 233
including resources, 237
index
  clause, 130, 135
  syntax, 32
  types valid in arrays, 18
index directive, 105
index parameter list
  array properties, 104
index specifiers
  property definitions, 105
indexed properties, 104
indexes in arrays, 18
indexing character pointers, 155
indirect unit references, 126
infinite loop. *See* loop, infinite
inherited reserved word, 88, 94
initialization
  code in DLL, 135
  part of a unit, 125
initialized variables, 35
  in assembler, 207
InitWinCrt procedure, 143, 144
inline
  directive, 73
  directives, 213
  statements, 211
InOut function, 146
Input text-file variable, 141
  and WinCrt unit, 142
  in WinCrt unit, 141
instances, 86
instruction opcodes, built-in assembler, 189
Integer data type, 12, 162
integer types, 14
interface part of a unit, 124, 174
interfacing=page, 208
internal data formats, 162
invalid
  built-in assembler, 196
IOResult function, 139, 140

J

jump sizing, automatic, built-in assembler, 189

K

$K compiler directive, 239
KERNEL dynamic-link library, 130
KeyPressed function, 144

L

$L compiler directive, 207, 208, 211, 233, 234
$L filename compiler directive, 73, 152

/L command-line option, 219
label
  declaration part syntax, 65
  syntax, 6
labels
  built-in assembler, 188
  defined, 6
left
  brace special symbol, 4
  bracket special symbol, 4
Length function, 181
length of
  a string-type value, 17
  character strings, 7, 165
  identifiers, 5
  program lines, 8
  record, 167
letters, defined, 3
libraries
  dynamic-link, 129
library header, 133
line
  input editing keys, 144
lines, maximum length of, 8
Link Buffer
  option, 219
linking
  buffer option, 219
  dynamically, 129
  object files, 233
  smart, 185
  statically, 129
Lo function, 181
local
  labels, 189
local symbol information switch, 234
Local Symbols
  command, 234
  option, 234
logical
  operators, 45
LongBool data type, 14, 163
Longint data type, 12
loop, infinite. *See* infinite loop
low bounds of index type of an array, finding, 18
Low function, 12, 18, 79
lowest value in a range, finding, 12

M

$M compiler directive, 30, 161, 218, 235
.MODEL directive, 208
/M command-line option, 219
machine code in program, 211
Make command, 219
make command-line option, 219
map file command-line option, 221
.MAP files, 221
Mem array, 168
MemL array, 168
memory
  allocation, 218
    compiler directive, 235
  model, 208
  Object Pascal and, 159
  references, built-in assembler, 198
  size, 235
MemW array, 168
metaclasses, 108
method activation, 93
method designator, 93
method identifiers
  qualified, 94
methods
  abstract, 92
  assembly language, 211
  calling, 93
  class, 87, 110
  constructors, 87
  defined, 87
  designators, 33
  destructors, 87
  dynamic, 92
  external, 211
  implementation, 93
  overriding, 91
  scope, 94
  static, 90
  virtual, 91
  virtual versus dynamic, 92
MkDir procedure, 139
mod operator, 45
modular programming, 124
MOVEABLE code segment attribute, 159
multi-language programming and DLLs, 129

N

$N compiler directive, 44
$N compiler directive, 16, 189, 235
name clause, 130, 135
near
  call, 173
nested procedures and functions, 24, 174
nesting include files, 233
New procedure, 21, 33
nil (reserved word), 21, 34
nil values
  destroying objects, 97
nodefault directive, 106
not operator, 45
NULL Character, 153
null strings, 7, 17
null-terminated strings, 19
  defined, 153
  NULL Character, 153
  pointers and, 154
  standard procedures and, 157
number
  constants, 6
  syntax, 6
numbers
  counting, 6, 162
  hexadecimal, 6
  integer, 6
  real, 6
numeric
  constants, built-in assembler, 194
  coprocessor
    emulating, assembly language and, 152
    evaluation stack, 152
numeric coprocessor
  compiler switch, 235

