7.0

TURBO PASCAL®

LANGUAGE GUIDE

- LANGUAGE DEFINITION
- STANDARD UNITS
- MEMORY MANAGEMENT
- ASSEMBLY LANGUAGE

BORLAND

Turbo Pascal®

Version 7.0

Language Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

Copyright © 1983, 1992 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Windows, as used in this manual, shall refer to Microsoft's implementation of a windows system. Other brand and product names are trademarks or registered trademarks of their respective holders.

PRINTED IN THE USA.
R1
10 9 8

CONTENTS

TABLES

F I G U R E S

INTRODUCTION

Read the Introduction in the User's Guide for an overview of the entire Turbo Pascal documentation set and how to use the Turbo Pascal manuals most effectively.

This manual is about the Turbo Pascal language. It

- Presents the formal definition of the Turbo Pascal language
- Introduces the run-time library and tells you how to use the units that make it up
- Describes what goes on inside Turbo Pascal in regards to memory, data formats, calling conventions, input and output, and automatic optimizations
- Explains how to use Turbo Pascal with assembly language

You'll find this manual most useful if you are an experienced Pascal programmer.

Read the *User's Guide* if

- You want to know how to install Turbo Pascal
- You've used Turbo Pascal before and you want to know what is new in this release
- You're not familiar with Borland's integrated development environment (the IDE)
- You want to know how to use the integrated debugger
- You want to refresh your knowledge about pointers
- You are new to object-oriented programming

Read the *Programmer's Reference* to look up reference material on

- The run-time library
- Compiler directives
- Error messages
- The command-line compiler
- The editor

What's in this manual

This book is split into four parts: language grammar, the run-time library, advanced programming issues, and using assembly language with Turbo Pascal.

Part I, "The Turbo Pascal language," defines the Turbo Pascal language. First you're introduced to the overall structure of a Turbo Pascal program; then you examine each element of a program in detail.

Part II, "The run-time library," contains information about using all the standard units: the *System*, *Dos*, *WinDos*, *Strings*, *Crt*, *Overlay*, and *Graph* units.

Part III, "Inside Turbo Pascal," presents technical information for advanced users about

- How Turbo Pascal uses memory
- How Turbo Pascal implements program control
- Using the 80x87
- Optimizing your code

Part IV, "Using Turbo Pascal with assembly language," explains how to use the built-in assembler and how to link your Turbo Pascal programs with code written in Turbo Assembler.

P A R T 1

The Turbo Pascal language

C H A P T E R

1

What is a Turbo Pascal program?

The next several chapters present the formal definition of the Turbo Pascal language. Each chapter discusses an element of Turbo Pascal. Together, these elements make up a Turbo Pascal program.

It's difficult to gain an understanding of the whole by examining only the parts, however. This chapter presents an overview of a Turbo Pascal program and omits the details. It gives you a brief description of each of the elements of a program and then shows you how they all fit together. You can then refer to Chapters 2 through 10 to find the details of the language.

A Turbo Pascal program

In its simplest form, a Turbo Pascal program is made up of a *program heading*, which names the program, and the *main program block*, which accomplishes the purpose of the program. Within the main program block is a section of code that occurs between two key words: **begin** and **end**. Here is a very simple program that illustrates these concepts:

```
program Welcome;
begin
   Writeln('Welcome to Turbo Pascal');
end.
```

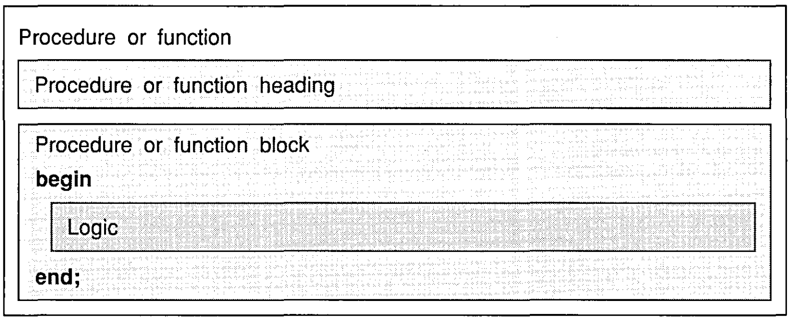
The first line is the program heading, which names the program. The remainder of the program is the code that starts with **begin** and stops with **end**. Although this particular code section contains only one line, it could contain many. In any Turbo Pascal program, all the action occurs between **begin** and **end**.

Procedures and functions

The code between the last **begin** and **end** in a program drives the logic of the program. In a very simple program, this section of code might be all you need. In larger, more complex programs, putting all your code here can make your program harder to read and understand—and more difficult to develop.

Procedures and *functions* let you divide the logic of a program into smaller, more manageable chunks, and are similar to subroutines in some other languages. All the action in a procedure or function occurs in the code between its **begin** and **end** just like in the main program block. Each of these segments of code performs a small, discrete task.

Figure 1.1
Procedure or function diagram



If you find your program does the same thing many times, you might want to put the logic into a procedure or function. You write the code in a procedure or function once and your program can use it as often as necessary.

Here is an example of a function. This *GetNumber* function gets a number from the user:

```
function GetNumber: Real;
var
   Response: Real;
begin
   Write('Enter a number: ');
   Readln(Response);
   GetNumber := Response;
end;
```

A procedure or function must appear before the main code section in the main program block. The main code section can then use the procedure or function.

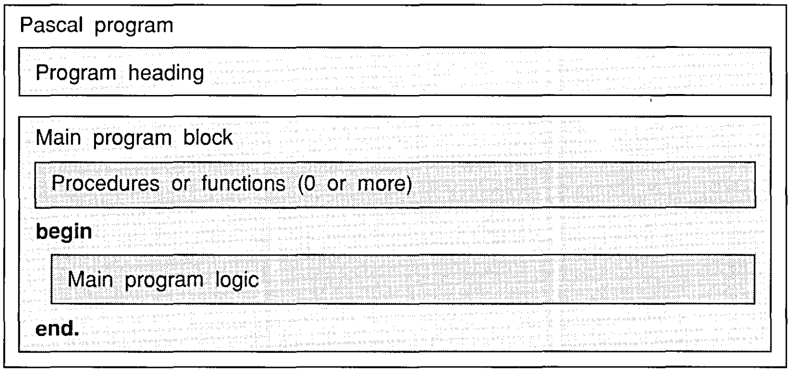


Figure 1.2
Simple Pascal program diagram

The following example is an outline of a program that uses the *GetNumber* function. The programmer has divided the logic of this program into three tasks:

1. Get a number from the user.
2. Perform the necessary calculations with the user-supplied number.
3. Print a report.

The main logic of the program is found between the last **begin** and **end**.

```
program Report;

var
   A: Real;

{more declarations}
   ⋮
  function GetNumber: Real;
  var
    Response: Real;
  begin
    Write('Enter a number: ');
    Readln(Response);
    GetNumber := Response;
  end;

  procedure Calculate(X: Real);
     ⋮
  procedure PrintReport;
     ⋮
begin
   A := GetNumber;
   Calculate(A);
   PrintReport;
end.
```

The primary logic in this program is very simple to understand. All the details are hidden within the bodies of the procedures and functions. Using procedures and functions encourages you to think about your program in a logical, modular way.

Statements

The code section between **begin** and **end** contains statements that describe the actions the program can take and is called the *statement part*. These are examples of statements:

```
A := B + C;                      {Assign a value}

Calculate(Length, Height);       {Activate a procedure}

if X < 2 then                    {Conditional statement}
   Answer := X * Y;

begin                            {Compound statement}
   X := 3;
   Y := 4;
   Z := 5;
end;
```

```
while not EOF(InFile) do          {Repetitive statement}
begin
   Readln(InFile, Line);
   Process(Line);
end;
```

Simple statements can either assign a value, activate a procedure or function, or transfer the running of the program to another statement in the code. The first two examples shown in the examples are simple statements.

Structured statements can be compound statements that contain multiple statements, conditional and repetitive statements that control the flow of logic within a program, and **with** statements that simplify access to data in a record.

You might compare a Pascal statement to a sentence in a human language such as English, Danish, or Greek. Simple Pascal statements and simple human sentences hold one complete thought. Structured Pascal statements and complex sentences contain more complicated logic.

Expressions

Just as a sentence is made up of phrases, so is a Pascal statement made up of expressions. The phrases of a sentence are made up of words, and the expressions of a statement are composed of elements called factors and operators. Expressions usually compare things or perform arithmetic, logical, or Boolean operations.

Just as phrases in a human language can be made up of smaller phrases, so can expressions in Pascal be made up of simpler expressions. You can read about all the combinations of factors and operators in Chapter 6 that make up expressions. They can be quite complex. For now, it might help to see some examples of expressions:

```
X + Y
Done <> Error
I <= Length
-X
```

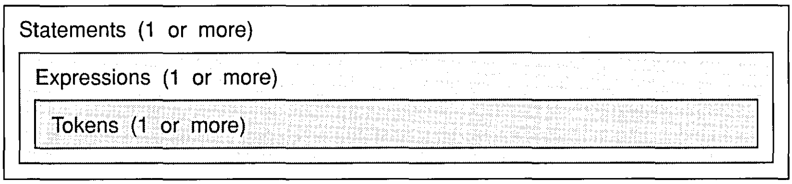
Tokens

Tokens are the smallest meaningful elements in a Pascal program. They make up the factors and operators of expressions. Tokens are special symbols, reserved words, identifiers, labels, numbers, and string constants; they are akin to the words and punctuation of a written human language. These are examples of Pascal tokens:

```
function                {reserved word}
(                       {special symbol}
:=                      {special symbol}
Calculate               {identifier for a procedure}
9                       {number}
```

Here is an illustration of a statement. You can see that statements are made up of expressions, which are made up of tokens.

Figure 1.3
Statement diagram



Types, variables, constants, and typed constants

A *variable* can hold a value that can change. Every variable must have a *type*. A variable's type specifies the set of values the variable can have.

For example, this program declares that variables *X* and *Y* are of type *Integer*; therefore, the only values *X* and *Y* can contain are integers, which are whole numbers. Turbo Pascal displays an error message if your program tries to assign any other type of value to these variables.

```
program Example;

const
  A = 12;                {Constant A never changes in value}
  B: Integer = 23;       {Typed constant B gets an initial value}

var
  X, Y: Integer;         {Variables X and Y are type Integer}
  J: Real;               {Variable J is type Real}

begin
  X := 7;                {Variable X is assigned a value}
  Y := 8;                {Variable Y is assigned a value}
  X := Y + Y;            {The value of variable X changes}
  B := 57;               {Typed constant B gets a new value}
  J := 0.075;            {Variable J gets a floating-point value}
end.
```

In this simple and not very useful program, *X* is assigned the value 7 originally; two statements later it is assigned a new value, *Y* + *Y*. The value of a variable can vary.

A is a *constant*. The program gives it a value of 12 and this value can't change—its value remains constant throughout the program.

B is a *typed constant*. It's given a value when it's declared, but it's also given a type of *Integer*. You can think of a typed constant as a variable with an initial value. The program can later change the initial value of *B* to some other value.

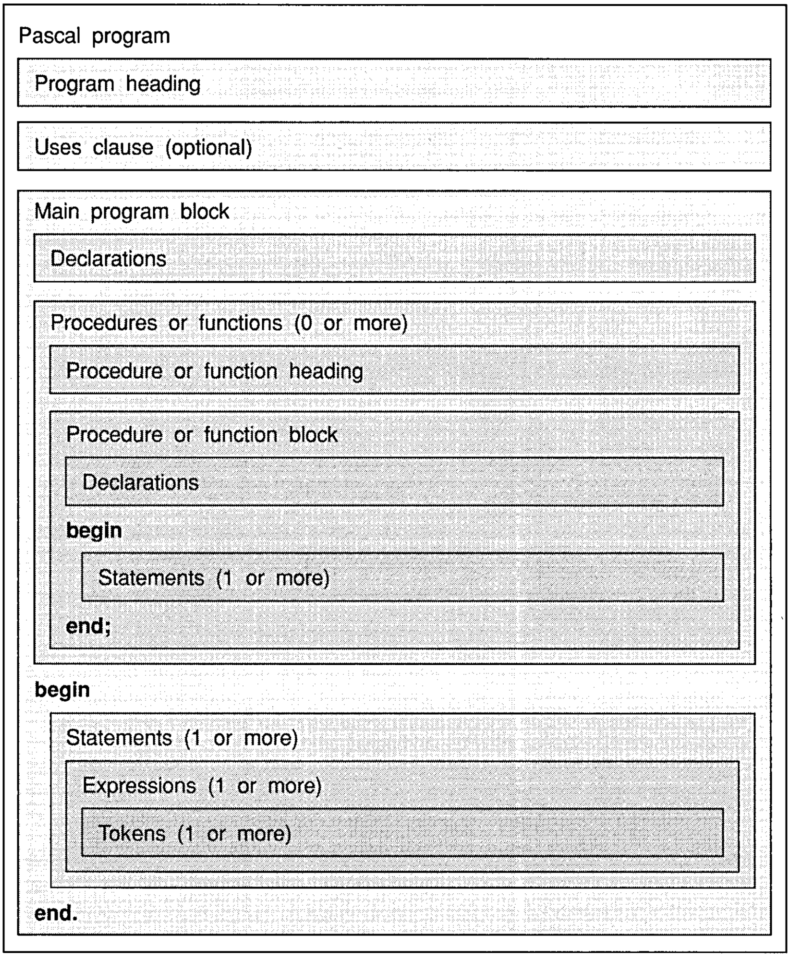
The part of this program that declares the constants and variables is called the *declaration part*.

If you'll look back at the code example on page 7, you'll see that the function *GetNumber* has a declaration part that declares a variable. Procedures and functions can contain a declaration part just as a program or unit can.

Putting it all together

Now that you've been introduced to the primary components of a Turbo Pascal program, you need to see how they all fit together. Here's an illustration of a Turbo Pascal program:

Figure 1.4
An expanded Pascal program



The program heading, the optional **uses** clause (we'll talk about this in the next section), and the main program block make up a Pascal program. Within the main program block can exist the smaller blocks of procedures and functions. Although the diagram doesn't show this, procedures and functions can be nested within other procedures and functions. In other words, blocks can contain other blocks.

Combined with other tokens and blank spaces, tokens make up expressions which make up statements.

In turn, statements combined with declaration parts make up blocks, either the main program block or a block in a procedure or function.

Units

A Turbo Pascal program can use blocks of code in separate modules called *units*. You can think of a unit as a mini-program your application can use. Like a program, it has a heading, called a unit heading, and a main block that contains a code section delineated by **begin** and **end**.

Any Turbo Pascal main program block can include a line that enables the program to use one or more units. For example, if you are writing a program called *Colors* and you want to change the color of the text as it appears on your screen, you can specify that your program use the standard *Crt* unit that is part of the Turbo Pascal run-time library:

```
program Colors;
uses Crt;
begin
   :
end.
```

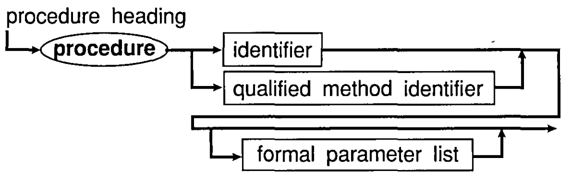
The **uses** Crt line tells Turbo Pascal to include the *Crt* unit in the executable program. The *Crt* unit contains all the necessary code to change the color of the text in your program, among other things. Simply by including **uses** Crt, your program can use all the procedures and functions in the *Crt* unit. If you put all the code required to create the functionality of the *Crt* unit within your program, it would be a lot more work, and it would sidetrack you from the main purpose of your program.

Turbo Pascal's run-time library includes several units you'll find useful. For example, use the *Dos* unit and your program has access to several operating system and file-handling routines.

You can also write your own units. Use them to divide large programs into logically related modules. Code you place in a unit can be used by any program. You only have to write the code once, then you can use it many times.

Syntax diagrams

As you read Chapters 2 through 10, which define the Turbo Pascal language, 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 identifier to the end of the procedure heading, or you can follow it to the formal parameter list before reaching the end.

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

C H A P T E R

2

Tokens

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

Separators can't be part of tokens except in string constants.

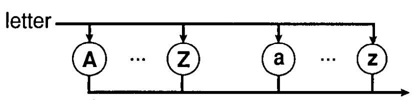
A Pascal program is made up of tokens and *separators*. A separator is either a blank or a comment. Two adjacent tokens must be separated by one or more separators if each token is a reserved word, an identifier, a label, or a number.

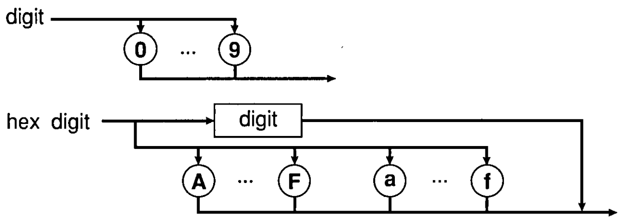
Special symbols

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

Following are Turbo Pascal's reserved words:

Table 2.1
Turbo Pascal reserved words

and	**file**	**not**	**then**
array	**for**	**object**	**to**
asm	**function**	**of**	**type**
begin	**goto**	**or**	**unit**
case	**if**	**packed**	**until**
const	**implementation**	**procedure**	**uses**
constructor	**in**	**program**	**var**
destructor	**inherited**	**record**	**while**
div	**inline**	**repeat**	**with**
do	**interface**	**set**	**xor**
downto	**label**	**shl**	
else	**mod**	**shr**	
end	**nil**	**string**	

The following are Turbo Pascal's standard (built-in) 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 2.2
Turbo Pascal directives

absolute	**far**	**near**	**virtual**
assembler	**forward**	**private**	
external	**interrupt**	**public**	

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

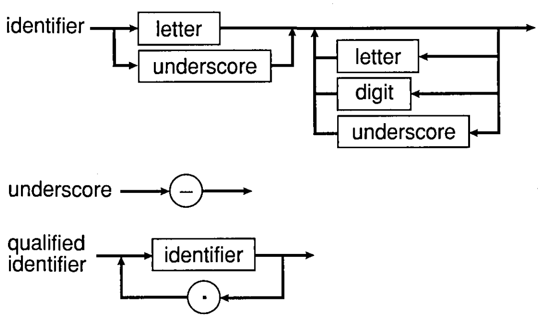
Identifiers

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.

Units are described in Chapter 5 of the User's Guide and Chapter 10 of this book.

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*.