O

.OBJ files, 207
/O command-line option, 221
.OBJ files, 221
  linking with, 233
object

  component designators, 33
  directories, compiler directive, 233
  files, 207
  files, linking with, 233
object directories command-line option, 221
object reference, 86
object types. *See also* objects
objects
  creating, 95
  defined, 86
  destroying, 96
  fields, 87
    designators, 33
  files in $L directive, 208
  methods, 87
  referencing, 86
  scope, 67
Odd function, 181
opcomp, 181
Open function, 146
open parameters, 76, 78
  array, 19, 76, 79
  how passed, 172
  string, 17, 78
open string parameters
  compiler switch, 235
OpenString identifier, 17
operands, 41, 192
  built-in assembler, 192
operators
  @ (address-of), 21, 34, 49, 52
  @@ (double address-of), 53
  and, 45
  arithmetic, 44
  binary arithmetic, 44
  bitwise, 45
  Boolean, 45
  character-pointer, 46
  div, 44
  logical, 45
  mod, 45
  not, 45
  or, 45
  precedence of, 41, 44
    built-in assembler, 201
  relational, 47
  set, 47
  shl, 45
  shr, 45
  string, 46
  structure member selector, 198
  types of, 44
  unary arithmetic, 44
  xor, 45
or operator, 45
Ord function, 12, 14, 15, 181
order of evaluation, 183
ordering between two string-type values, 17
ordinal
  types
    predefined, 12
    user-defined, 12
ordinality
  defined, 12
  enumerated constant, 15
  finding enumerated type's value, 15
  returning, 12
    Char values, 14
Origin variable, 145
Output text-file
  in WinCrt unit, 141
Output text-file variable, 141
  and WinCrt unit, 142
overflow checking, 236
override directive, 91
overriding methods, 91
overriding properties, 107

P

$P compiler directive, 78, 235
packed
  reserved word, 17
  string type, 18
  strings, comparing, 48
parameters
  actual, 56
  constant, 76
  floating-point, 172
  formal, 56
  open, 78
    array, 79
    string, 78
  passing, 56
  types, 75
  untyped, 77
  value, 76, 172
  variable, 77
  virtual method, 176
passing parameters
  by reference, 171
  by value, 171
passing parmeters, 171
passing string variables of varying sizes, 17
PChar data type, 22
PERMANENT code segment attribute, 160
pointer (^) symbol, 21, 22, 33
Pointer data type, 22
pointers
  comparing, 48
  types, 21, 165
  values, 33
  variables, 33
pointer-type constants, 39
Port array, 169
ports, accessing, 169
PortW array, 169
pound (#) character, 7
precedence of operators, 41, 44
precision
  of real-type values, 16
  rules of arithmetic, 13
Pred function, 12, 181
predecessor of a value, returning, 12
PrefixSeg variable, 137
PRELOAD code segment attribute, 160
private
  directive, 5
  procedures and functions, 125
private sections
  classes, 90
PROC directive, 208
procedural
  types
    type compatibility of, 24
    variable typecasts, 35
procedural-type constants, 40
procedure
  declaration syntax, 69
  declarations
    assembler, 72
    external, 72
    forward, 72
    inline, 73
  headings, 70

  - statements, 56
- procedure and function declaration part, 66
- procedures, 69. *See also* procedures and
  - entry/exit code, built-in assembler, 203
  - exit, 137
  - external, 152
  - far, 174
  - near, 174
  - nested, 174
  - parameters, built-in assembler, 203
  - stack frame, built-in assembler, 203
- procedures and functions. *See also* procedures; functions
  - importing, 130
  - nested, 24
  - written in assembler, 207
    - call model, 207
- program
  - comments, 7
  - heading, 123
  - lines, maximum length of, 8
  - parameters, 123
  - syntax, 123
  - termination, 178
- programs
  - rebuilding, 219
- properties, 101
  - access, 102
  - array, 104
  - defined, 87
  - definitions, 102
    - index specifiers, 105
    - overrides, 107
    - storage specifiers, 106
  - overriding, 107
- property override, 107
- protected sections
  - classes, 90
- Ptr function, 21, 33, 181
- public
  - directive, 5
  - procedures and functions, 124
- PUBLIC directives, 207
- public sections
  - classes, 89
- published sections
  - classes, 89

Q

- /Q command-line option, 220
- qualified
  - identifiers, 5
  - method
    - identifiers, 33, 50
- qualified method identifer, 94
- qualifier syntax, 32
- quiet mode command-line option, 220

R

- $R compiler directive, 237
- $R filename directive, 237
- range
  - checking, 156
    - compile-time, 184
  - finding higest value, 12
  - finding lowest value, 12
  - of real-type values, 16
- range checking, 217
  - command, 237
  - compiler switch, 237
  - option, 237
- read directive, 102
- Read procedure
  - for values not of type Char, 141
  - text files, 139, 140
- ReadBuf function, 144
- ReadKey function, 144
- Readln procedure, 139
- real
  - data types, 16
  - numbers, 16, 149, 163
- Real data type, 16
- real-type operations
  - 80x87 floating type, 16
  - software floating point, 16
- record
  - scope, 67
  - types, 19
- records, 19, 33, 38, 166
  - fields, 33
  - variant part, 19
- record-type constant syntax, 38
- recursive loop. *See* recursive loop
- redeclaration of variables, 29
- references
  - class, 86
- registers
  - and inline statements, 212
  - AX, 173, 213
  - BP, 178
  - built-in assembler, 195, 198
  - BX, 173
  - DS, 178
  - DX, 173
  - SP, 178
  - SS, 178
  - use, built-in assembler, 188
  - using, 173, 178, 188
- register-saving conventions, 178
- regs, 195
- relaxed string parameter checking, 241
- relocation expressions, built-in assembler, 199
- Rename procedure, 139
- repeat statement syntax, 59
- repetitive statement syntax, 59
- reserved words, 4
  - built-in assembler, 192
  - defined, 4
  - external, 211
  - how identified in manuals, 4
  - list of, 4
- Reset procedure, 139, 140
- resident option in exports clause, 135
- Resource directories command-line option, 221
- resource file, 237
- resources
  - including, 237
- RET instruction, built-in assembler, 189
- RETF instruction, built-in assembler, 189
- RETN instruction, built-in assembler, 189
- return character, defined, 3
- returning
  - Char values, 14

the ordinality of a value, 12
the predecessor of a value, 12
the successor of a value, 12
Rewrite procedure, 139, 140
right
brace special symbol, 4
bracket special symbol, 4
RmDir procedure, 139
Round function, 181
rules
governing boolean variables, 14
run-time
errors, 178
in a DLL, 138
run-time errors
Find Error command and, 219
finding, 219
run-time type information, 89