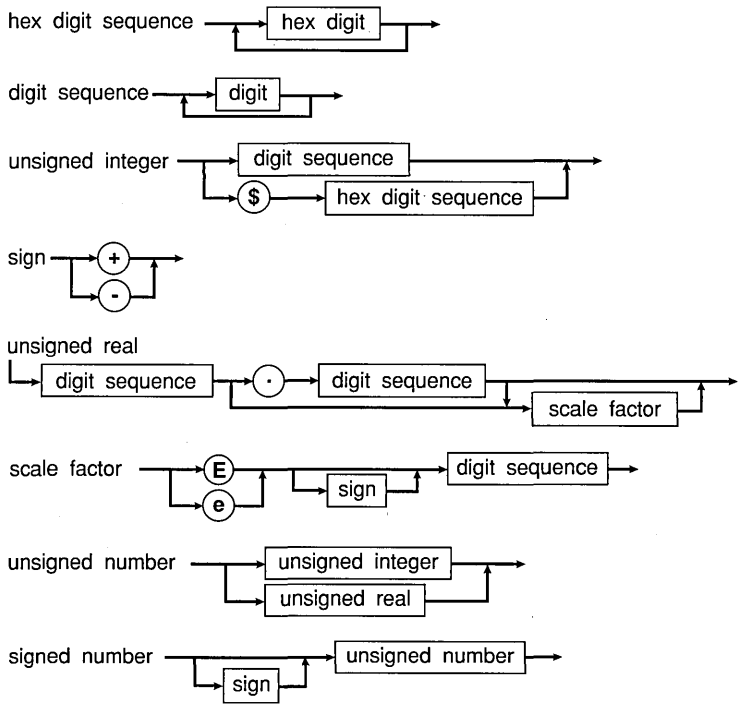
Here are some examples of identifiers and qualified identifiers:

```
Writeln
Exit
Real2String
System.MemAvail
Strings.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 type *Integer* and *Real*. 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 7×10^{-2}; 12.25e+6 or 12.25e6 both mean $12.25 \times 10^{+6}$. Syntax diagrams for writing numbers follow:

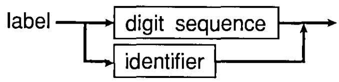


Numbers with decimals or exponents denote real-type constants. Other decimal numbers denote integer-type constants; they must be within the range –2,147,483,648 to 2,147,483,647.

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, Turbo Pascal also allows identifiers to function as labels.

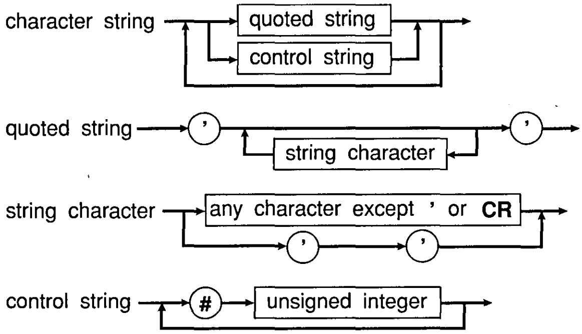
Character strings

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 within a character string denote a single character, an apostrophe. For example,

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

As an extension to Standard Pascal, Turbo 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 control characters 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 *)
```

The compiler directives are explained in Chapter 2 of the Programmer's Reference.

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

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

CHAPTER 3

Constants

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

Wherever Standard Pascal allows only a simple constant, Turbo Pascal allows a constant expression.

As an extension to Standard Pascal, Turbo 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. Examples of constant expressions follow:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Turbo' + ' ' + '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 59)
- Function calls (except those noted in the following text)
- The address operator (@) (except in constant address expressions as described on page 59)

For expression syntax, see Chapter 6, "Expressions."

Except for these restrictions, constant expressions follow the same syntactical rules as ordinary expressions.

The following standard functions are allowed in constant expressions:

Abs	*High*	*Low*	*Pred*	*SizeOf*
Chr	*Length*	*Odd*	*Ptr*	*Succ*
Hi	*Lo*	*Ord*	*Round*	*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;
```

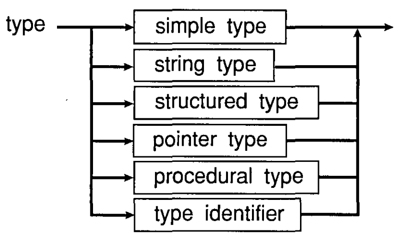
C H A P T E R

4

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.

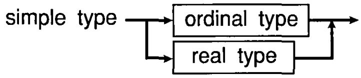
When an identifier occurs on the left side of a type declaration, it's declared as a *type identifier* for the block in which the type declaration occurs. A type identifier's scope doesn't include itself except for pointer types.

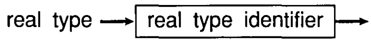


There are five major type classes. They are described in the following sections.

Simple types

Simple types define ordered sets of values.





Chapter 2 explains how to denote constant integer-type and real-type values.

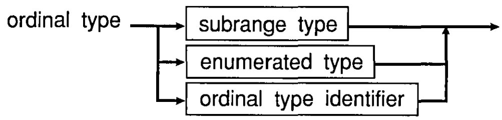
A real-type identifier is one of the standard identifiers: *Real, Single, Double, Extended,* or *Comp.*

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 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 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:



Turbo Pascal has ten predefined ordinal types: *Integer, Shortint, Longint, Byte, Word, 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

There are five predefined integer types: *Shortint, Integer, Longint, Byte,* and *Word*. Each type denotes a specific subset of the whole numbers, according to the following table:

Table 4.1
Predefined integer types

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

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 *Integer* and *Byte* is *Integer*, and the common type of *Integer* 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 *Integer* and *Word* before any arithmetic operation is performed.

Typecasting is described in Chapters 5 and 6.

An integer-type value can be explicitly converted to another integer type through typecasting.

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* exist primarily 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*.

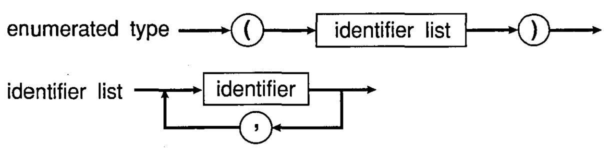
Char type

Char's set of values are characters, ordered according to the extended's 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:

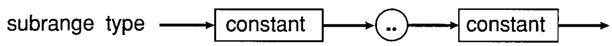
```
type
  Suit = (Club, Diamond, Heart, Spade);
```

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 4.2
Real data types

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}.

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

Turbo Pascal supports two models of code generation for performing real-type operations: *software* floating point and *80x87* floating point. Use the **$N** compiler directive to select the appropriate model. If no 80x87 is present, enable the **$E** compiler directive to provide full 80x87 emulation in software.

Software floating point

In the {**$N-**} state, which is selected by default, 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.

80x87 floating point

In the {**$N+**} state, 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 and software emulation, refer to Chapter 14, "Using the 80x87."

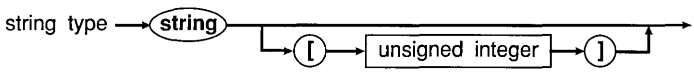
Turbo Pascal includes a run-time library that will automatically *emulate* an 80x87 in software if one isn't present. The **$E** compiler directive is used to determine whether or not the 80x87 emulator should be included in a program.

String types

Operators for the string types are described in the sections "String operator" and "Relational operators" in Chapter 6.

String-type standard procedures and functions are described in "String procedures and functions" on page 131.

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*.



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 the section "Arrays, strings, and indexes" on page 55.

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.

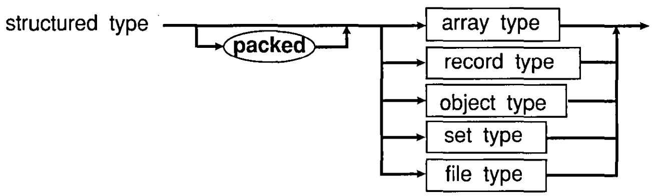
Read about open string parameters on page 111.

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.

Structured types

The maximum permitted size of any structured type in Turbo Pascal is 65,520 bytes.

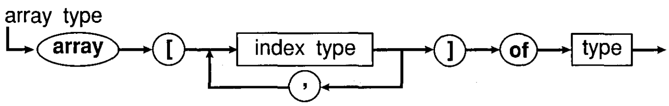
A structured type, characterized by its structuring method and by its component type(s), holds more than one value. If a component type is structured, the resulting structured type has more than one level of structuring. A structured type can have unlimited levels of structuring.

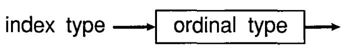


In Standard Pascal, the word **packed** in a structured type's declaration tells the compiler to compress data storage, even at the cost of diminished access to a component of a variable of this type. In Turbo Pascal, however, **packed** has no effect; instead packing occurs automatically whenever possible.

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
```

See "Arrays, strings, and indexes" on page 55.

You access an array's components by supplying the array's identifier with one or more indexes in brackets.

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
```

See "Identical and compatible types" on page 46.

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

An array type of the form

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

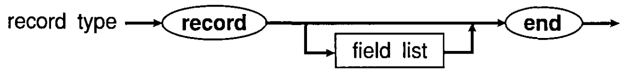
where *X* is a positive nonzero integer is called a *zero-based character array*. Zero-based character arrays are used to store *null-terminated strings*, and when the extended syntax is enabled (using a {**$X+**} compiler directive), a zero-based character array is compatible with a *PChar* value. For a complete discussion of this topic, read Chapter 16, "Using null-terminated strings," beginning on page 167.

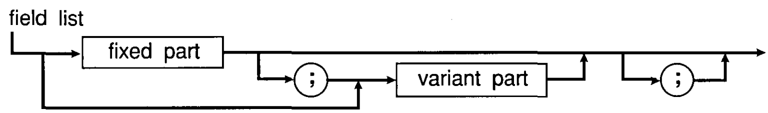
Read about open array parameters on page 113.

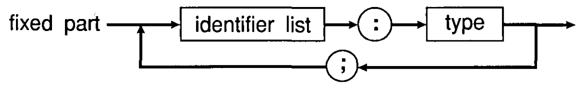
A parameter declared using the **array of** *T* syntax is an *open array parameter*. Open array parameters allow arrays of varying sizes to be passed to the same procedure or function.

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.





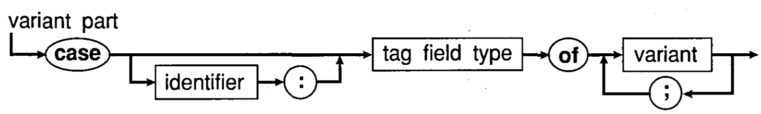


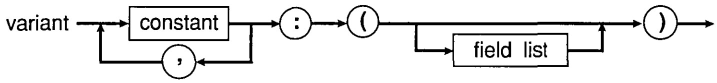
The fixed part of a record type sets out the list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

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:

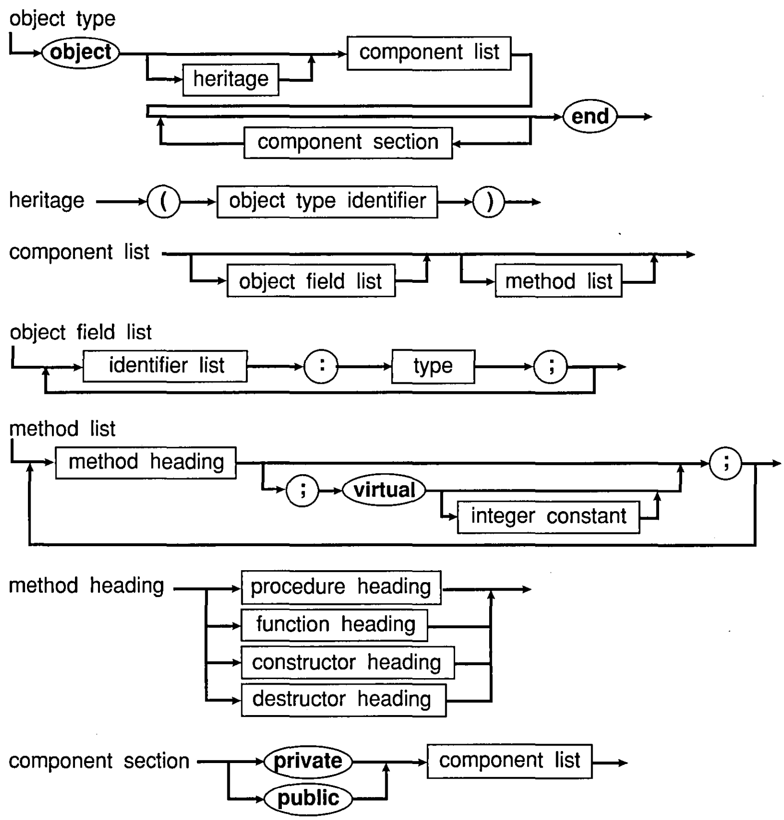
```
type
  TPerson = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Citizen: Boolean of
      True: (BirthPlace: string[40]);
      False: (Country: string[20];
        EntryPort: string[20];
        EntryDate: TDate;
        ExitDate: TDate);
  end;

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

Object types

An object type is a structure consisting of a fixed number of components. Each component is either a *field*, which contains data of a particular type, or a *method*, which performs an operation on the object. Similar to a variable declaration, the declaration of a field specifies the field's data type and an identifier that names the field. Similar to a procedure or function declaration, the declaration of a method specifies a procedure, function, constructor, or destructor heading.

An object type can *inherit* components from another object 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 an object type consists of itself and all its descendants.



The following code shows examples of object-type declarations:

These declarations are referred to by other examples throughout this chapter.

```
type
  TPoint = object
    X, Y: Integer;
  end;

  TRectangle = object
    A, B: TPoint;
    procedure Init(XA, YA, XB, YB: Integer);
    procedure Copy(var R: TRectangle);
    procedure Move(DX, DY: Integer);
```

```
    procedure Grow(DX, DY: Integer);
    procedure Intersect(var R: TRectangle);
    procedure Union(var R: TRectangle);
    function Contains(D: TPoint): Boolean;
  end;

  PString = ^String;

  PField = ^TField;

  TField = object
  private
    X, Y, Len: Integer;
    Name: String;
  public
    constructor Copy(var F: TField);
    constructor Init(FX, FY, FLen: Integer; FName: String);
    destructor Done; virtual;
    procedure Display; virtual;
    procedure Edit; virtual;
    function GetStr: String; virtual;
    function PutStr(S: String): Boolean; virtual;
  private
    procedure DisplayStr(X, Y: Integer; S: String);
  end;

  PStrField = ^TStrField;

  TStrField = object(TField)
  private
    Value: PString;
  public
    constructor Init(FX, FY, FLen: Integer; FName: String);
    destructor Done; virtual;
    function GetStr: String; virtual;
    function PutStr(S: String): Boolean; virtual;
    function Get: String;
    procedure Put(S: String);
  end;

  PNumField = ^TNumField;

  TNumField = object(TField)
  private
    Value, Min, Max: Longint;
  public
    constructor Init(FX, FY, FLen: Integer; FName: String;
      FMin, FMax: Longint);
    function GetStr: String; virtual;
    function PutStr(S: String): Boolean; virtual;
    function Get: Longint;
```

```
  procedure Put(N: Longint);
end;

PZipField = ^TZipField;

TZipField = object(TNumField)
public
  function GetStr: String; virtual;
  function PutStr(S: String): Boolean; virtual;
end;
```

Contrary to other types, an object type can be declared only in a type declaration part in the outermost scope of a program or unit. Therefore, an object type can't be declared in a variable declaration part or within a procedure, function, or method block.

The component type of a file type can't be an object type, or any structured type with an object-type component.

Components and scope

The scope of a component identifier extends over the domain of its object type. Also, the scope of a component identifier extends over procedure, function, constructor, and destructor blocks that implement methods of the object type and its descendants. For this reason, the spelling of a component identifier must be unique within an object type and all its descendants and all its methods.

Component identifiers declared in the component list that immediately follows the object-type heading and component identifiers declared in **public** component sections have no special restrictions on their scope. In contrast, the scope of component identifiers declared in **private** component sections is restricted to the module (program or unit) that contains the object-type declaration. In other words, **private** component identifiers act like normal public component identifiers within the module that contains the object-type declaration, but outside the module, any **private** component identifiers are unknown and inaccessible. By placing related object types in the same module, these object types can gain access to each other's **private** components without making the **private** components known to other modules.

Within an object-type declaration, a method heading can specify parameters of the object type being declared, even though the declaration isn't yet complete. In the previous example on page 34, the *Copy*, *Intersect*, and *Union* methods of the *TRectangle* type illustrate this.

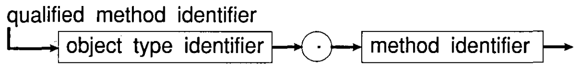
Methods

Methods can be called only through an object instance variable.

Read more about methods on page 103.

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

When unique identification of a method is required, a *qualified-method identifier* is used. It consists of an object-type identifier, followed by a period (.), followed by a method identifier. Like any other identifier, a qualified-method identifier can be prefixed with a unit identifier and a period, if required.



Virtual methods

By default, methods are *static*. With the exception of constructor methods, they can be made *virtual* by including a **virtual** directive in the method declaration. The compiler resolves calls to static methods at compile time. Calls to virtual methods are resolved at run time; this is known as *late binding*.

If an object type declares or inherits any virtual methods, then variables of that type must be *initialized* through a constructor call before any call to a virtual method. Therefore, any object type that declares or inherits any virtual methods must also declare or inherit at least one constructor method.

An object type can *override* (redefine) any of the methods it inherits from its ancestors. If a method declaration in a descendant specifies the same method identifier as a method declaration in an ancestor, then the declaration in the descendant overrides the declaration in the ancestor. The scope of an override method extends over the domain of the descendant in which it's introduced, or until the method identifier is again overridden.

An override of a static method is free to change the method heading any way it pleases. In contrast, an override of a virtual method must match exactly the order, types, and names of the parameters, and the type of the function result, if any. The override must again include a **virtual** directive.

Dynamic methods

Turbo Pascal supports an additional class of late-bound methods called *dynamic methods*. Dynamic methods differ from virtual methods only in the way dynamic method calls are dispatched at run time. For all other purposes, a dynamic method can be considered equivalent to a virtual method.

The declaration of a dynamic method is like that of a virtual method except that a dynamic method declaration must include a *dynamic method index* right after the **virtual** keyword. The dynamic method index must be an integer constant in the range 1..65535 and it must be unique among the dynamic method indexes of any other dynamic methods contained in the object type or its ancestors. For example,

```
procedure FileOpen(var Msg: TMessage); virtual 100;
```

An override of a dynamic method must match the order, types, and names of the parameters and the type of the function result of the ancestral method exactly. The override must also include a **virtual** directive followed by the same dynamic method index as was specified in the ancestor object type.

Instantiating objects

An object is *instantiated*, or created, through the declaration of a variable or typed constant of an object type, or by applying the *New* procedure to a pointer variable of an object type. The resulting object is called an *instance* of the object type. For example, given these variable declarations,

```
var
  F: TField;
  Z: TZipField;
  FP: PField;
  ZP: PZipField;
```

F is an instance of *TField* and Z is an instance of *TZipField*. Likewise, after applying *New* to *FP* and *ZP*, *FP* points to an instance of *TField* and *ZP* points to an instance of *TZipField*.

If an object type contains virtual methods, then instances of that object type must be initialized through a constructor call before any call to a virtual method. Here's an example:

```
var
  S: TStrField;
begin
  S.Init(1, 1, 25, 'Firstname');
```

```
  S.Put('Frank');
  S.Display;
    ⋮
  S.Done;
end;
```

If *S.Init* had not been called, then the call to *S.Display* causes this example to fail.

Assignment to an instance of an object type doesn't initialize the instance.

An object is initialized by compiler-generated code that executes between the time that the constructor call takes place and when execution actually reaches the first statement of the constructor's code block.

If an object instance isn't initialized and range checking is on {**$R+**}, the first call to a virtual method of the object instance results in a run-time error. If range checking is off {**$R-**}, calling a virtual method of an uninitialized object instance results in undefined behavior.

The rule of required initialization also applies to instances that are components of structured types. For example,

```
var
  Comment: array[1..5] of TStrField;
  I: Integer;
begin
  for I := 1 to 5 do Comment[I].Init(1, I + 10, 40, 'Comment');
    ⋮
  for I := 1 to 5 do Comment[I].Done;
end;
```

For dynamic instances, initialization is typically coupled with allocation, and cleanup is typically coupled with deallocation, using the extended syntax of the *New* and *Dispose* procedures. Here's an example:

```
var
  SP: PStrField;
begin
  New(SP, Init(1, 1, 25, 'Firstname'));
  SP^.Put('Frank');
  SP^.Display;
    ⋮
  Dispose(SP, Done);
end;
```

A pointer to an object type is assignment-compatible with a pointer to any ancestor object type. Therefore, during execution of a program, a pointer to an object type might point to an instance of that type or to an instance of any descendant type.

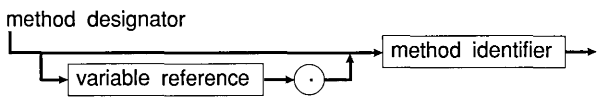
For example, a pointer of type *PZipField* can be assigned to pointers of type *PZipField*, *PNumField*, and *PField*, and during execution of a program, a pointer of type *PField* might be either **nil** or point to an instance of *TField*, *TStrField*, *TNumField*, or *TZipField*, or any other instance of a descendant of *TField*.

Pointer assignment-compatibility rules also apply to object-type variable parameters. For example, the *TField.Copy* method might be passed an instance of *TField*, *TStrField*, *TNumField*, *TZipField*, or any other instance of a descendant of *TField*.

Method activations

See "Function calls" on page 76 and "Procedure statements" on page 82.

A method is activated 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 a *method activation*.



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

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

For static methods, the *declared* (compile-time) type of the instance determines which method to activate. For example, the designators *F.Init* and *FP^.Init* will always activate *TField.Init* because the declared type of *F* and *FP^* is *TField*.

For virtual methods, the *actual* (run-time) type of the instance governs the selection. For example, the designator *FP^.Display* might activate *TField.Display*, *TStrField.Display*, *TNumField.Display*, or *TZipField.Display*, depending on the actual type of the instance pointed to by *FP*.

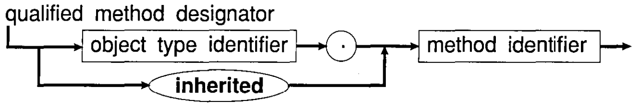
See "With statements" on page 90 and "Method declarations" on page 103.

Within a **with** statement that references an instance of an object type, 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. Likewise, within a method, the variable-reference part of a method designator can be omitted. In that case, the implicit *Self* parameter of the method activation becomes the *Self* of the method containing the call.

Qualified-method activations

See "Function calls" on page 76 and "Procedure statements" on page 82.

Within a method, a function call or procedure statement allows a *qualified-method designator* to denote activation of a specific method. This type of call is known as a *qualified-method activation*.



The object type specified in a qualified-method designator must be the same as the enclosing method's object type or an ancestor of it.

The reserved word **inherited** can be used to denote the ancestor of the enclosing method's object type; **inherited** can't be used within methods of an object type that has no ancestor.

The implicit *Self* parameter of a qualified-method activation becomes the *Self* of the method containing the call. A qualified-method activation never employs the virtual method dispatch mechanism—the call is always static and always invokes the specified method.

A qualified-method activation is generally used within an override method to activate the overridden method. Referring to the types declared earlier on page 34, here are some examples of qualified-method activations:

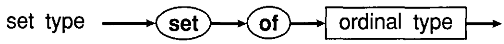
```
constructor TNumField.Init(FX, FY, FLen: Integer;
  FName: String; FMin, FMax: Longint);
begin
  inherited Init(FX, FY, FLen, FName);
  Value := 0;
  Min := FMin;
  Max := FMax;
end;

function TZipField.PutStr(S: String): Boolean;
begin
  PutStr := (Length(S) = 5) and TNumField.PutStr(S);
end;
```

As these examples demonstrate, a qualified-method activation allows an override method to "reuse" the code of the method it overrides.

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.



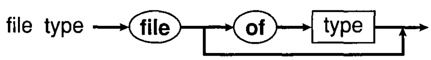
Set-type operators are described in the section "Set operators" in Chapter 6. "Set constructors" in the same chapter shows how to construct set values.

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.

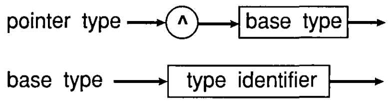


If the word **of** and the component type are omitted, the type denotes an untyped file. Untyped files are low-level I/O (input/output) channels primarily used for direct access to any disk file regardless of its internal format.

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, or the *Ptr* function. *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.

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

Type Pointer

See Chapter 5's section entitled "Pointers and dynamic variables" on page 56 for the syntax of referencing the dynamic variable pointed to by a pointer variable.

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.

Type PChar

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

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

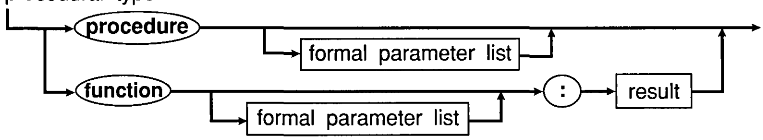
Turbo Pascal supports a set of *extended syntax* rules to facilitate handling of null-terminated strings using the *PChar* type. For a complete discussion of this topic, see Chapter 16, "Using null-terminated strings."

Procedural types

Standard Pascal regards procedures and functions as program parts that can be executed through procedure or function calls. Turbo Pascal has a much broader view of procedures and functions: It 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*.

A procedural-type declaration specifies the parameters and, for a function, the result type.

procedural type



In essence, the syntax for writing a procedural-type declaration is exactly the same as for writing a procedure or function header, except that the identifier after the **procedure** or **function** keyword is omitted. Some examples of procedural-type declarations follow:

```
type
  Proc = procedure;
  SwapProc = procedure(var X, Y: Integer);
  StrProc = procedure(S: string);
  MathFunc = function(X: Real): Real;
  DeviceFunc = function(var F: Text): Integer;
  MaxFunc = function(A, B: Real; F: MathFunc): Real;
```

The parameter names in a procedural-type declaration are purely decorative—they have no effect on the declaration's meaning.

Turbo Pascal doesn't let you declare functions that return procedural-type values; a function result must be a string, real, integer, char, boolean, pointer, or user-defined enumeration-type value. But you can return the address of a procedure or function using a function result of type *Pointer* and then typecast it to the procedural type you desire.

Procedural values

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

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

In the context of procedural values, a procedure or function declaration can be viewed as a special kind of constant declaration, the value of the constant being the procedure or function. For example, given the following declarations:

```
var
  P: SwapProc;
  F: MathFunc;

procedure Swap(var A, B: Integer); far;
var
  Temp: Integer;
begin
  Temp := A;
  A := B;
  B := Temp;
end;

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

the variables *P* and *F* can be assigned values as follows:

```
P := Swap;
F := Tan;
```

and calls can be made using *P* and *F* as follows:

```
P(I, J);        { Equivalent to Swap(I, J) }
X := F(X);      { Equivalent to X := Tan(X) }
```

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

```
if @P <> nil then P(I, J);
```

See "Procedural types in expressions" on page 44.

Notice the use of the @ operator to indicate that *P* is being examined rather than being called.

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.

To be used as procedural values, 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, methods, **inline** procedures and functions, and **interrupt** procedures 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 *MathFunc* 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 Integer;
T6 = set of Integer;
```

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

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

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

```
V1: set of Integer;
V2: set of Integer;
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.
- 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 *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.
- T_2 is assignment-compatible with an object type T_1 if T_2 is an object type in the domain of T_1.
- A pointer type P_2, pointing to an object type T_2, is assignment-compatible with a pointer type P_1, pointing to an object type T_1, if T_2 is in the domain of T_1.

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 = object
    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, *Range*, *Number*, and *Integer* are identical types. *TTestIndex* is compatible and assignment-compatible with, but not identical to, the types *Number, Range,* and *Integer*. Notice the use of constant expressions in the declarations of *TCharVal* and *TPersonBuf.*

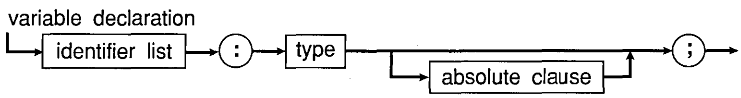
C H A P T E R

5

Variables and typed constants

Variable declarations

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



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: Real;
  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 Real;
```

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.

For information on dynamic variables, see "Pointers and dynamic variables" on page 56.

If you need more than 65,520 bytes of global data, you should allocate the larger structures as dynamic variables.

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.

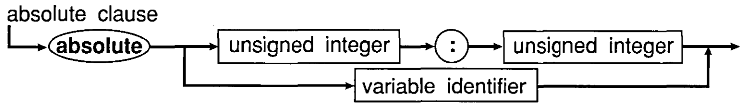
Each time a procedure or function is activated (called), it allocates a set of local variables on the stack. On exit, the local variables are disposed of. At any time during the execution of a program, the total size of the local variables allocated by the active procedures and functions can't exceed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



The variable declaration's identifier list can only specify 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).

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;
```

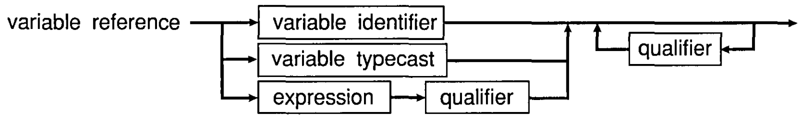
This declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and because the first byte of a string variable contains the dynamic length of the string, *StrLen* will contain the length of *Str*.

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

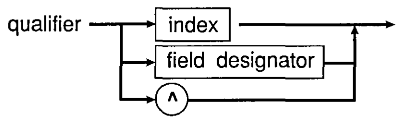
This is the syntax of a variable reference:



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^
```

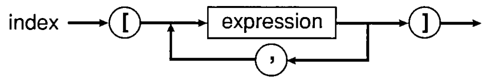
If the variable being pointed to is an array, indexes can be added to denote components of this array:

```
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]
```

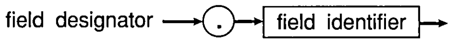
You can index a string variable with a single index expression, whose value must be in the range 0..*N*, where *N* is the declared size of the string. This accesses one character of the string value, with the type *Char* given to that character value.

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 an object type. In that case, the instance and the period can be omitted in referencing components of the object 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.

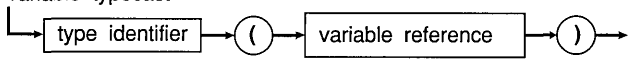
nil doesn't point to any variable. The results are undefined if you access a dynamic variable when the pointer's value is **nil** or undefined. These are examples of references to dynamic variables:

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

Variable typecasts

A variable reference of one type can be changed into a variable reference of another type through a *variable typecast*.

variable typecast



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 (the number of bytes occupied by 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 specified type.

These are examples of variable typecasts:

```
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.

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

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

you can construct the following assignments:

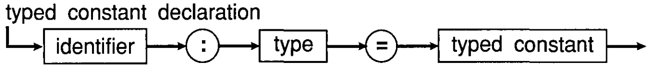
```
F := Func(P);                    { Assign procedural value in P to F }
Func(P) := F;                    { Assign procedural value in F to P }
@F := P;                            { Assign pointer value in P to F }
P := @F;                            { Assign pointer value in F to P }
N := F(N);  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

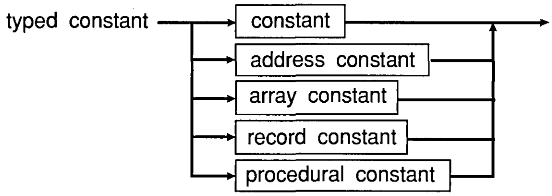
In particular, notice that the address operator (@), when applied to a procedural variable, can be used on the left side of an assignment. Also, notice the typecast on the last line to call a function via a pointer variable.

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 constant declaration





Typed constants can be used exactly like variables of the same type, and 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
  TVector = array[Min..Max] of Integer;
```

The *TVector* 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:

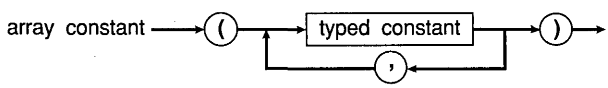
```
const
  Heading: string[7] = 'Section';
  NewLine: string[2] = #13#10;
  TrueStr: string[5] = 'Yes';
  FalseStr: string[5] = 'No';
```

Structured-type constants

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

Array-type constants

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



This is an example of an array-type constant:

```
type
  TStatus = (Active, Passive, Waiting);
  TStatusMap = array[Status] 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';
```

For more about null-terminated strings, see Chapter 16.

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

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

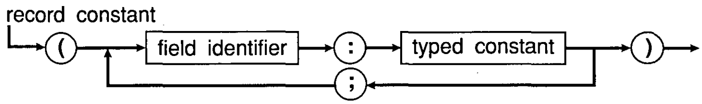
provides an initialized array *Maze* with the following values:

```
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.



These are examples of record constants:

```
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.

Object-type constants

The declaration of an object-type constant uses the same syntax as the declaration of a record-type constant. No value is, or can be, specified for method components. Referring to the earlier object-type declarations starting on page 34, these are examples of object-type constants:

```
const
  ZeroPoint: TPoint = (X: 0; Y: 0);
  ScreenRect: TRect = (A: (X: 0; Y: 0); B: (X: 80; Y: 25));
  CountField: TNumField = (X: 5; Y: 20; Len: 4; Name: nil;
    Value: 0; Min: -999; Max: 999);
```

Constants of an object type that contains virtual methods need *not* be initialized through a constructor call—this initialization is handled automatically by the compiler.

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
  Digits = set of 0..9;
  Letters = set of 'A'..'Z';
const
  EvenDigits: Digits = [0, 2, 4, 6, 8];
  Vowels: Letters = ['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;
  PNode = ^TNode;
  TNode = record
    Next: PNode;
    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: PNode = @N4;
```

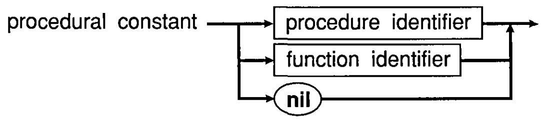
When the extended syntax is enabled (using a {**$X+**} compiler directive), a typed constant of type *PChar* can be initialized with a string constant. For example,

```
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 Chapter 16, "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;
```

C H A P T E R

6

Expressions

Expressions are made up of *operators* and *operands*. Most 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 6.1
Precedence of operators

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

There are three basic rules of precedence:

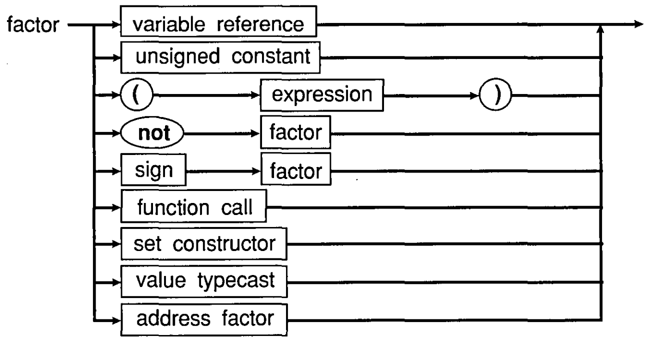
- An operand between two operators of different precedence is bound to the operator with higher precedence.
- An operand between two equal operators is bound to the one on its left.
- Expressions within parentheses are evaluated prior to being treated as a single operand.

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:



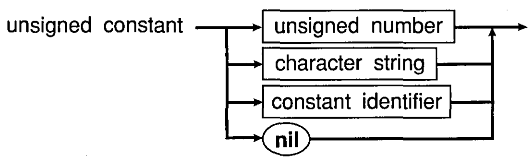
A function call activates a function and denotes the value returned by the function. See "Function calls" on page 76.

A set constructor denotes a value of a set type. See "Set constructors" on page 76.

A value typecast changes the type of a value. See "Value typecasts" on page 77.

An address factor computes the address of a variable, procedure, function, or method. See "The @ operator" on page 75.

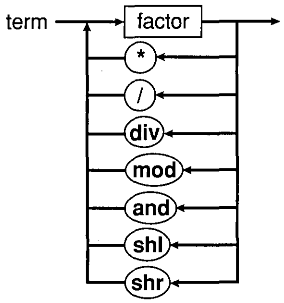
An unsigned constant has the following syntax:



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 }
```

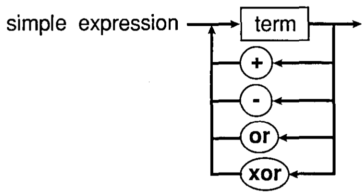
Terms apply the multiplying operators to factors:



Here are some examples of terms:

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

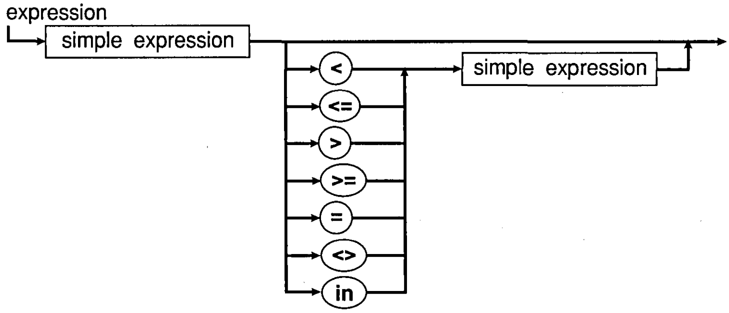
Simple expressions apply adding operators and signs to terms:



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 Huel
```

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 6.2
Binary arithmetic operations

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

Operator	Operation	Operand types	Result type
+	addition	integer type real type	integer type real type
–	subtraction	integer type real type	integer type real type
*	multiplication	integer type real type	integer 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

Table 6.1
Unary arithmetic operations

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

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

For a definition of common types, see page 25.