S

$S compiler directive, 30, 137, 238, 239
scale factor syntax diagram, 6
scope
block, 66
class, 88
object, 67
record, 67
rules of, 66
type identifiers, 11
unit, 67
within methods, 94
ScreenSize variable, 143, 145
ScrollTo procedure, 144
Seek procedure, 139, 140
SeekEof function, 139
SeekEoln function, 140
segment
attributes, 159
sub-allocator, 161
Segment size compiler directive, 238
segments, 207
Self identifer, 87
Self identifier, 110
Self parameter, 176
separating tokens, 3
separators, defined, 3
set
constructors
syntax, 50
membership
testing, 49
operators, 47
types, 20, 165
set constructors, 42
sets. *See also* set
comparing, 49
small, 183
SetTextBuf procedure, 140
set-type constants, 39
Shift instructions faster than multiply or divide, 184
shl operator, 45
short-circuit Boolean evaluation, 46, 182, 227
Shortint data type, 12
shr operator, 45
signed number syntax diagram, 6
significand, 163
simple
expression syntax, 43
statement syntax, 55
types
comparing, 48
simple-type constants, 36
single character special symbols, 4
Single data type, 16, 150, 164
size
of a given string, finding, 17
of structured types, maximum, 17
SizeOf function, 79
small sets, 183
smart callbacks, 71, 239
smart linking, 185
software
floating-point
model, 16
restrictions, 16
source debugging compiler switch, 228
SP register, 178
space characters, 3
special symbols
built-in assembler, 196
character pairs listed, 4
single characters listed, 4
SS register, 178
stack, 161
80x87, 152
changing size of, 161
checking, 137
checking switch directive, 239
frame, built-in assembler use of, 203
overflow, 30
switch directive, 239
passing parameters and, 171
segment, 30
DLLs and, 138
size, 235
Stack Checking
command, 239
option, 239
standard, 139
directives, 4
procedure or function
used as a procedural value, 24
statement part syntax, 66
statements, 55
assignment, 55
case, 58
compound, 57
conditional, 57
for, 61
goto, 56
if, 57
procedure, 56
repeat, 59
repetitive, 59
simple, 55
structured, 57
while, 60
with, 62
static
data area, 161
importing, 132
linking, 129
static methods, 90
storage specifiers, 106
stored directive, 106
storing
null-terminated strings, 19
strict string parameter checking, 241
string. *See also* strings

  - constants, built-in assembler, 194
  - literals, assigning to PChar, 154
  - operator, 46
  - type
    - default size, 17
    - defined, 17
    - ordering between two values, 17
    - packed, 19
  - typed constants, 36
  - variables, 32
- String Var Checking
  - command, 241
  - option, 241
- string variables
  - passing, 17
- strings. *See also* string
  - character, 7
    - length of, 7
  - comparing, 48
  - concatenating, 46
  - embedding control characters in, 7
  - length byte, 165
  - maximum length, 165
  - null, 7, 17
  - null-terminated, 19
  - relaxed parameter checking of, 241
  - strict parameter checking of, 241
  - types, 165
- Strings unit, 153
- structure member selector operator, 198
- structured
  - statement syntax, 57
- structured types, 17
- structured-type constants, 37
- subrange type, 15
- Succ function, 12, 181
- successor of a value, returning, 12
- Swap function, 181
- switch compiler directives, 225
- symbol reference compiler switch, 240
- symbols, 3, 196
  - built-in assembler, 196
  - conditional, 243
  - CPU, 244
  - invalid, built-in assembler, 196
  - list of special, 4
  - local information, 234
  - reserved, built-in assembler, 195
  - scope access, built-in assembler, 198
  - special, built-in assembler, 196
- syntax
  - extended, 230
- System unit, 123

T

- $T compiler directive, 26, 49
- /T command-line option, 220
- tag field, 20
- task header, 160
- TClass type, 111
- term syntax, 43
- terminating, 178
- testing
  - set membership, 49
- text
  - files, 141
    - devices
      - drivers, 145
- Text type, 141
- TObject class, 88
  - Destroy method, 97
  - Free method, 97
- TObject type, 111
- tokens, 3
  - categories of, 3
  - defined, 3
  - separating, 3
- TrackCursor procedure, 144
- trapping I/O errors, 233
- True predefined constant identifier, 14
- Trunc function, 181
- Truncate procedure, 140
- TTextRec records, 145, 166
- Turbo Assembler, 208
  - 80x87 emulation and, 152
- type. *See also* types
  - declaration, 11
  - declaration part syntax, 66
  - identifier, 11
- type checking
  - strings and, 241
- type checking, built in assembler, 200
- typecasting integer-type values, 14
- typecasts, value, 34, 51
- typed
  - constant
    - syntax, 35
  - files, 166
- types
  - array, 18, 166
  - boolean, 14, 163
  - Byte, 12
  - ByteBool, 163
  - Char, 14, 162
  - class, 85
  - class reference, 108
  - Comp, 12, 150
  - compatibility, 24, 25
  - declaration part, 27
  - Double, 12, 150
  - enumerated, 14, 163
  - Extended, 12, 150
  - file, 21
  - floating-point, 16, 150, 163
    - Comp, 165
    - comparing values of, 151
    - Double, 164
    - Extended, 164
    - Single, 164
  - host, 15
  - identical, 24
  - identity, 25
  - Integer, 12, 162
    - converting through typecasting, 14
    - format of, 13
    - range of, 13
  - LongBool, 14, 163
  - Longint, 12
  - major classes, 11
  - ordinal
    - characteristics of, 12
    - predefined, 12
    - user-defined, 12
  - packed string, 19
  - PChar, 22
  - Pointer, 21, 22, 165
  - procedural, 52
  - Real, 12, 16
  - real numbers, 163

record, 19, 166
set, 20, 165
Shortint, 12
Single, 12, 150
string, 17, 165
subrange, 15
Text, 141
Word, 12
WordBool, 14, 163

U

/U command-line option, 221
unary
arithmetic operators, 44
operands, 41
$UNDEF compiler directive, 241, 243
Unit Directories
option, 221
Unit segment grouping compiler directive, 231
unit syntax, 124
units
heading, 124
identifiers, 5
implementation part, 125
import, 131
indirect references, 126
initialization part, 125
interface part, 124
scope of, 67
Strings, 153
unit directories option, 221
uses clause, 123
version number, 126
WinCrt, 142
unsigned
constant syntax, 42
integer syntax diagram, 6
number syntax diagram, 6
real syntax diagram, 6
untyped
files, 142, 166
parameters, 77
uses clause, 123

V

$V compiler directive, 241
/V command-line option, 222
value
parameters, 76, 172
typecast syntax, 51
var
declaration section, 185
parameters, 77, 171
built-in assembler and, 197
string checking
compiler switch, 241
variable. *See also* variables
declaration part syntax, 66
declaration syntax, 29
parameters, 77
reference
qualifiers, 31
syntax, 31
typecasts, 34
and procedural types, 35
syntax, 34
variables
absolute, 30
array, 32
declarations, 29
dynamic, 21, 33
global, 30
initialized in assembler, 207
initializing, 35
local, 30
parameters, 171
pointer, 33
record, 33
references, 31
string, 32
variant part of records, 19
VER80 symbol, 244
virtual
method
parameter, 176
virtual directive, 91
virtual methods, 91
virtual versus dyanmic methods, 92
visibility of class components, 89

W

$W compiler directive, 241
WEP exported function, 136
wep_Free_DLL value for ExitCode, 136
wep_System_Exit value for ExitCode, 136
WhereX function, 144
WhereY function, 144
while statement syntax, 60
wincrt, 142
WinCrt unit, 141, 142
editing keys in, 144
using the, 142
variables in, 144
windows stack frames, 241
Windows symbols, 244
WindowSize variable, 145
WindowTitle variable, 145
WinOrg variable, 144
with statement syntax, 62
with statements, 88, 93
word alignment, automatic, 184
Word data type, 12
WordBool data type, 14, 163
write directive, 102
Write procedure, 140
for values not of type
Char, 141
WriteBuf procedure, 144
WriteChar procedure, 144
Writeln procedure, 140

X

$X compiler directive, 7, 19, 22, 31, 46, 230
xor operator, 45

Y

$Y compiler directive, 240

Z

zero-based character arrays, 37, 153, 155

Object Pascal Language Guide

VERSION 1.0

Delphi

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

Printed in the U.S.A.