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.

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 Table 6.4.

Table 6.4
Logical operations

*The **not** operator is a unary operator.*

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	shift left	integer type	*Boolean*
shr	shift right	integer type	*Boolean*

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

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 *I* **shr** *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 Table 6.5.

Table 6.5
Boolean operations

*The **not** operator is a unary operator.*

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.*

Turbo 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 is not 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 are not 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 Table 6.6.

Table 6.6
String operation

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

Turbo 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 6.7
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 Table 6.8.

Table 6.8
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 Table 6.9.

Table 6.9 Relational operations

Operator type	Operation	Operand types	Result type
=	equal	compatible simple, pointer, set, string, or packed string types	*Boolean*
<>	not equal	compatible simple, 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

The operators **=** and **<>** can be used on compatible pointer-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:

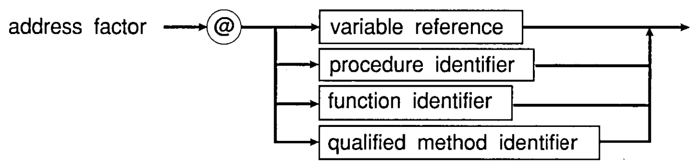
- *A* **=** *B* is *True* only if *A* and *B* contain exactly the same members; otherwise, *A* **<>** *B*.
- *A* **<=** *B* is *True* only if every member of *A* is also a member of *B*.
- *A* **>=** *B* is *True* only if every member of *B* is also a member of *A*.

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*.

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

Special rules apply to use of the @ operator with a procedural variable. For more details, see "Procedural types in expressions" on page 78.

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.

@ with a procedure, function, or method

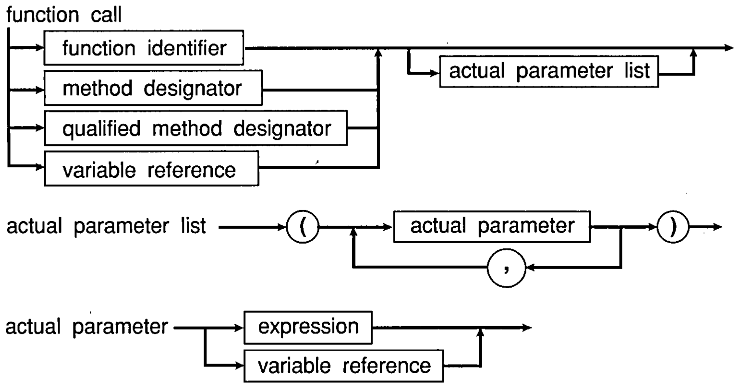
You can apply @ to a procedure, function, or method to produce a pointer to the routine's entry point. The type of the resulting pointer is always *Pointer*, regardless of the state of the **$T** compiler directive. In other words, the result is always an untyped pointer, which is compatible with all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

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

Function calls

See "Method activations" on page 40, "Qualified-method activations" on page 41, and "Procedural types" on page 44.

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 9, "Procedures and functions," on page 107.



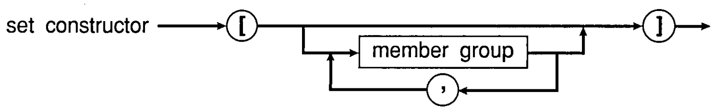
Some examples of function calls follow:

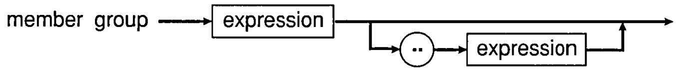
```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
```

In the extended syntax {**$X+**} mode, function calls can be used as statements; that is, the result of a function call can be discarded.

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.

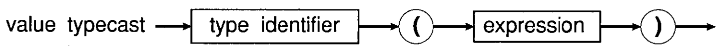
All expression values in member groups in a particular set constructor must be of the same ordinal type.

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.

See "Variable typecasts" on page 57.

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.

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
  WriteLn('Equal');
```

Standard 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
  Writeln('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. Here is an example:

```
procedure SomeProc;
begin
   :
end;

var
  SomePtr: Pointer;
begin
   :
  SomePtr := @SomeProc;
     :
end.
```

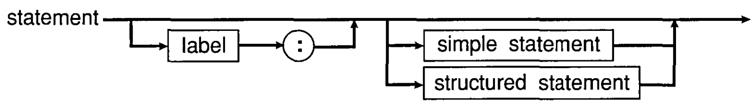
☞ To get the memory address of a procedural variable rather than the address stored in it, use a double address (@@) operator. For example, where *@P* means convert *P* into an untyped pointer variable, *@@P* means return the physical address of the variable *P*.

C H A P T E R

7

Statements

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

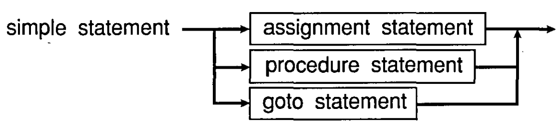


A label is either a digit sequence in the range 0 to 9999 or an identifier.

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

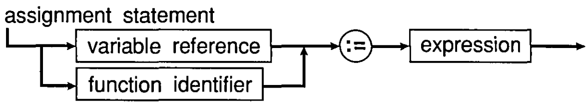
Simple statements

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.



See the section "Type compatibility" on page 47.

The expression must be assignment-compatible with the type of the variable or the type of the function result.

These are examples of assignment statements:

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

Object-type assignments

The rules of object-type assignment compatibility allow an instance of an object type to be assigned an instance of any of its descendant types. Such an assignment constitutes a *projection* of the descendant onto the space spanned by its ancestor. In the example code starting on page 34, given an instance *F* of type *TField* and an instance *Z* of type *TZipField*, the assignment *F* **:=** *Z* copies only the fields *X*, *Y*, *Len*, and *Name*.

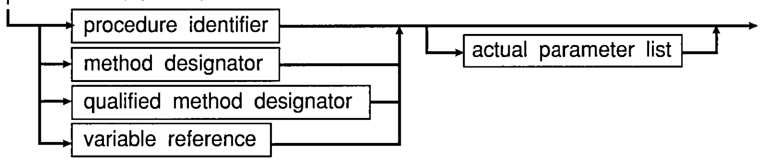
Assigning as instance of an object doesn't initialize the instance. Referring to the preceding example, the assignment *F* **:=** *Z* doesn't mean that a constructor call for *F* can be omitted.

Procedure statements

See Chapter 9, "Procedures and functions."

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 actual parameters (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.

procedure statement



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:

goto statement

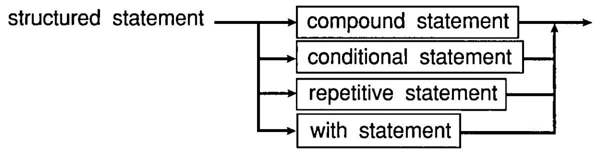
When using **goto** statements, observe the following rules:

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

- 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.

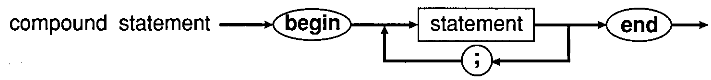
Structured statements

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 Pascal syntax only allows one statement. **begin** and **end** bracket the statements, which are separated by semicolons.

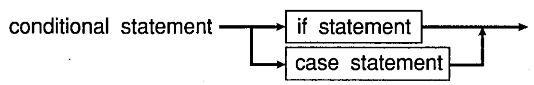


Here's an example of a compound statement:

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

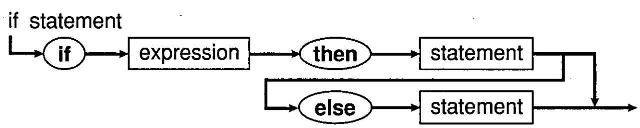
Conditional statements

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;
```

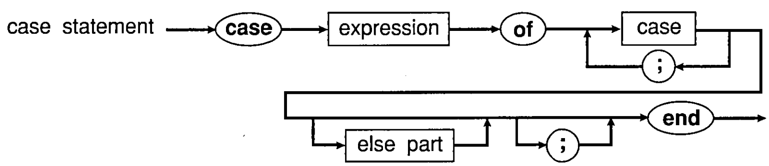
Usually, an **else** is associated with the closest **if** not already associated with an **else**.

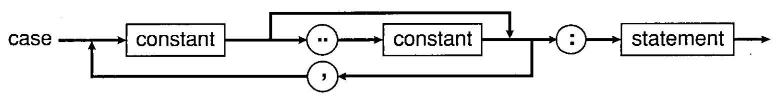
Two examples of **if** statements follow:

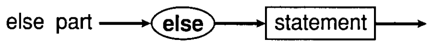
```
if X < 1.5 then
  Z := X + Y
else
  Z := 1.5;

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.







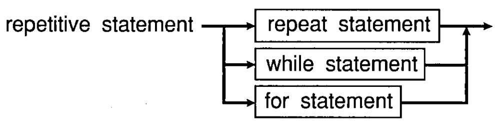
The **case** statement executes the statement prefixed by a **case** constant equal to the value of the selector or a **case** range containing the value of the selector. If no such **case** constant of the **case** range exists and an **else** part is present, the statement following **else** is executed. 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: Writeln('Even digit');
  1, 3, 5, 7, 9: Writeln('Odd digit');
  10..100: Writeln('Between 10 and 100');
else
  Writeln('Negative or greater than 100');
end;
```

Repetitive statements

Repetitive statements specify certain statements to be executed repeatedly.

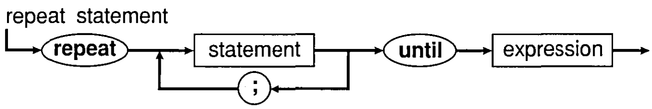


If the number of repetitions is known beforehand, the **for** statement is the appropriate construct. Otherwise, the **while** or **repeat** statement should be used.

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 Chapter 1, "Library reference," in the *Programmer's 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.

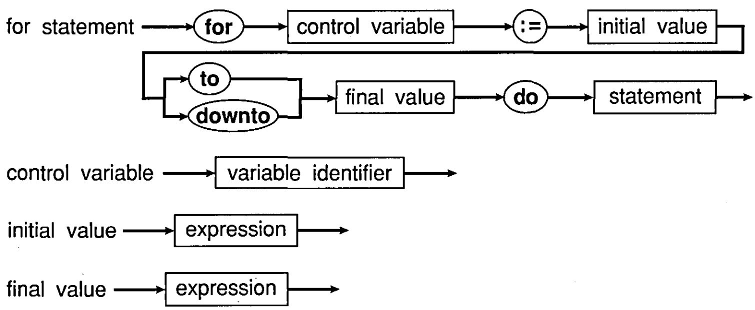
These are examples of **while** statements:

```
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.



See Chapter 8 for a discussion of locality and scope.

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.

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;
```

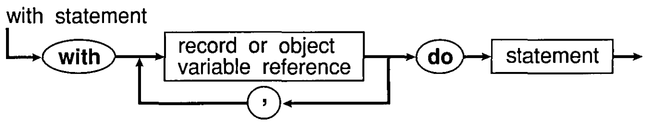
where *Temp1* and *Temp2* are auxiliary variables of the host type of the variable *V* and don't occur elsewhere in the program.

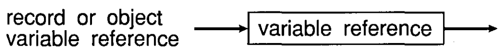
These are examples of **for** statements:

```
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.

C H A P T E R

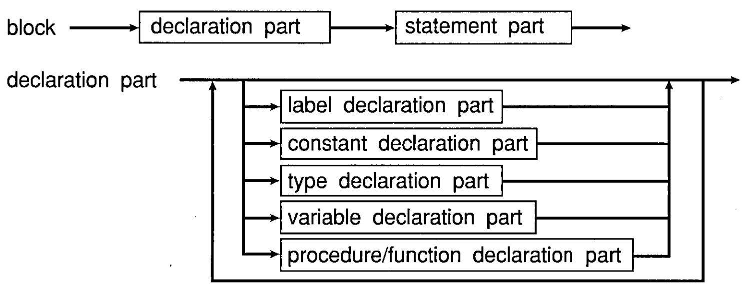
8

Blocks, locality, and scope

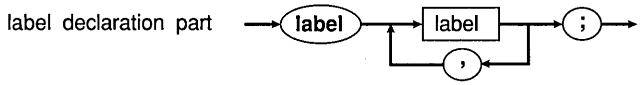
A *block* is made up of declarations, which are written and combined in any order, and statements. Each block is part of a procedure declaration, a function declaration, a method declaration, or a program or unit. All identifiers and labels declared in the declaration part are local to the block.

Blocks

The overall syntax of any block follows this format:

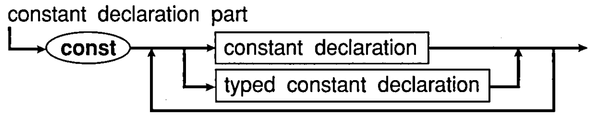


Labels that mark statements in the corresponding statement part are declared in the *label declaration part*. Each label must mark only one statement.

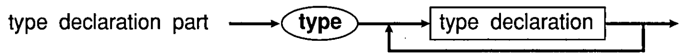


A label must be an identifier or a digit sequence in the range 0 to 9999.

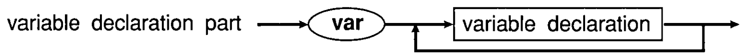
The *constant declaration part* consists of constant declarations local to the block.



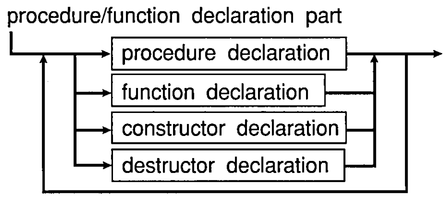
The *type declaration part* includes all type declarations local to the block.



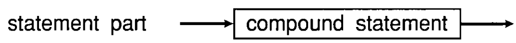
The *variable declaration part* is composed of variable declarations local to the block.



The *procedure and function declaration part* is made up of procedure and function declarations local to the block.



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

See "Record types" on page 32.

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.

Object scope

See "Object types" on page 33.

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

Unit scope

The scope of identifiers declared in the interface section of a unit follow 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. But by writing 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 Turbo 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

9

Procedures and functions

See Chapter 8, "Blocks, locality, and scope," for a definition of a block.

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. 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.

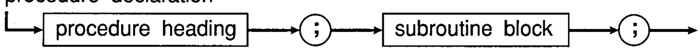
Standard procedures and functions are those that are defined in Turbo Pascal's System unit.

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

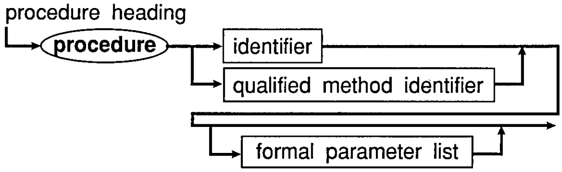
Procedure declarations

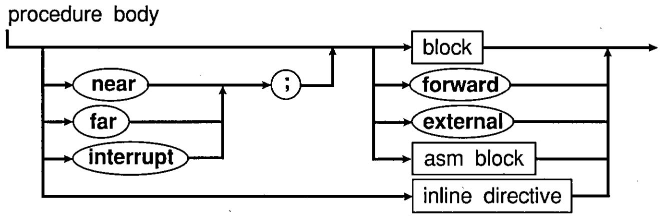
A *procedure declaration* associates an identifier with a block as a procedure; that procedure can then be activated by a procedure statement.

procedure declaration



procedure heading





The syntax for a formal parameter list is shown in the section "Parameters" on page 107.

The procedure heading names the procedure's identifier and specifies the formal parameters (if any).

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

Near and far calls are described in Chapter 20, "Control issues."

Turbo 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.

Interrupt declarations

Optionally, a procedure declaration can specify an **interrupt** directive before the block; the procedure is then considered an *interrupt procedure*. For now, note that **interrupt** procedures can't be called from procedure statements, and that every **interrupt** procedure must specify a parameter list like the following:

See "Writing interrupt procedures" on page 243.

```
procedure MyInt(Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES,
  BP: Word);
interrupt;
```

The parameter list doesn't have to match this syntax perfectly; it can be shorter and use different names, but the register contents are passed in the order listed above.

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**, **interrupt**, or **inline** declaration or another **forward** declaration.

External declarations

For further details on linking with assembly language, see Chapter 23.

External declarations let you interface with separately compiled procedures and functions written in assembly language. The external code must be linked with the Pascal program or unit through {**$L** *filename*} directives.

Examples of **external** procedure declarations follow:

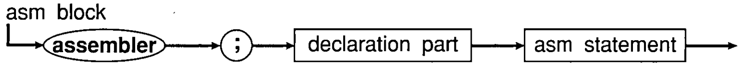
```
procedure MoveWord(var Source, Dest; Count: Word); external;
procedure MoveLong(var Source, Dest; Count: Word); external;

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

{$L BLOCK.OBJ}
```

Assembler declarations

For more details on assembler procedures and functions, see Chapter 22.

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

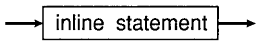
asm block



Inline declarations

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

inline directive



See the syntax of an inline statement on page 284.

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.

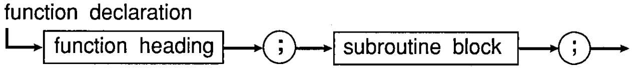
Here's a short example of two **inline** procedures:

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

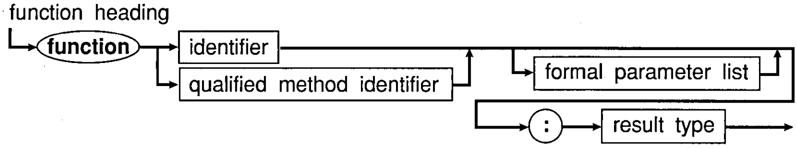
Function declarations

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

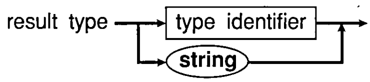
function declaration



function heading



A function can't return a structured type or a procedural type.



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.

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
  Z: Extended;
  I: Integer;
begin
  Z := 1.0; I := Y;
  while I > 0 do
  begin
    if Odd(I) then Z := Z * X;
    I := I div 2;
    X := Sqr(X);
  end;
```

```
    Power := Z;
  end;
```

Like procedures, functions can be declared as **near**, **far**, **forward**, **external**, **assembler**, or **inline**; but **interrupt** functions aren't allowed.