1E0R395
9596979899-9 8 7 6 5 4 3 2 1
W1

Contents

Introduction 1
What's in this manual? 1
Syntax Diagrams 1

Chapter 1
Tokens 3
Special symbols 3
Reserved words and standard directives 4
Identifiers 5
Numbers 6
Labels 6
Character strings 7
Comments 7
Program lines 8

Chapter 2
Constants 9

Chapter 3
Types 11
Simple types 11
Ordinal types 12
Integer types 12
Boolean types 14
Char type 14
Enumerated types 14
Subrange types 15
Real types 16
80x87 floating point 16
Software floating point 16
String types 17
Structured types 17
Array types 18
Record types 19
Set types 20
File types 21
Pointer types 21
Type Pointer 22
Type PChar 22
Procedural types 22
Global procedure pointers 22
Method pointers 23
Procedural values 23
Procedural type compatibility 24
Identical and compatible types 24
Type identity 25
Type compatibility 25
Assignment compatibility 26
The type declaration part 27

Chapter 4
Variables and typed constants 29
Variable declarations 29
The data segment 30
The stack segment 30
Absolute variables 30
Variable references 31
Qualifiers 32
Arrays, strings, and indexes 32
Records and field designators 33
Object component designators 33
Pointers and dynamic variables 33
Variable typecasts 34
Typed constants 35
Simple-type constants 36
String-type constants 37
Structured-type constants 37
Array-type constants 37
Record-type constants 38
Set-type constants 39
Pointer-type constants 39
Procedural-type constants 40

Chapter 5
Expressions 41
Expression syntax 41
Operators 44
Arithmetic operators 44
Logical operators 45
Boolean operators 45
String operator 46
Character-pointer operators 46
Set operators 47
Relational operators 47
Comparing simple types 48

        - Comparing strings 48
        - Comparing packed strings 48
        - Comparing pointers and references 48
        - Comparing character pointers 48
        - Comparing sets 49
        - Testing set membership 49
    - Class operators 49
    - The @ operator 49
        - @ with a variable 49
        - @ with a procedure, function, or method 50
- Function calls 50
- Set constructors 50
- Value typecasts 51
- Procedural types in expressions 52

Chapter 6
Statements 55

- Simple statements 55
    - Assignment statements 55
    - Procedure statements 56
    - Goto statements 56
- Structured statements 57
    - Compound statements 57
    - Conditional statements 57
        - If statements 57
        - Case statements 58
    - Repetitive statements 59
        - Repeat statements 60
        - While statements 60
        - For statements 61
    - With statements 62

Chapter 7
Blocks, locality, and scope 65

- Blocks 65
- Rules of scope 66
    - Block scope 66
    - Record scope 67
    - Class scope 67
    - Unit scope 67

Chapter 8
Procedures and functions 69

- Procedure declarations 69
    - Near and far declarations 70
    - Export declarations 71
    - cdecl declarations 71
    - Forward declarations 72
    - External declarations 72
    - Assembler declarations 73
    - Inline declarations 74
- Function declarations 74
- Parameters 75
    - Value parameters 76
    - Constant parameters 76
    - Variable parameters 77
    - Untyped parameters 77
    - Open parameters 78
        - Open-string parameters 78
        - Open-array parameters 79
- Open-array constructors 81
- Type variant open-array parameters 81

Chapter 9
Class types 85

- Instances and references 86
- Class components 87
    - Fields 87
    - Methods 87
    - Properties 87
- Inheritance 87
- Components and scope 88
- Forward references 88
- Class type compatibility rules 88
- Component visibility 89
    - Public components 89
    - Published components 89
    - Protected components 90
    - Private components 90
- Static methods 90
- Virtual methods 91
- Dynamic methods 92
- Abstract methods 92
- Method activations 93
- Method implementations 93
- Constructors and destructors 94
    - Constructors 95
    - Destructors 96
- Class operators 98
    - The is operator 98
    - The as operator 98
- Message handling 99

  - Message handler declarations 99
  - Message handler implementations 99
  - Message dispatching 100
- Properties 101
  - Property definitions 102
  - Property access 102
  - Access specifiers 103
  - Array properties 104
  - Index specifiers 105
  - Storage specifiers 106
  - Property overrides 107
- Class-reference types 108
- Constructors and class references 109
- Class methods 110
- The TObject and TClass types 111

Chapter 10
Exceptions 113
- Using exception handling 113
- Exception declarations 114
- The raise statement 114
- The try...except statement 115
  - Re-raising exceptions 117
  - Nested exceptions 118
- The try...finally statement 119
- Exit, Break, and Continue procedures 120
- Predefined exceptions 120
- Exception handling support routines 122

Chapter 11
Programs and units 123
- Program syntax 123
  - The program heading 123
  - The uses clause 123
- Unit syntax 124
  - The unit heading 124
  - The interface part 124
  - The implementation part 125
  - The initialization part 125
  - Indirect unit references 126
- Circular unit references 127

Chapter 12
Dynamic-link libraries 129
- What is a DLL? 129
- Using DLLs 130
  - Import units 131
  - Static and dynamic imports 132
- Writing DLLs 133
  - The export procedure directive 134
  - The exports clause 135
  - Library initialization code 135
- Library programming notes 137
  - Global variables in a DLL 137
  - Global memory and files in a DLL 137
  - DLLs and the System unit 137
  - Run-time errors in DLLs 138
  - DLLs and stack segments 138

Chapter 13
Input and output 139
- File input and output 140
  - Text files 141
  - Untyped files 142
- Input and output with the WinCrt unit 142
  - Using the WinCrt unit 142
    - Special characters 143
    - Line input 144
  - WinCrt procedures and functions 144
  - WinCrt unit variables 144
- Text-file device drivers 145
    - The Open function 146
    - The InOut function 146
    - The Flush function 146
    - The Close function 146

Chapter 14
Using the 80x87 149
- The 80x87 data types 150
- Extended range arithmetic 150
- Comparing reals 151
- The 80x87 evaluation stack 152
- Detecting the 80x87 152
- Emulation in assembly language 152
- Exception statements 152

Chapter 15
Using null-terminated strings 153
- What is a null-terminated string? 153
- Using null-terminated strings 153
  - Character pointers and string literals 154
  - Character pointers and character arrays155
  - Character pointer indexing 155

Null-terminated strings and standard procedures 157

Chapter 16

Memory issues 159

Windows memory management.............. 159
Code segments...................................... 159
Segment attributes 159
MOVEABLE or FIXED................ 159
PRELOAD or DEMANDLOAD.. 160
DISCARDABLE or PERMANENT160
Changing attributes 160
The automatic data segment................ 160
The heap manager 161
Internal data formats 162
Integer types 162
Char types 162
Boolean types....................................... 163
Enumerated types................................ 163
Floating-point types 163
The Real type 163
The Single type 164
The Double type 164
The Extended type 164
The Comp type 165
Pointer types 165
String types 165
Set types 165
Array types 166
Record types 166
File types 166
Procedural types................................... 167
Class types 167
Class reference types............................ 168
Direct memory access.............................. 168
Direct port access..................................... 169

Chapter 17

Control issues 171

Calling conventions 171
Variable parameters 171
Value and constant parameters........... 172
Open parameters 172
Function results 173
NEAR and FAR calls........................... 173
Nested procedures and functions 174
Method calling conventions 175
Constructors and destructors................176
Entry and exit code 176
Register-saving conventions..................178
Exit procedures...178

Chapter 18

Optimizing your code 181

Constant folding 181
Constant merging 181
Short-circuit evaluation.............................182
Constant parameters 182
Redundant pointer-load elimination.........182
Constant set inlining..................................183
Small sets 183
Order of evaluation 183
Range checking 184
Shift instead of multiply or divide 184
Automatic word alignment........................184
Eliminating dead code...............................184
Smart linking..185