Method declarations

See page 37 for more about declaring methods in objects.

The declaration of a method within an object type corresponds to a **forward** declaration of that method. Therefore, somewhere after the object-type declaration and within the same scope as the object-type declaration, the method must be implemented by a defining declaration.

For procedure and function methods, the defining declaration takes the form of a normal procedure or function, but the procedure or function identifier is a *qualified-method identifier*. This is an object-type identifier followed by a period (.) and then by a method identifier.

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

Optionally, a method's defining declaration can repeat the formal parameter list of the method heading in the object type. If it does, 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 variable parameter that possesses the object type. In the method block, *Self* represents the instance whose method component was designated to activate the method. Therefore, any changes made to the values of the fields of *Self* are reflected in the instance.

The scope of a component identifier in an object type extends over any procedure, function, constructor, or destructor block that implements a method of the object 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
```

For this reason, the spellings of component identifiers, formal method parameters, *Self*, and any identifiers introduced in a method implementation must be unique.

Here are some examples of method implementations:

See the object-type declarations of these examples on page 34.

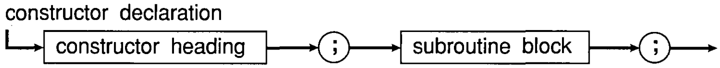
```
procedure TRectangle.Intersect(var R: TRectangle);
begin
  if A.X < R.A.X then A.X := R.A.X;
  if A.Y < R.A.Y then A.Y := R.A.Y;
  if B.X > R.B.X then B.X := R.B.X;
  if B.Y > R.B.Y then B.Y := R.B.Y;
  if (A.X >= B.X) or (A.Y >= B.Y) then Init(0, 0, 0, 0);
end;

procedure TField.Display;
begin
  GotoXY(X, Y);
  Write(Name^, ' ', GetStr);
end;

function TNumField.PutStr(S: String): Boolean;
var
  E: Integer;
begin
  Val(S, Value, E);
  PutStr := (E = 0) and (Value >= Min) and (Value <= Max);
end;
```

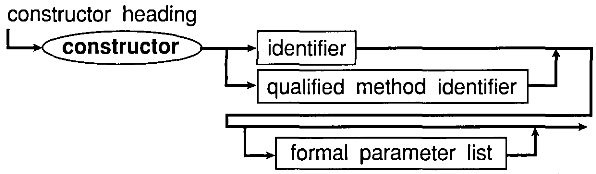
Constructors and destructors

Constructors and *destructors* are specialized forms of methods. Used in connection with the extended syntax of the *New* and *Dispose* standard procedures, constructors and destructors have the ability to allocate and deallocate dynamic objects. In addition, constructors have the ability to perform the required initialization of objects that contain virtual methods. Like other methods, constructors and destructors can be inherited, and an object can have any number of constructors and destructors.

Constructors are used to initialize newly created objects. Usually, the initialization is based on values passed as parameters to the constructor. Constructors can't be virtual because the virtual-method dispatch-mechanism depends on a constructor first having initialized the object.

constructor declaration





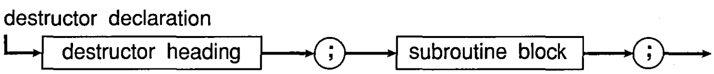
Here are some examples of constructors:

```
constructor TField.Copy(var F: TField);
begin
  Self := F;
end;

constructor TField.Init(FX, FY, FLen: Integer; FName: String);
begin
  X := FX;
  Y := FY;
  Len := FLen;
  GetMem(Name, Length(FName) + 1);
  Name^ := FName;
end;

constructor TStrField.Init(FX, FY, FLen: Integer; FName: String);
begin
  inherited Init(FX, FY, FLen, FName);
  GetMem(Value, Len);
  Value^ := '';
end;
```

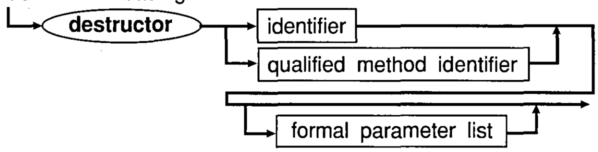
The first action of a constructor of a descendant type, such as the preceding *TStrField.Init*, is almost always to call its immediate ancestor's corresponding constructor to initialize the inherited fields of the object. Having done that, the constructor then initializes the fields of the object that were introduced in the descendant.

Destructors can be virtual, and often are. Destructors seldom take any parameters.

Destructors are the counterparts of constructors, and are used to clean up objects after their use. Typically, the cleanup consists of disposing of any pointer fields that were allocated by the object.



destructor heading



Here are some examples of destructors:

```
destructor TField.Done;
begin
  FreeMem(Name, Length(Name^) + 1);
end;

destructor TStrField.Done;
begin
  FreeMem(Value, Len);
  inherited Done;
end;
```

A destructor of a descendant type, such as the preceding *TStrField.Done*, usually disposes of the pointer fields introduced in the descendant, and then, as its last action, calls the corresponding destructor of its immediate ancestor to dispose of any inherited pointer fields of the object.

Constructor-error recovery

Turbo Pascal allows you to install a heap-error function through the *HeapError* variable in the *System* unit; see Chapter 19. This functionality affects the way object-type constructors work.

By default, when there isn't enough memory to allocate a dynamic instance of an object type, a constructor call using the extended syntax of the *New* standard procedure generates runtime error 203. If you install a heap-error function that returns 1 rather than the standard function result of 0, a constructor call through *New* will return **nil** when it can't complete the request (instead of aborting the program).

The code that performs allocation and virtual method table (VMT) field initialization of a dynamic instance is part of a constructor's entry sequence: When control arrives at the **begin** of the constructor's statement part, the instance will have been allocated and initialized already. If allocation fails and the heap-error function returns 1, the constructor skips execution of the statement part and returns a **nil** pointer. The pointer specified in the *New* construct that called the constructor is set to **nil**.

Once control arrives at the **begin** of a constructor's statement part, the object-type instance is guaranteed to have been allocated and initialized successfully. The constructor itself, however, might attempt to allocate dynamic variables to initialize pointer fields in the instance and, in turn, these allocations might fail. If that happens, a well-behaved constructor should reverse any successful allocations and deallocate the object-type instance so that the net result becomes a **nil** pointer. To make such backing out possible, Turbo Pascal implements the *Fail* standard procedure that takes no parameters and can be called only from within a constructor. A call to *Fail* causes a constructor to deallocate the dynamic instance that was allocated upon entry to the constructor and causes the return of a **nil** pointer to indicate its failure.

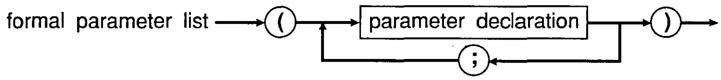
When dynamic instances are allocated through the extended syntax of *New*, a resulting value of **nil** in the specified pointer variable indicates that the operation failed. Unfortunately, there is no such pointer variable to inspect after the construction of a static instance or when an inherited constructor is called. Instead, Turbo Pascal allows a constructor to be used as a Boolean function in an expression: A return value of *True* indicates success, and a return value of *False* indicates failure due to a call to *Fail* within the constructor.

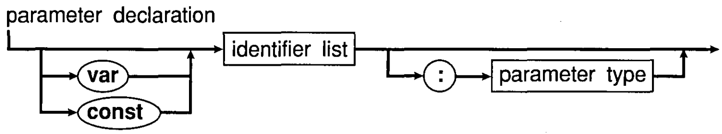
On disk you'll find two programs, NORECVER.PAS and RECOVER.PAS. Both implement two simple object types that contain pointers. The NORECVER version of the program does not implement constructor-error recovery.

RECOVER.PAS demonstrates how the program can be rewritten to implement error recovery. Notice how the corresponding destructors in *Base.Init* and *Derived.Init* are used to reverse any successful allocations before *Fail* is called to finally fail the operation. Also notice that in *Derived.Init*, the call to *Base.Init* is coded within an expression so that the success of the inherited constructor can be tested.

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. Your program can refer to it by its identifier in the block associated with the procedure or function.





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.

Open parameters are described on page 111.

String and array type parameters can be *open parameters.* 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 aren't allowed. Similarly, 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.

File types can be passed only as variable parameters.

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).

For more information on open-string parameters, see page 111.

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

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.

The rules of object-type assignment compatibility also apply to object-type variable parameters: For a formal parameter of type *T1*, the actual parameter might be of type *T2* if *T2* is in the domain of *T1*. For example, given the object-type declarations found on page 34, the *TField.Copy* method might be passed an instance of *TField*, *TStrField*, *TNumField*, *TZipField*, or any other instance of a descendant of *TField*.

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

The *OpenString* identifier is declared in the *System* unit. It denotes a special string type that can only be used in the declaration of string parameters. For reasons of backward compatibility, *OpenString* isn't a reserved word—this means that *OpenString* can be redeclared as a user-defined identifier.

When backward compatibility isn't an issue, a {**$P+**} compiler directive can be used to change the meaning of the **string** keyword. In the {**$P+**} state, a variable declared using the **string** keyword is an open-string parameter.

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;
```

☞ Value and constant parameters declared using the *OpenString* identifier or the **string** keyword in the **{$P+}** state aren't 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

When open parameters are enabled (using a {**$P+**} compiler directive), 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's 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 aren't allowed, and assignments to elements of a formal open-array value parameter don't affect the actual parameter.

☞ 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 *Real*, and the *Sum* function computes the sum of all elements in an array of *Real*. Because the *A* parameter in both cases is an open-array parameter, the subroutines can operate on any array with an element type of *Real*.

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

function Sum(const A: array of Real): Real;
var
  I: Word;
  S: Real;
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)*.

Dynamic object-type variables

The *New* and *Dispose* standard procedures allow a constructor call or destructor call as a second parameter for allocating or disposing of a dynamic object-type variable. This is the syntax:

```
New(P, Construct)
```

and

```
Dispose(P, Destruct)
```

where *P* is a pointer variable, pointing to an object type, and *Construct* and *Destruct* are calls to constructors and destructors of that object type. For *New,* the effect of the extended syntax is the same as executing

```
New(P);
P^.Construct;
```

And for *Dispose,* the effect of the extended syntax is the same as executing

```
P^.Destruct;
Dispose(P);
```

Without the extended syntax, you would frequently have to call *New* followed by a constructor call or call a destructor followed by a call to *Dispose.* The extended syntax improves readability and generates shorter and more efficient code.

The following illustrates the use of the extended *New* and *Dispose* syntax:

```
var
  SP: PStrField;
  ZP: PZipField;
begin
  New(SP, Init(1, 1, 25, 'Firstname'));
  New(ZP, Init(1, 2, 5, 'Zip code', 0, 99999));
  SP^.Edit;
  ZP^.Edit;
     ⋮
  Dispose(ZP, Done);
  Dispose(SP, Done);
end;
```

You can also use *New* as a *function* that allocates and returns a dynamic variable of a specified type:

```
New(T)
```

or

```
New(T, Construct)
```

In the first form, *T* can be any pointer type. In the second form, *T* must point to an object type and *Construct* must be a call to a constructor of that object type. In both cases, the type of the function result is *T*.

Here's an example:

```
var
  F1, F2: PField;
begin
  F1 := New(PStrField, Init(1, 1, 25, 'Firstname'));
  F2 := New(PZipField, Init(1, 2, 5, 'Zip code', 0, 99999));
    :
  WriteLn(F1^.GetStr);                    { Calls TStrField.GetStr }
  WriteLn(F2^.GetStr);                    { Calls TZipField.GetStr }
    :
  Dispose(F2, Done);                         { Calls TField.Done }
  Dispose(F1, Done);                       { Calls TStrField.Done }
end;
```

Notice that even though *F1* and *F2* are of type *PField*, the extended-pointer assignment-compatibility rules allow *F1* and *F2* to be assigned a pointer to any descendant of *TField*. Because *GetStr* and *Done* are virtual methods, the virtual-method dispatch-mechanism correctly calls *TStrField.GetStr*, *TZipField.GetStr*, *TField.Done*, and *TStrField.Done*, respectively.

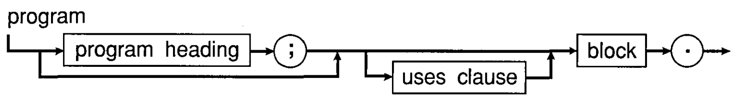
C H A P T E R

10

Programs and units

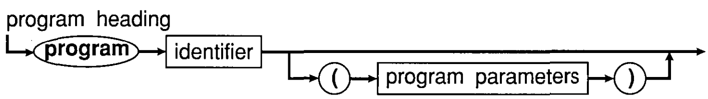
Program syntax

A Turbo Pascal program consists of a program heading, an optional **uses** clause, and a block.



The program heading

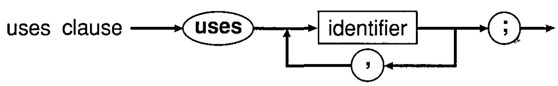
The program heading specifies the program's name and its parameters.



If a program heading is present, the compiler ignores it.

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 procedures and functions to support such features as file input and output (I/O), string handling, floating point, dynamic memory allocation, and others.

Apart from *System*, Turbo Pascal implements many standard units, such as *Dos* and *Crt*. These aren't used automatically; you must include them in your **uses** clause. For example,

```
uses Dos, Crt;                          { Can now use Dos and Crt }
```

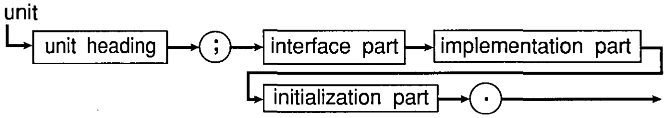
The order of the units listed in the **uses** clause determines the order of their initialization (see "The initialization part" on page 120).

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 .TPU. For example, a unit named *MathFunctions* will be stored in a file called MATHFUNC.TPU. Even though the file name is truncated, a **uses** clause must still specify the full unit identifier.

Unit syntax

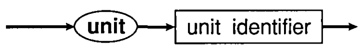
Units are the basis of modular programming in Turbo Pascal. They're used to create libraries that you can include in various programs without making the source code available, and to divide large programs into logically related modules.

unit



The unit heading

The unit heading specifies the unit's name.

unit heading 

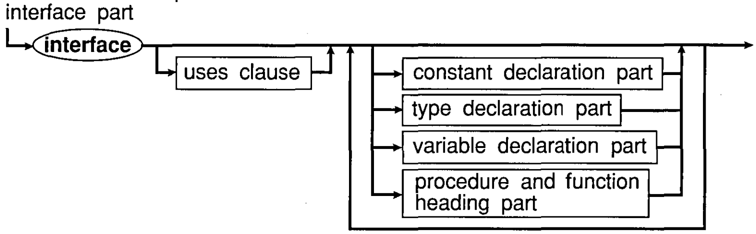
The unit name is used when referring to the unit in a **uses** clause. The name must be unique: Two units with the same name can't be used at the same time.

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.

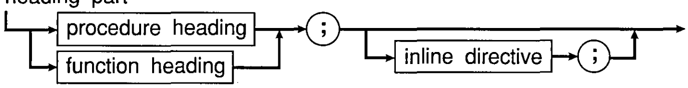
The interface part

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



procedure and function heading part

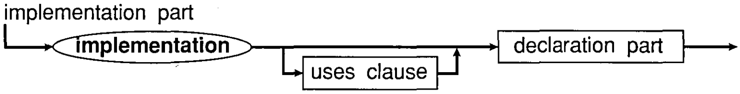


Unless a procedure or function is **inline**, the interface part only lists the procedure or function heading. The block of the procedure or function follows in the implementation part.

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, they aren't available to the host.

implementation part



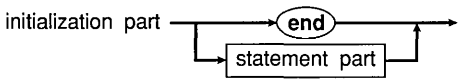
The procedure and function declarations in the interface part are similar to forward declarations, although the **forward** directive isn't specified. Therefore, these procedures and functions can be defined and referenced in any sequence in the implementation part.

☞ Procedure and function headings can be duplicated from the interface part. You don't have to specify the formal parameter list.

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 a statement part to be executed to initialize the unit.