Chapter 19

The built-in assembler 187

The asm statement.....................................187
Register use..188
Assembler statement syntax 188
Labels 188
Instruction opcodes...............................189
RET instruction sizing 189
Automatic jump sizing 189
Assembler directives.............................190
Operands..192
Expressions..193
Differences between Object Pascal and Assembler expressions 193
Expression elements..............................194
Constants 194
Numeric constants 194
String constants 194
Registers...195
Symbols 196
Expression classes 198
Expression types....................................200
Expression operators 201
Assembler procedures and functions........203

Chapter 20

Linking assembler code 207

Turbo Assembler and Delphi.................... 208
Examples of assembly language routines 209
Assembly language methods.................... 211
Inline machine code.................................. 211
Inline statements.................................. 211
Inline directives 213

Appendix A

The command-line compiler 215

Command-line compiler options............. 215
Compiler directive options.................. 217
The switch directive option 217
The conditional defines option 218
Compiler mode options 218
The make (/M) option 219
The build all (/B) option 219
The find error (/F) option 219
The link buffer (/L) option 219
The quiet (/Q) option 220
Directory options................................. 220
The DSL & CFG Directory (/T) option 220
The EXE & DCU directory (/E) option 221
The include directories (/I) option 221
The unit directories (/U) option 221
The resource directories (/R) option 221
The object files directories (/O) option 221
Debug options...................................... 221
The map file (/G) option 221
The debugging (/V) option 222
The DCC.CFG file 222

Appendix B

Compiler directives 225

Align data 226
Boolean evaluation.................................... 227
Code segment attribute 227
Debug information.................................... 228
DEFINE directive...................................... 229
Description 229
ELSE directive 229
ENDIF directive 230
Extended syntax..230
Force Far calls..230
Generate 80286 Code 231
Group unit segments 231
IFDEF directive 232
IFNDEF directive.......................................232
IFOPT directive 232
Include file..233
Input/output checking.............................233
Link object file 233
Local symbol information 234
Memory allocation sizes...........................235
Numeric coprocessor................................235
Open String Parameters 235
Overflow checking....................................236
Pentium safe FDIV operations..................236
Range checking 237
Resource file 237
Run-time type information 238
Segment size preference 238
Smart callbacks 239
Stack-overflow checking 239
Symbol reference information 240
Type-checked pointers 240
UNDEF directive 241
Var-string checking...................................241
Windows stack frames 241
Word sized enumeration types................242
Using conditional compilation directives .242
Conditional symbols.................................243

Appendix C

Error Messages 245

Compiler error messages 245
Run-time errors...273
File errors...273
I/O errors...275
Fatal errors...275

Index 277

Tables

1-1 Object Pascal reserved words 4
1-2 Object Pascal directives 5
3-1 Fundamental integer types 13
3-2 Generic integer types for 16-bit implementations of Object Pascal 13
3-3 Generic integer types for 32-bit implementations of Object Pascal 13
3-4 Real data types 16
5-1 Precedence of operators 41
5-2 Binary arithmetic operations 44
5-3 Unary arithmetic operations 44
5-4 Logical operations 45
5-5 Boolean operations 45
5-6 String operation 46
5-7 Permitted PChar constructs 47
5-8 Set operations 47
5-9 Relational operations 47
8-1 Type variant open-array expressions83
10-1 Predefined exception classes 121
10-2 Exception support routines 122
13-1 Input and output procedures and functions 139
13-2 Special characters in the WinCrt window 143
13-3 WinCrt procedures and functions 144
13-4 WinCrt variables 144
16-1 Virtual Method Table layout 168
19-1 Built-in assembler reserved words 192
19-2 String examples and their values 195
19-3 CPU registers 195
19-4 Values, classes, and types of symbols 197
19-5 Predefined type symbols 201
19-6 Summary of built-in asssembler expression operators 201
19-7 Definitions of built-in assembler expression operators 202
A-1 Command-line options 216
B-1 TestFDIV values 237