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.
```

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, either directly or indirectly. So, to compile *Prog*, the compiler must be able to locate both *Unit1* and *Unit2*, or an error occurs.

Note for C and other language users: The uses clauses of a Turbo Pascal program provide the "make" logic information traditionally found in make or project files of other languages. With the uses clause, Turbo 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, other units that use the unit *need not* be recompiled. In the previous example, if the interface part of *Unit1* is changed (for example, $c = 2$) *Unit2* must be recompiled, but changing the implementation part (for example, $d = 1$) doesn't require recompiling *Unit2*.

Turbo 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

If you place a **uses** clause in the implementation section of a unit, you hide the inner details of the unit referenced in the **uses** clause; the referenced unit is private and not available to the program or unit using the unit it's referenced in. You can use this technique to construct mutually-dependent units.

The following program demonstrates how two units can "use" each other:

```
program Circular;
{ Display text using WriteXY }

uses
  Crt, Display;

begin
  ClrScr;
  WriteXY(1, 1, 'Upper left corner of screen');
  WriteXY(1000, 1000, 'Way off the screen');
  WriteXY(81 - Length('Back to reality'), 15, 'Back to reality');
end.
```

The main program, *Circular*, uses a unit named *Display*:

```
unit Display;
{ Contains a simple video display routine }

interface

procedure WriteXY(X, Y: Integer; Message: String);

implementation
uses
  Crt, Error;

procedure WriteXY(X, Y: Integer; Message: String);
begin
  if (X in [1..80]) and (Y in [1..25] then
  begin
    GoToXY(X, Y);
    Write(Message);
  end

  else
    ShowError('Invalid WriteXY coordinates');
end;

end.
```

The *Display* unit declares the *WriteXY* procedure in its interface section. The *WriteXY* procedure writes a message on the screen. The program *Circular* specifies the content and screen position of the message in the parameters passed to *WriteXY*. If the screen coordinates aren't onscreen, *WriteXY* calls the *ShowError* procedure.

ShowError isn't in the *Display* unit, but in another unit, *Error*, referenced in the **uses** section of the *Display* unit's implementation section. This is the *Error* unit:

```
unit Error;
{ Contains a simple error-reporting routine }

interface

procedure ShowError(ErrMsg: String)

implementation

uses
  Display;
```

```
procedure ShowError(ErrMsg: String);
begin
  WriteXY(1, 25, 'Error: ' + ErrMsg);
end;

end.
```

The *Error* unit is somewhat unusual: its one declared procedure, *ShowError*, uses the *WriteXY* procedure declared in the *Display* unit, the unit that calls the *ShowError* procedure. The **uses** clause in the implementation sections of both the *Display* and *Error* units refer to each other. This is possible because Turbo Pascal can compile complete interface sections for both. The compiler accepts a reference to a partially-compiled unit in the implementation section of another unit, as long as neither unit's interface section depends upon the other. Therefore, the units follow Pascal's strict rules for declaration order.

☞ If the interface sections are interdependent, you get a circular unit-reference error.

Sharing other declarations

If you want to modify the *WriteXY* and *ShowError* procedures to take an additional parameter that specifies a rectangular window onscreen, you might write this:

```
procedure WriteXY(SomeWindow: WindRec; X, Y: Integer;
  Message: String);

procedure ShowError(SomeWindow: WindRec; ErrMsg: String);
```

These procedures are declared in the interface sections of different units. Because both need to use the *WindRec* type, *WindRec* can't be declared in either of the interface sections—that would make them depend on each other. The solution is to create a third unit that contains only the definition of the window record:

```
unit WindData;
interface
type
  WindRec = record
    X1, Y1, X2, Y2: Integer;
    ForeColor, BackColor: Byte;
    Active: Boolean;
  end;
implementation
end.
```

You can now add *WindData* to the **uses** clause in interface sections of both the *Display* and *Error* units. Both of these units can use the new record type, but *Display* and *Error* still refer to each other only in their respective implementation sections.

☞ Mutually-dependent units can be useful in special situations, but use them judiciously. If you use them when the aren't needed, they can make your program harder to maintain and more susceptible to errors.

P A R T 2

The run-time library

C H A P T E R

11

Overview of the run-time library

Turbo Pascal's run-time library is made up of all the standard units found in the TURBO.TPL file: *System*, *Dos*, *Overlay*, *Printer*, and *Crt*. This chapter briefly describes each of these units. They are loaded into memory with Turbo Pascal and are readily available to your programs.

Five other units that aren't in TURBO.TPL but that come with Turbo Pascal are *Strings*, *WinDos*, *Graph*, *Turbo3*, and *Graph3*. They are described briefly here also.

System unit

The *System* unit implements low-level, run-time support routines for all built-in features, such as file I/O, string handling, floating point, and dynamic memory allocation. The *System* unit is used automatically by any unit or program and doesn't need to be referred to in a **uses** clause.

Dos and WinDos units

To read about the Dos and WinDos units, see Chapter 15, "Interfacing with DOS."

The *Dos* and *WinDos* units implement a number of very useful operating system and file-handling routines. None of the routines in these units are defined by Standard Pascal, so they have been placed in their own modules. For a complete description of DOS operations, refer to a DOS programmer's reference.

Crt unit

For information on the Crt unit, see page 142 in Chapter 13, "Input and output."

The *Crt* unit permits you to write programs that send their screen output directly to the BIOS or to video memory. The result is increased speed and flexibility.

Printer unit

For more about the Printer unit, see page 141 in Chapter 13, "Input and output."

The *Printer* unit lets you send standard Pascal output to your printer using *Write* and *Writeln*.

Overlay unit

To read about the Overlay unit, see Chapter 18, "Using overlays."

The *Overlay* unit enables you to reduce your program's total run-time memory requirements. In fact, you can write programs that are larger than the total available memory because only parts of your program will reside in memory at any given time.

Strings unit

See page 167 in Chapter 16, "Using null-terminated strings," for information about using the Strings unit.

With Turbo Pascal's extended syntax and the *Strings* unit, your programs can use null-terminated strings, so that they are more compatible with any Windows programs you write.

Graph unit

Read about the Graph unit in Chapter 17, "Using the Borland Graphics Interface."

The *Graph* unit supplies a set of fast, powerful graphics routines. It implements the device-independent Borland graphics handler that supports CGA, EGA, VGA, Hercules, AT&T 400, MCGA, 3270 PC, and 8514 graphics. The *Graph* unit isn't built into TURBO.TPL, but is on the same disk with the .BGI (Borland Graphic Interface) and .CHR files.

Turbo3 and Graph3 units

You'll find information on the Turbo3 and Graph3 units in the online file TURBO3.INT.

These units are provided for backward compatibility only. *Turbo3* contains two variables and several procedures no longer supported by Turbo Pascal. *Graph3* supports the full set of graphics routines—basic, advanced, and turtlegraphics—from version 3.0.

C H A P T E R

12

Standard procedures and functions

This chapter briefly describes standard (built-in) procedures and functions in Turbo Pascal and the predeclared variables defined in the *System* unit. For in-depth information about a particular procedure, function, or predeclared variable, look it up in the alphabetical listing in Chapter 1, "Library reference," in the *Programmer's Reference*.

This chapter covers

There are other standard procedures and functions also. You can read about them in Chapter 13, "Input and output."

- Flow-control procedures
- Transfer functions
- Arithmetic functions
- Ordinal procedures and functions
- String procedures and functions
- Dynamic-allocation procedures and functions
- Pointer and address functions
- Miscellaneous procedures and functions
- Predeclared variables in the *System* unit

Standard procedures and functions are predeclared. Because all predeclared entities act as if they were declared in a block surrounding the program, you can redefine the same identifier within the program.

Flow-control procedures

These are the procedures that change the flow of logic in your program:

Table 12.1
Flow-control procedures

Procedure	Description
Break	Terminates a **for**, **while**, or **repeat** statement.
Continue	Continues with the next iteration of a **for**, **while**, or **repeat** statement.
Exit	Exits immediately from the current block.
Halt	Stops program execution and returns to the operating system.
RunError	Stops program execution and generates a run-time error.

Transfer functions

The Transfer functions are listed here:

Table 12.2
Transfer functions

The transfer procedures Pack and Unpack, as defined in Standard Pascal, are not implemented by Turbo Pascal.

Function	Description
Chr	Returns a character of a specified ordinal number.
Ord	Returns the ordinal number of an ordinal-type value.
Round	Rounds a real-type value to a type *Longint* value.
Trunc	Truncates a real-type value to a type *Longint* value.

Arithmetic functions

These functions are useful in performing arithmetic operations. When you're compiling in numeric processing mode, **{$N+}**, the return values of the floating-point routines in the *System* unit (*Sqrt*, *Pi*, *Sin*, and so on) are of type *Extended* instead of *Real*.

Table 12.3
Arithmetic functions

Function	Description
Abs	Returns the absolute value of the argument.
ArcTan	Returns the arctangent of the argument.
Cos	Returns the cosine of the argument.
Exp	Returns the exponential part of the argument.
Frac	Returns the fractional part of the argument.
Int	Returns the integer part of the argument.
Ln	Returns the natural logarithm of the argument.
Pi	Returns the value of *Pi* (3.1415926535897932385).

Table 12.3: Arithmetic functions (continued)

Sin	Returns the sine of the argument.
Sqr	Returns the square of the argument.
Sqrt	Returns the square root of the argument.

Ordinal procedures and functions

The ordinal routines operate on the ordinality of a variable.

Table 12.4
Ordinal procedures and functions

Procedure or function	Description
Dec	Decrements a variable.
Inc	Increments a variable.
High	Returns the highest value in the range of the argument.
Low	Returns the lowest value in the range of the argument.
Odd	Tests if the argument is an odd number.
Pred	Returns the predecessor of the argument.
Succ	Returns the successor of the argument.

String procedures and functions

These procedures and functions are used on the traditional Pascal-style strings:

Table 12.5
String procedures and functions

Procedure or function	Description
Concat	Concatenates a sequence of strings.
Copy	Returns a substring of a string.
Delete	Deletes a substring from a string.
Insert	Inserts a substring into a string.
Length	Returns the dynamic length of a string.
Pos	Searches for a substring in a string.
Str	Converts a numeric value to its string representation.
Val	Converts the string value to its numeric representation.

Dynamic-allocation procedures and functions

The dynamic-allocation procedures and functions are used to manage the *heap*—a memory area that occupies all or some of the free memory left when a program is executed. Heap-management techniques are discussed in the section "The heap manager" in Chapter 19.

Table 12.6
Dynamic-allocation procedures and functions

Procedure or function	Description
Dispose	Disposes of a dynamic variable.
FreeMem	Disposes of a dynamic variable of a given size.
GetMem	Creates a new dynamic variable of a given size and sets a pointer variable to point to it.
MaxAvail	Returns the size of the largest contiguous free block in the heap, indicating the size of the largest dynamic variable that can be allocated at the time of the call to *MaxAvail*.
MemAvail	Returns the number of free bytes of heap storage available.
New	Creates a new dynamic variable and sets a pointer variable to point to it.

Pointer and address functions

The pointer and address functions are listed in this table:

Table 12.7
Pointer and address functions

Function	Description
Addr	Returns the address of a specified object.
Assigned	Tests to determine if a pointer or procedural variable is **nil**.
CSeg	Returns the current value of the CS register.
DSeg	Returns the current value of the DS register.
Ofs	Returns the offset of a specified object.
Ptr	Converts a segment base and an offset address to a pointer-type value.
Seg	Returns the segment of a specified object.
SPtr	Returns the current value of the SP register.
SSeg	Returns the current value of the SS register.

Miscellaneous routines

Listed below are the procedures and functions that don't fit in any other category:

Table 12.8
Miscellaneous procedures and functions

Procedure or function	Description
Exclude	Excludes an element from a set.
FillChar	Fills a specified number of contiguous bytes with a specified value.
Hi	Returns the high-order byte of the argument.
Include	Includes an element in a set.
Lo	Returns the low-order byte of the argument.
Move	Copies a specified number of contiguous bytes from a source range to a destination range.
ParamCount	Returns the number of parameters passed to the program on the command line.
ParamStr	Returns a specified command-line parameter.
Random	Returns a random number.
Randomize	Initializes built-in random generator with a random value.
SizeOf	Returns number of bytes occupied by the argument.
Swap	Swaps the high- and low-order bytes of the argument.
TypeOf	Points to an object type's virtual method table.
UpCase	Converts a character to uppercase.

Predeclared variables

The *System* unit also supplies several predeclared variables:

Table 12.9
Predeclared variables in the System unit

Variable	Type	Description
ErrorAddr	*Pointer*	Run-time error address
ExitCode	*Integer*	Exit code
ExitProc	*Pointer*	Exit procedure
FileMode	*Byte*	File open mode
FreeList	*Pointer*	Free heap-block list
FreeZero	*Pointer*	Free zero
HeapEnd	*Pointer*	Heap end
HeapError	*Pointer*	Heap-error function
HeapOrg	*Pointer*	Heap origin
HeapPtr	*Pointer*	Heap pointer
Input	*Text*	Input standard file
InOutRes	*Integer*	I/O result buffer
Output	*Text*	Output standard file

Table 12.9: Predeclared variables in the System unit (continued)

OvrCodeList	*Word*	Overlay code-segment list
OvrDebugPtr	*Pointer*	Overlay-debugger hook
OvrDosHandle	*Word*	Overlay DOS handle
OvrEmsHandle	*Word*	Overlay EMS handle
OvrHeapEnd	*Word*	Overlay-buffer end
OvrHeapOrg	*Word*	Overlay-buffer origin
OvrHeapPtr	*Word*	Overlay-buffer pointer
OvrHeapSize	*Word*	Initial overlay-buffer size
OvrLoadList	*Word*	Loaded-overlays list
PrefixSeg	*Word*	Program Segment Prefix
RandSeed	*Longint*	Random seed
SaveInt00	*Pointer*	Saved interrupt $00
SaveInt02	*Pointer*	Saved interrupt $02
SaveInt1B	*Pointer*	Saved interrupt $1B
SaveInt21	*Pointer*	Saved interrupt $21
SaveInt23	*Pointer*	Saved interrupt $23
SaveInt24	*Pointer*	Saved interrupt $24
SaveInt34	*Pointer*	Saved interrupt $34
SaveInt35	*Pointer*	Saved interrupt $35
SaveInt36	*Pointer*	Saved interrupt $36
SaveInt37	*Pointer*	Saved interrupt $37
SaveInt38	*Pointer*	Saved interrupt $38
SaveInt39	*Pointer*	Saved interrupt $39
SaveInt3A	*Pointer*	Saved interrupt $3A
SaveInt3B	*Pointer*	Saved interrupt $3B
SaveInt3C	*Pointer*	Saved interrupt $3C
SaveInt3D	*Pointer*	Saved interrupt $3D
SaveInt3E	*Pointer*	Saved interrupt $3E
SaveInt3F	*Pointer*	Saved interrupt $3F
SaveInt75	*Pointer*	Saved interrupt $75
Seg0040	*Word*	Selector for segment $0040
SegA000	*Word*	Selector for segment $A000
SegB000	*Word*	Selector for segment $B000
SegB800	*Word*	Selector for segment $B800
SelectorInc	*Word*	Selector increment
StackLimit	*Word*	Minimum stack pointer
Test8087	*Byte*	80x87 test result

For more information about these variables, look them up in the alphabetical listing in Chapter 1, "Library reference," of the *Programmer's Reference.*

C H A P T E R

13

Input and output

This chapter describes the standard (or built-in) input and output (I/O) procedures and functions of Turbo Pascal; you'll find them in the *System* unit. It also discusses input and output issues such as file input and output, devices, using the *Crt* unit, printing, and text-file device drivers.

Table 13.1
Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
Assign	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.
Close	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.

Table 13.1: Input and output procedures and functions (continued)

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

A Pascal file variable is any variable whose type is a file type. There are three classes of Pascal files: *typed*, *text*, and *untyped*.

The syntax for writing file types is given in the section "File types" on page 42.

Before a file variable can be used, it must be associated with an external file through a call to the *Assign* 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. Nontext files always allow both reading and writing whether or not 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 via 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 *Close*. 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: If an error occurs, the program terminates, displaying a run-time error message. 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 the program to halt. 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 might 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

In Turbo Pascal, the type Text is distinct from the type file of Char.

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's 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).

For text files, there are special forms of *Read* and *Write* that let you read and write values that aren't 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*.

Turbo 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). *Input* and *Output* are automatically opened before a program begins execution, as if the following statements were executed:

```
Assign(Input, '');
Reset(Input);
Assign(Output, '');
Rewrite(Output);
```

Input and *Output* are automatically closed after a program finishes executing.

If a program uses the *Crt* standard unit, *Input* and *Output* no longer refer to the standard input and output files.

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 *Assign*, and opened using *Reset*, *Rewrite*, or *Append*. A run-time error occurs if you pass a file that was opened with *Reset* to an output-oriented procedure or function. Likewise, it's an error to 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, because 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.

The FileMode variable

The *FileMode* variable defined by the *System* unit determines the access code to pass to DOS when typed and untyped files (not text files) are opened using the *Reset* procedure.

New files created using Rewrite are always opened in read/write mode, corresponding to FileMode = 2.

The default *FileMode* is 2, which allows both reading and writing. Assigning another value to *FileMode* causes all subsequent *Resets* to use that mode.

The range of valid *FileMode* values depends on the version of DOS in use. For all versions, however, the following modes are defined:

0	Read only
1	Write only
2	Read/Write

DOS version 3.x and higher defines additional modes, which are primarily concerned with file-sharing on networks. (For more details, see your DOS programmer's reference manual.)

Devices in Turbo Pascal

Turbo Pascal and the DOS operating system regard external hardware, such as the keyboard, the display, and the printer, as *devices*. From the programmer's point of view, a device is treated as a file and is operated on through the same standard procedures and functions as files.

Turbo Pascal supports two kinds of devices: DOS devices and text-file devices.

DOS devices

DOS devices are implemented through reserved file names that have a special meaning attached to them. DOS devices are completely transparent—in fact, Turbo Pascal isn't even aware when a file variable refers to a device instead of a disk file. For example, the program

```
var
  Lst: Text;
begin
  Assign(Lst, 'LPT1');
  Rewrite(Lst);
  Writeln(Lst, 'Hello World...');
  Close(Lst);
end.
```

writes the string `"Hello World..."` on the printer, even though the syntax for doing so is exactly the same as for a disk file.

The devices implemented by DOS are used for obtaining or presenting legible input or output. Therefore, DOS devices are normally used only in connection with text files. On rare occasions, untyped files can also be useful for interfacing with DOS devices.

The CON device

CON refers to the CONsole device, in which output is sent to the display, and input is obtained from the keyboard. The *Input* and *Output* standard files and all files assigned an empty name refer to the CON device when input or output isn't redirected.

Input from the CON device is line-oriented and uses the line-editing facilities described in your DOS manual. Characters are read from a line buffer, and when the buffer becomes empty, a new line is input.

An end-of-file character is generated by pressing *Ctrl+Z*, after which the *Eof* function will return *True*.

The LPT1, LPT2, and LPT3 devices

The line-printer devices are the three possible printers you can use. If only one printer is connected, it's usually referred to as LPT1, for which the synonym can also be used.

The line-printer devices are output-only devices—an attempt to *Reset* a file assigned to one of these generates an immediate end-of-file.

☞ The standard unit *Printer* declares a text-file variable called *Lst*, and makes it refer to the LPT1 device. To easily write something on the printer from one of your programs, include *Printer* in the program's **uses** clause, and use *Write(Lst,...)* and *Writeln(Lst,...)* to produce your output.

The COM1 and COM2 devices

The communication-port devices are the two serial communication ports. The synonym AUX can be used instead of COM1.

The NUL device

The NUL device ignores anything written to it, and generates an immediate end-of-file when read from. You should use this when you don't want to create a particular file, but the program requires an input or output file name.

Text-file devices

Text-file devices are used to implement devices unsupported by DOS or to provide another set of features similar to those supplied by another DOS device. A good example of a text-file device is the CRT device implemented by the *Crt* standard unit. It provides an interface to the display and the keyboard, like the CON device in DOS, but the CRT device is much faster and supports such invaluable features as color and windows.

Unlike DOS devices, text-file devices have no reserved file names; in fact, they have no file names at all. Instead, a file is associated with a text-file device through a customized *Assign* procedure. For example, the *Crt* standard unit implements an *AssignCrt* procedure that associates text files with the CRT window.

Input and output with the Crt unit

The *Crt* unit implements a range of powerful routines that give you full control of your PC's features, such as screen mode control, extended keyboard codes, colors, windows, and sound. *Crt* can only be used in programs that run on IBM PCs, ATs, PS/2s, and true compatibles.

One of the major advantages to using *Crt* is the added speed and flexibility of screen output operations. Programs that don't use the *Crt* unit send their screen output through DOS, which adds a lot of overhead. With the *Crt* unit, output is sent directly to the BIOS or, for even faster operation, directly to video memory.

Using the Crt unit

To use the *Crt* unit, include it in your program's **uses** clause as you would any other unit:

```
uses Crt;
```

The initialization code of the *Crt* unit assigns the *Input* and *Output* standard text files to refer to the CRT instead of to DOS's standard input and output files. These statements execute at the beginning of a program:

```
AssignCrt(Input); Reset(Input);
AssignCrt(Output); Rewrite(Output);
```

This means that I/O redirection of the *Input* and *Output* files is no longer possible unless these files are explicitly assigned back to standard input and output by executing this:

```
Assign(Input,''); Reset(Input);
Assign(Output,''); Rewrite(Output);
```

Windows

Crt supports a simple yet powerful form of windows. The *Window* procedure lets you define a window anywhere on the screen. When you write in such a window, the window behaves exactly as if you were using the entire screen, leaving the rest of the screen untouched. In other words, the screen outside the window isn't accessible. Inside the window, lines can be inserted and deleted, the cursor wraps around at the right edge, and the text scrolls when the cursor reaches the bottom line.

All screen coordinates, except the ones used to define a window, are relative to the current window, and screen coordinates (1,1) correspond to the upper left corner of the window.

The default window is the entire screen.

Special characters

When writing to *Output* or a file that has been assigned to the CRT, the following control characters have special meanings:

Table 13.2
Control characters

Char	Name	Description
#7	BELL	Emits a beep from the internal speaker.
#8	BS	Moves the cursor left one column. If the cursor is already at the left edge of the current window, nothing happens.
#10	LF	Moves the cursor down one line. If the cursor is already at the bottom of the current window, the window is scrolled up one line.
#13	CR	Returns the cursor to the left edge of the current window.

Line input

When reading from *Input* or from a text file that has been assigned to *Crt*, 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. Whenever the buffer has been emptied, a new line is input.

When entering lines, the following editing keys are available:

Table 13.3
Line input editing keys

Editing key	Description
Backspace	Deletes the last character entered.
Esc	Deletes the entire input line.
Enter	Terminates the input line and stores the end-of-line marker (carriage return/line feed) in the buffer.
Ctrl+S	Same as *Backspace*.
Ctrl+D	Recalls one character from the last input line.
Ctrl+A	Same as *Esc*.
Ctrl+F	Recalls the last input line.
Ctrl+Z	Terminates the input line and generates an end-of-file marker.

Ctrl+Z will only generate an end-of-file marker if the *CheckEOF* variable has been set to *True*; it's *False* by default.

To test keyboard status and input single characters under program control, use the *KeyPressed* and *ReadKey* functions.

Crt procedures and functions

The following table lists the procedures and functions defined in the *Crt* unit.

Table 13.4
Crt unit procedures and functions

See the Programmer's Reference for more details about using the Crt 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.
Delay	Delays a specified number of milliseconds.
DelLine	Deletes the line containing the cursor and moves all lines below that line one line up. The bottom line is cleared.
GotoXY	Positions the cursor. *X* is the horizontal position. *Y* is the vertical position.
HighVideo	Selects high-intensity characters.
InsLine	Inserts an empty line at the cursor position.
KeyPressed	Returns *True* if a key has been pressed on the keyboard.
LowVideo	Selects low-intensity characters.
NormVideo	Selects normal characters.
NoSound	Turns off the internal speaker.
Sound	Starts the internal speaker.
TextBackground	Selects the background color.
TextColor	Selects the foreground character color.
TextMode	Selects a specific text mode.
Window	Defines a text window onscreen.
ReadKey	Reads a character from the keyboard.

Table 13.4: Crt unit procedures and functions (continued)

WhereX	Returns the x-coordinate of the current cursor location, relative to the current window.
WhereY	Returns the y-coordinate of the current cursor location, relative to the current window.

Crt unit constants and variables

The *Crt* unit has several constants that your programs can use. To learn more about using them, look them up in Chapter 1 of the *Programmer's Reference*. You'll find them grouped like this:

Table 13.5
Crt unit constants

Constant group	**Description**
Crt mode constants	Graphics-mode constants used as parameters for the *TextMode* procedure.
Text color constants	Constants used to set the colors of the CRT window using the *TextColor* and *TextBackground* procedures.

For example, to find the value of a constant that will color the text in your program red, look up Text Color constants, and you'll discover that the constant *Red* has a value of 4.

These are the variables in the *Crt* unit and the functions they perform:

Table 13.6
Crt unit variables

Variable	**Description**
CheckBreak	Enables and disables checks for *Ctrl+Break*.
CheckEOF	Enables and disables the end-of-file character.
CheckSnow	Enables and disables "snow checking".
DirectVideo	Enables and disables direct memory access for *Write* and *Writeln* statements that output to the screen.
LastMode	Stores the current video mode when each time *TextMode* is called.
TextAttr	Stores the currently-selected text attributes.
WindMin	Stores the screen coordinates of the upper-left corner of the current window.
WindMax	Stores the screen coordinates of the lower-right corner of the current window.

Text-file device drivers

Turbo Pascal lets you define your own text-file device drivers. A text-file device driver is a set of four functions that completely implement an interface between Turbo 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 *TTextRec* is the text file record type defined in the "Internal data formats," section in Chapter 19. 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 (like the *AssignCrt* procedure in the *Crt* unit). 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.

For example, assuming 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 Turbo Pascal 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 *Close* procedure from determining the current mode.

The InOut function

The *InOut* function is called by the *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln*, and *Close* 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 won'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 *Close* 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*, Turbo Pascal's file system calls the *InOut* function to ensure that all characters have been written to the device.

C H A P T E R

14

Using the 80x87

There are two kinds of numbers you can work with in Turbo Pascal: integers (*Shortint*, *Integer*, *Longint*, *Byte*, *Word*) 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 80486 processor, which has a numeric coprocessor built in.

Turbo Pascal provides optimal floating-point performance whether or not you have an 80x87:

- For programs running on any PC, with or without an 80x87, Turbo Pascal provides the *Real* type and an associated library of software routines that handle floating-point operations. The *Real* type occupies 6 bytes of memory, providing a range of 2.9 $\times 10^{-39}$ to 1.7×10^{38} with 11 to 12 significant digits. The software floating-point library is optimized for speed and size, trading in some of the fancier features provided by the 80x87 processor.
- If you need the added precision and flexibility of the 80x87, you can instruct Turbo Pascal to produce code that uses the 80x87 chip. This gives you access to four additional real types (*Single*,

Double, *Extended*, and *Comp*), and an *Extended* floating-point range of 3.4×10^{-4951} to 1.1×10^{4932} with 19 to 20 significant digits.

You switch between the two different models of floating-point code generation using the **$N** compiler directive or the 80x87 Code check box in the Options | Compiler dialog box. The default state is {**$N-**}, and in this state, the compiler uses the 6-byte floating-point library, allowing you to operate only on variables of type *Real*. In the {**$N+**} state, the compiler generates code for the 80x87, giving you increased precision and access to the four additional real types.

☞ When you're compiling in 80x87 Code mode, {**$N+**}, the return values of the floating-point routines in the *System* unit (*Sqrt*, *Pi*, *Sin*, and so on) are of type *Extended* instead of *Real*:

```
{$N+}
begin
  Writeln(Pi);                          { 3.14159265358979E+0000 }
end.

{$N-}
begin
  Writeln(Pi)                               { 3.1415926536E+00 }
end.
```

Even if you don't have an 80x87 in your machine, you can instruct Turbo Pascal to include a run-time library that emulates the numeric coprocessor. Then, if an 80x87 is present, it's used. If it's not present, the run-time library emulates it, although your program runs a bit slower than if an 80x87 were present.

The **$E** compiler directive and the Emulation check box in the Options | Compiler dialog box are used to enable and disable 80x87 emulation. The default state is {**$E+**}, and in this state, the full 80x87 emulator is automatically included in programs that use the 80x87. In the {**$E-**} state, a substantially smaller floating-point library is used, and the final .EXE file can run only on machines with an 80x87.

☞ The **$E** compiler directive has no effect if used in a unit; it only applies to the compilation of a program. Also, if the program is compiled in the {**$N-**} state, and if all the units used by the program were compiled with {**$N-**}, then an 80x87 run-time library isn't required, and the **$E** compiler directive is ignored.

The remainder of this chapter discusses special issues concerning Turbo Pascal programs that use the 80x87 coprocessor.

The 80x87 data types

For programs that use the 80x87, Turbo Pascal provides four floating-point types in addition to the type *Real*.

- 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* can 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 appropriate for representing monetary values as integral values of cents or mils (thousandths) in business applications.

Whether or not you have an 80x87 processor, the 6-byte *Real* type is always available, so you don't have to modify your source code when switching to the 80x87, and you can still read data files generated by programs that use software floating point.

Note, however, that 80x87 floating-point calculations on variables of type *Real* are slightly slower than on other types. This is because the 80x87 can't directly process the *Real* format—instead, calls must be made to library routines to convert *Real* values to *Extended* before operating on them. If you're concerned with optimum speed and always run on a system with an 80x87, you might want to 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. Turbo Pascal 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:

```
{$N+}
var
  X ,A ,B ,C: Real;
begin
  X := (B + Sqrt(B * B - A * C)) / A;
end;
```

Turbo Pascal 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 Turbo Pascal'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 the following 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 differ after 7 to 8 digits. Similarly, the 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, Turbo Pascal uses the 80x87's stack for storing temporary results.

In theory, very complicated real-type expressions can overflow the 80x87 stack. This isn't likely to occur, however, because the expression would need to generate more than eight temporary results.

A more tangible danger lies in recursive function calls. If such constructs aren't coded correctly, they can easily overlow the 80x87 stack.

Consider the following procedure that calculates Fibonacci numbers using recursion:

```
function Fib(N: Integer): Extended;
begin
  if N = 0 then
    Fib := 0.0
  else
    if N = 1 then
      Fib := 1.0
    else
  Fib := Fib(N - 1) + Fib(N - 2);
end;
```

A call to this version of *Fib* will cause an 80x87 stack overflow for values of *N* larger than 8. This is because the calculation of the last assignment requires a temporary on the 80x87 stack to store the result of *Fib(N-1)*. Each recursive invocation allocates one such temporary, causing an overflow the ninth time. The correct construct in this case is this:

```
function Fib(N: Integer): Extended;
var
  F1, F2: Extended;
begin
  if N = 0 then
    Fib := 0.0
  else
    if N = 1 then
      Fib := 1.0
    else
    begin
      F1 := Fib(N - 1);
      F2 := Fib(N - 2);
      Fib := F1 + F2;
    end;
end;
```

The temporary results are now stored in variables allocated on the 8086 stack. (The 8086 stack can also overflow, but this would usually require many more recursive calls.)

Writing reals with the 80x87

In the {**$N+**} state, the *Write* and *Writeln* standard procedures output four digits, not two, for the exponent in a floating-point decimal string to provide for the extended numeric range. The *Str* standard procedure also returns a four-digit exponent when floating-point format is selected.

Units using the 80x87

Units that use the 80x87 can be used only by other units or programs that are compiled in the {**$N+**} state.

The fact that a unit uses the 80x87 is determined by whether it contains 80x87 instructions—not by the state of the **$N** compiler directive at the time of its compilation. This makes the compiler more forgiving in cases where you accidentally compile a unit that doesn't use the 80x87 in the {**$N+**} state.

☞ When you compile in numeric processing mode ({**$N+**}), the return values of the floating-point routines in the *System* unit—*Sqrt*, *Pi*, *Sin*, and so on—are of type *Extended* instead of *Real*.

Detecting the 80x87

The Turbo Pascal 80x87 run-time library built into your program (compiled with {**$N+**}) includes startup code that automatically detects the presence of an 80x87 chip. If an 80x87 is available, then the program will use it. If one isn't present, the program will use the emulation run-time library. If the program was compiled in the {**$E-**} state, and an 80x87 could not be detected at startup, the program displays "Numeric coprocessor required," and ends.

You might want to override this default autodetection behavior occasionally. For example, your own system might have an 80x87, but you want to verify that your program will work as intended on systems without a coprocessor. Or your program might need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that an 80x87 is present when it's not, or vice versa).

Turbo Pascal provides an option for overriding the startup code's default autodetection logic: the *87* environment variable.

You set the *87* environment variable at the DOS prompt with the SET command, like this:

```
SET 87 = Y
```

or

```
SET 87 = N
```

Setting the *87* environment variable to *N* (for no) tells the startup code that you don't want to use the 80x87, even though it might be present in the system. Conversely, setting the *87* environment variable to *Y* (for yes) means that the coprocessor is there, and you want the program to use it.

☞ If you set *87* = *Y* when there is no 80x87 available, your program will either crash or hang!

If the *87* environment variable has been defined (to any value) but you want to undefine it, enter this at the DOS prompt:

```
SET 87 =
```

If an *87* = *Y* entry is present in the DOS environment, or if the autodetection logic succeeds in detecting a coprocessor, the startup code executes additional checks to determine what kind of coprocessor it's (8087, 80287, or 80387). This is required so that Turbo Pascal can correctly handle certain incompatibilities that exist between the different coprocessors.

The result of the autodetection and the coprocessor classification is stored in the *Test8087* variable (which is declared by the *System* unit). The following values are defined:

Table 14.1
Test8087 variable values

Value	Definition
0	No coprocessor detected
1	8087 detected
2	80287 detected
3	80387 or 80486 detected

Your program can examine the *Test8087* variable to determine the characteristics of the system it's running on. In particular, *Test8087* can be examined to determine if floating-point instructions are being emulated or truly executed.

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.

C H A P T E R

15

Interfacing with DOS

The *Dos* and *WinDos* units implement a number of operating system and file-handling routines. None of the routines in the *Dos* and *WinDos* units are defined by Standard Pascal, so they have been placed in their own modules.

For a complete description of DOS operations, refer to a DOS programmer's reference manual.

Read more about the differences between standard Pascal-style and null-terminated strings on page 167.

The primary difference between the *Dos* and *WinDos* units is that the procedures and functions of the *Dos* unit use standard Pascal-style strings and the *WinDos* procedures and functions use null-terminated strings. A standard Pascal-style string is a length byte followed by a sequence of characters. A null-terminated string is a sequence of non-null characters followed by a NULL (#0) character.

To read about the Strings unit, see Chapter 16, "Using null-terminated strings."

Most of the time, you'll probably want to use the *Dos* unit for the programs you write, as most Pascal programs traditionally use the Pascal-style strings. If you also develop applications for the Windows environment, however, you'll be able to write code you can more easily share between the DOS and Windows platforms if you use the *WinDos* unit along with the *Strings* unit; Windows requires the use of null-terminated strings.

You also might want to use the *WinDos* and *Strings* units if you have a C data file you want to use or convert. The C language uses null-terminated strings.

This chapter discusses the *Dos* unit first. To read about the *WinDos* unit, turn to page 163.

Dos unit procedures and functions

These are the procedures and functions in the *Dos* unit. To use them, you must refer to the *Dos* unit with the **uses** statement in your program.

Table 15.1
Dos unit date and time procedures

Procedure	Description
GetDate	Returns the current date set in the operating system.
GetFTime	Returns the date and time a file was last modified.
GetTime	Returns the current time set in the operating system.
PackTime	Converts a *DateTime* record into a 4-byte, packed date-and-time *Longint* used by *SetFTime*.
SetDate	Sets the current date in the operating system.
SetFTime	Sets the date and time a file was last modified.
SetTime	Sets the current time in the operating system.
UnpackTime	Converts a 4-byte, packed date-and-time *Longint* returned by *GetFTime*, *FindFirst*, or *FindNext* into an unpacked *DateTime* record.

Table 15.2
Dos unit interrupt support procedures

Procedure	Description
GetIntVec	Returns the address stored in a specified interrupt vector.
Intr	Executes a specified software interrupt with a specified *Registers* package.
MsDos	Executes a DOS function call with a specified *Registers* package.
SetIntVec	Sets a specified interrupt vector to a specified address.

Table 15.3
Dos unit disk status functions

Function	Description
DiskFree	Returns the number of free bytes of a specified disk drive.
DiskSize	Returns the total size in bytes of a specified disk drive.

Table 15.4
Dos unit file-handling procedures and functions

Procedure or function	Description
FExpand	Takes a file name and returns a fully qualified file name (drive, directory, name, and extension).
FSearch	Searches for a file in a list of directories.
FSplit	Splits a file name into its three component parts (drive and directory, file name, and extension).
FindFirst	Searches the specified directory for the first entry matching the specified file name and set of attributes.
FindNext	Returns the next entry that matches the name and attributes specified in a previous call to *FindFirst*.
GetFAttr	Returns the attributes of a file.
SetFAttr	Sets the attributes of a file.

Table 15.5
Dos unit environment-handling functions

Function	Description
EnvCount	Returns the number of strings contained in the DOS environment.
EnvStr	Returns a specified environment string.
GetEnv	Returns the value of a specified environment variable.

Table 15.6
Dos unit process-handling procedures

Procedure	Description
Exec	Executes a specified program with a specified command line.
Keep	*Keep* (or Terminate Stay Resident) terminates the program and makes it stay in memory.
SwapVectors	Swaps all saved interrupt vectors with the current vectors.

Table 15.7
Dos unit miscellaneous procedures and functions

Procedure or function	Description
DosVersion	Returns the DOS version number.
GetCBreak	Returns the state of *Ctrl+Break* checking in DOS.
GetVerify	Returns the state of the verify flag in DOS.
SetCBreak	Sets the state of *Ctrl-Break* checking in DOS.
SetVerify	Sets the state of the verify flag in DOS.

Dos unit constants, types, and variables

Each of the constants, types, and variables defined by the *Dos* unit are briefly discussed in this section. For more information, look them up in Chapter 1, "Library reference," in the *Programmer's Reference.*

Constants

The *Dos* unit defines several constants. These constants can be grouped by their function. To learn more about these constants, look them up as part of the group they belong to. For example, to find the value of *FParity*, look up "Flag constants" in the *Programmer's Reference.*

Table 15.8
Dos unit constants

Constant group	Description
Flag	Used to test individual flag bits in the Flags register after a call to *Intr* or *MsDos*: *FCarry, FParity, FAuxiliary, FZero, FSign, FOverflow*
fmXXXX	Defines the allowable values for *Mode* field of a *TextRec* text file record: *fmClosed, fmInput, fmOutput, fmInOut*
File attribute	Used to construct file attributes for use with *FindFirst*, *GetFAttr*, and *SetFAttr*: *ReadOnly, Hidden, SysFile, VolumeID, Directory, Archive, AnyFile*

Types

The *Dos* unit defines these types:

Table 15.9
Dos unit types

Types	Description
File record types	*FileRec* defines the internal data format for both typed and untyped files; *TextRec* is the internal format of a variable of type *Text*.
Registers	Variables of this type are used by *Intr* and *MsDos* to specify the input register contents and examine the output register contents of a software interrupt.
DateTime	Variables of this type are used to examine and construct 4-byte, packed date-and-time values for *GetFTime*, *SetFTime*, *FindFirst*, and *FindNext*.
SearchRec	Variables of this type are used by *FindFirst* and *FindNext* to scan directories.
File-handling string types	String types used by various procedures and functions in the *Dos* unit: *ComStr*, *PathStr*, *DirStr*, *NameStr*, *ExtStr*.

Variables

DosError is used by many of the routines in the *Dos* unit to report errors.

WinDos unit procedures and functions

These are the procedures and functions in the *WinDos* unit. To use them, you must refer to the *WinDos* unit with the **uses** statement in your program:

Table 15.10
WinDos date and time procedures

Procedure	Description
GetDate	Returns the current date set in the operating system.
GetFTime	Returns the date and time a file was last modified.
GetTime	Returns the current time set in the operating system.
PackTime	Converts a *TDateTime* record into a 4-byte, packed date-and-time *Longint* used by *SetFTime*.
SetDate	Sets the current date in the operating system.
SetFTime	Sets the date and time a file was last modified.

Table 15.10: WinDos date and time procedures (continued)

SetTime	Sets the current time in the operating system.
UnpackTime	Converts a 4-byte, packed date-and-time *Longint* returned by *GetFTime*, *FindFirst*, or *FindNext* into an unpacked *TDateTime* record.

Table 15.11
WinDos unit interrupt support procedures

Don't use these functions when running Windows in protected mode.

Procedure	Description
GetIntVec	Returns the address stored in a specified interrupt vector.
Intr	Executes a specified software interrupt with a specified *TRegisters* package.
MsDos	Executes a DOS function call with a specified *TRegisters* package.
SetIntVec	Sets a specified interrupt vector to a specified address.

Table 15.12
WinDos unit disk status functions

Function	Description
DiskFree	Returns the number of free bytes of a specified disk drive.
DiskSize	Returns the total size in bytes of a specified disk drive.

Table 15.13
File-handling procedures and functions

Procedure or function	Description
FileExpand	Takes a file name and returns a fully qualified file name (drive, directory, name, and extension).
FileSearch	Searches for a file in a list of directories.
FileSplit	Splits a file name into its three component parts (directory, file name, and extension).
FindFirst	Searches the specified directory for the first entry matching the specified file name and set of attributes.
FindNext	Returns the next entry that matches the name and attributes specified in a previous call to *FindFirst*.
GetFAttr	Returns the attributes of a file.
SetFAttr	Sets the attributes of a file.

Table 15.14
WinDos unit directory-handling procedures and functions

Procedure or function	Description
CreateDir	Creates a new subdirectory.
GetCurDir	Returns the current directory of a specified drive.
RemoveDir	Removes a subdirectory.
SetCurDir	Changes the current directory.

Table 15.15
WinDos unit environment-handling functions

Function	Description
GetArgCount	Returns the number of parameters passed to the program on the command line.
GetArgStr	Returns a specified command-line argument.
GetEnvVar	Returns a pointer to the value of a specified environment variable.

Table 15.16
WinDos unit miscellaneous procedures and functions

Procedure or function	Description
DosVersion	Returns the DOS version number.
GetCBreak	Returns the state of *Ctrl+Break* checking in DOS.
GetVerify	Returns the state of the verify flag in DOS.
SetCBreak	Sets the state of *Ctrl+Break* checking in DOS.
SetVerify	Sets the state of the verify flag in DOS.

WinDos unit constants, types, and variables

Each of the constants, types, and variables defined by the *WinDos* unit are briefly discussed in this section.

Constants

The *WinDos* unit uses several constants. These constants can be grouped by their function. To learn more about these constants, look them up as part of the constant group they belong to. For example, to find the value of *fParity*, look up "Flag constants" in the *Programmer's Reference*.

Table 15.17
WinDos constants

Constant group	Description
Flag	Test individual flag bits in the Flags register after a call to *Intr* or *MsDos*: *fCarry, fParity, fAuxiliary, fZero, fSign, fOverflow*
fmXXXX	Used by file-handling procedures when opening and closing disk files: *fmClosed, fmInput, fmOutput, fmInOut*
faXXXX	Test, set, and clear file attribute bits in connection with the file-handling procedures: *faReadOnly, faHidden, faSysFile, faVolumeID, faDirectory, faArchive, faAnyFile*
fsXXXX	Maximum file-name component string lengths used by *FileSearch* and *FileExpand*: *fsPathName, fsDirectory, fsFileName, fsExtension*
fcXXXX	Return flags used by the *FileSplit* function: *fcExtension, fcFileName, fcDirectory, fcWildcards*

Types

The *WinDos* unit defines these types:

Table 15.18
WinDos unit types

Types	Description
File record types	*TFileRec* is used for both typed and untyped files; *TTextRec* is the internal format of a variable of type text.
TRegisters	Variables of this type are used by *Intr* and *MsDos* to specify the input register contents and examine the output register contents of a software interrupt.
TDateTime	Variables of this type are used to examine and construct 4-byte, packed date-and-time values for *GetFTime*, *SetFTime*, *FindFirst*, and *FindNext*.
TSearchRec	Variables of this type are used by *FindFirst* and *FindNext* to scan directories.

Variables

DosError is used by many of the routines in the *WinDos* unit to report errors.

C H A P T E R

16

Using null-terminated strings

Turbo Pascal supports a class of character strings called *null-terminated strings*. With Turbo Pascal's extended syntax and the *Strings* unit, your programs can use null-terminated strings by simply referring to the *Strings* unit with the **uses** statement in your program.

What is a null-terminated string?

The compiler stores a traditional Turbo 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, but the 16-bit architecture of DOS does impose an upper limit of 65,535 characters.

Strings unit functions

Turbo Pascal has no built-in routines specifically for null-terminated string handling. Instead you'll find all such functions in the *Strings* unit. Among them are *StrPCopy*, which you can use to copy a Pascal string to a null-terminated string, and *StrPas*, which you can use to convert a null-terminated string to a Pascal string. Here's a brief description of each function:

Table 16.1
Strings unit functions

Function	Description
StrCat	Appends a source string to the end of a destination string and returns a pointer to the destination string.
StrComp	Compares two strings, *S1* and *S2*, and returns a value less than zero if *S1* < *S2*, zero if *S1* = *S2*, or greater than zero if *S1* > *S2*.
StrCopy	Copies a source string to a destination string and returns a pointer to the destination string.
StrECopy	Copies a source string to a destination string and returns a pointer to the end of the destination string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Appends a source string to the end of a destination string, ensuring that the length of the resulting string doesn't exceed a given maximum, and returns a pointer to the destination string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies up to a given number of characters from a source string to a destination string and returns a pointer to the destination string.
StrEnd	Returns a pointer to the end of a string (that is, a pointer to the null character that terminates a string).
StrDispose	Disposes of a previously allocated string.
StrLen	Returns the length of a string.
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lowercase and returns a pointer to the string.
StrMove	Moves a block of characters from a source string to a destination string, and returns a pointer to the destination string. The two blocks may overlap.
StrNew	Allocates a string on the heap.
StrPas	Converts a null-terminated string to a Pascal string.
StrPCopy	Copies a Pascal string to a null-terminated string and returns a pointer to the null-terminated string.
StrPos	Returns a pointer to the first occurrence of a given substring within a string, or **nil** if the substring doesn't occur within the string.

Table 16.1: Strings unit functions (continued)

StrRScan	Returns a pointer to the last occurrence of a given character within a string, or **nil** if the character doesn't occur within the string.
StrScan	Returns a pointer to the first occurrence of a given character within a string, or **nil** if the character doesn't occur within the string.
StrUpper	Converts a string to uppercase and returns a pointer to the 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. Turbo 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 you enable the extended syntax with **$X**, 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's 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 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.

☞ For information about *PChar* operations, see page 71.

Null-terminated strings and standard procedures

Turbo 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. For more details, see the descriptions of these standard procedures in the *Programmer's Reference.*

An example using string-handling functions

Here's a code example that shows how we used some of the string-handling functions when we wrote the *FileSplit* function in the *WinDos* unit:

```
{ Maximum file name component string lengths }

const
  fsPathName  = 79;
  fsDirectory = 67;
  fsFileName  = 8;
  fsExtension = 4;

{ FileSplit return flags }

const
  fcExtension = $0001;
  fcFileName  = $0002;
  fcDirectory = $0004;
  fcWildcards = $0008;

{ FileSplit splits the file name specified by Path into its     }
{ three components. Dir is set to the drive and directory path  }
{ with any leading and trailing backslashes, Name is set to the }
{ file name, and Ext is set to the extension with a preceding   }
{ period. If a component string parameter is NIL, the          }
{ corresponding part of the path is not stored. If the path     }
{ does not contain a given component, the returned component    }
{ string is empty. The maximum lengths of the strings returned  }
```

```
{ in Dir, Name, and Ext are defined by the fsDirectory,        }
{ fsFileName, and fsExtension constants. The returned value is  }
{ a combination of the fcDirectory, fcFileName, and fcExtension }
{ bit masks, indicating which components were present in the    }
{ path. If the name or extension contains any wildcard          }
{ characters (* or ?), the fcWildcards flag is set in the       }
{ returned value.                                               }

function FileSplit(Path, Dir, Name, Ext: PChar): Word;
var
  DirLen, NameLen, Flags: Word;
  NamePtr, ExtPtr: PChar;
begin
  NamePtr := StrRScan(Path, '\');
  if NamePtr = nil then NamePtr := StrRScan(Path, ':');
  if NamePtr = nil then NamePtr := Path else Inc(NamePtr);
  ExtPtr := StrScan(NamePtr, '.');
  if ExtPtr = nil then ExtPtr := StrEnd(NamePtr);
  DirLen := NamePtr - Path;
  if DirLen > fsDirectory then DirLen := fsDirectory;
  NameLen := ExtPtr - NamePtr;
  if NameLen > fsFilename then NameLen := fsFilename;
  Flags := 0;
  if (StrScan(NamePtr, '?') <> nil) or
    (StrScan(NamePtr, '*') <> nil) then
    Flags := fcWildcards;
  if DirLen <> 0 then Flags := Flags or fcDirectory;
  if NameLen <> 0 then Flags := Flags or fcFilename;
  if ExtPtr[0] <> #0 then Flags := Flags or fcExtension;
  if Dir <> nil then StrLCopy(Dir, Path, DirLen);
  if Name <> nil then StrLCopy(Name, NamePtr, NameLen);
  if Ext <> nil then StrLCopy(Ext, ExtPtr, fsExtension);
  FileSplit := Flags;
end;
```

C H A P T E R

17

Using the Borland Graphics Interface

The *Graph* unit features a complete library of more than 50 graphics routines that range from high-level calls such as *SetViewPort*, *Circle*, *Bar3D*, and *DrawPoly*, to bit-oriented routines such as *GetImage* and *PutImage*. It supports several fill and line styles and there are several fonts that may be magnified, justified, and oriented horizontally or vertically.

Your license agreement permits you to distribute the .CHR and .BGI files along with your programs.

To compile a program that uses the *Graph* unit, you'll need your program's source code, the compiler, and access to the standard units in the run-time library (TURBO.TPL) and the *Graph* unit (GRAPH.TPU):

To run a program that uses the *Graph* unit, you'll need one or more of the graphics drivers (.BGI files listed in the next section) in addition to your .EXE program. Also, if your program uses any stroked fonts, you'll need one or more font (.CHR) files as well.

Drivers

Graphics drivers are provided for the following graphics adapters (and true compatibles):

- CGA
- MCGA
- EGA
- VGA
- Hercules
- AT&T 400 line
- 3270 PC
- IBM 8514

Each driver contains code and data and is stored in a separate file on disk. At run time, the *InitGraph* procedure identifies the graphics hardware, loads and initializes the appropriate graphics driver, puts the system into graphics mode, and then returns control to the calling routine. The *CloseGraph* procedure unloads the driver from memory and restores the previous video mode. You can switch back and forth between text and graphics modes using the *RestoreCrtMode* and *SetGraphMode* routines. To load the driver files yourself or link them into your .EXE file, refer to *RegisterBGIdriver* in Chapter 1, "Library reference," in the *Programmer's Reference*.

Graph supports computers with dual monitors. When *Graph* is initialized by calling *InitGraph*, the correct monitor will be selected for the graphics driver and mode requested. When terminating a graphics program, the previous video mode will be restored. If autodetection of graphics hardware is requested on a dual monitor system, *InitGraph* will select the monitor and graphics card that will produce the highest quality graphics output.

Table 17.1
BGI drivers

Driver	**Equipment**
ATT.BGI	AT&T 6300 (400 line)
CGA.BGI	IBM CGA, MCGA
EGAVGA.BGI	IBM EGA, VGA
HERC.BGI	Hercules monochrome
IBM8514.BGI	IBM 8514
PC3270.BGI	IBM 3270 PC

IBM 8514 support

Turbo Pascal supports the IBM 8514 graphics card, a high-resolution graphics card capable of resolutions up to 1024 × 768 pixels and a color palette of 256 colors from a list of 256K colors. The driver file name is IBM8514.BGI.

Turbo Pascal can't properly autodetect the IBM 8514 graphics card (the autodetection logic recognizes it as VGA). Therefore, to use the IBM 8514 card, the *GraphDriver* variable must be assigned the value IBM8514 (which is defined in the *Graph* unit) when *InitGraph* is called. You should not use *DetectGraph* (or *Detect* with *InitGraph*) with the IBM 8514 unless you want the emulated VGA mode.

The supported modes of the IBM 8514 card are IBM8514LO (640 × 480 pixels), and IBM8514HI (1024 × 768 pixels). Both mode constants are defined in the interface for GRAPH.TPU or GRAPH.TPP.

The IBM 8514 uses three 6-bit values to define colors. There is a 6-bit Red, Green, and Blue component for each defined color. The *SetRGBPalette* procedure allows you to define colors for the IBM 8514; it's defined in the *Graph* unit as:

```
procedure SetRGBPalette(ColorNum, Red, Green, Blue: Word);
```

The argument *ColorNum* defines the palette entry to be loaded. *ColorNum* is an integer from 0 to 255 (decimal). The arguments *Red*, *Green*, and *Blue* define the component colors of the palette entry. Only the lower byte of these values is used, and out of this byte, only the 6 most-significant bits are loaded in the palette.

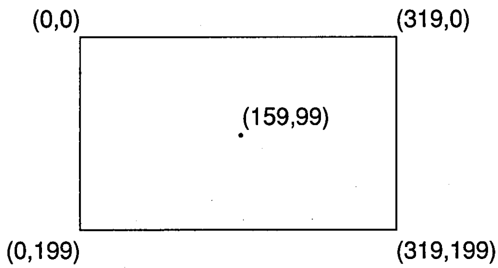
The other palette manipulation routines of the graphics library can't be used with the IBM 8514 driver (that is, *SetAllPalette*, *SetPalette*, and *GetPalette*).

For compatibility with the balance of the IBM graphics adapters, the BGI driver defines the first 16 palette entries of the IBM 8514 to the default colors of the EGA/VGA. These values can be used as is, or changed using the *SetRGBPalette* routine.

Coordinate system

By convention, the upper left corner of the graphics screen is (0,0). The *x* values, or columns, increment to the right. The *y* values, or rows, increment downward. In 320×200 mode on a CGA, the screen coordinates for each of the four corners with a specified point in the middle of the screen would look like this:

Figure 17.1
Screen with xy-coordinates



Current pointer

Many graphics systems support the notion of a current pointer (CP). The CP is similar in concept to a text mode cursor except that the CP isn't visible.

In text mode, this *Write* statement

```
Write('ABC');
```

leaves the cursor in the column immediately following the letter C. If the C is written in column 80, then the cursor wraps around to column 1 of the next line. If the C is written in column 80 on the 25th line, the entire screen scrolls up one line, and the cursor is in column 1 of line 25.

```
MoveTo(0,0)
LineTo(20,20)
```

In graphics mode, the preceding *LineTo* statement leaves the CP at the last point referenced (20,20). The actual line output is clipped to the current viewport if clipping is active. Note that the CP is never clipped.

The *MoveTo* command is the equivalent of *GoToXY*. Its only purpose is to move the CP. Only the commands that *use* the CP *move* the CP: *InitGraph, MoveTo, MoveRel, LineTo, LineRel, OutText, SetGraphMode, GraphDefaults, ClearDevice, SetViewPort,* and *ClearViewPort*. The latter five commands move the CP to (0,0).

Text

An 8x8 bitmapped font and several stroked fonts are included for text output while in graphics mode. A bitmapped character is defined by an 8x8 matrix of pixels. A stroked font is defined by a series of vectors that tell the graphics system how to draw the font.

The advantage of using a stroked font is apparent when you start to draw large characters. Because a stroked font is defined by vectors, it retains good resolution and quality when the font is enlarged.

When a bitmapped font is enlarged, the matrix is multiplied by a scaling factor and, as the scaling factor becomes larger, the characters' resolution becomes coarser. For small characters, the bitmapped font is usually sufficient, but for larger text you will want to select a stroked font.

The justification of graphics text is controlled by the *SetTextJustify* procedure. Scaling and font selection is done with the *SetTextStyle* procedure. Graphics text is output by calling either the *OutText* or *OutTextXY* procedures. Inquiries about the current text settings are made by calling the *GetTextSettings* procedure. The size of stroked fonts can be customized by the *SetUserCharSize* procedure.

Each stroked font is kept in its own file on disk with a .CHR file extension. Font files can be loaded from disk automatically by the *Graph* unit at run time (as described), or they can be linked in or loaded by the user program and "registered" with the *Graph* unit.

Turbo Pascal provides a special utility, BINOBJ.EXE, that converts a font file (or any binary data file, for that matter) to an .OBJ file that can be linked into a unit or program using the {**$L**} compiler directive. This makes it possible for a program to have all its font files built into the .EXE file. (Read the comments at the beginning of the BGILINK.PAS sample program.)

Figures and styles

All kinds of support routines are provided for drawing and filling figures, including points, lines, circles, arcs, ellipses, rectangles, polygons, bars, 3-D bars, and pie slices. Use *SetLineStyle* to control whether lines are thick or thin, or whether they are solid, dotted, or built using your own pattern.

Use *SetFillStyle* and *SetFillPattern*, *FillPoly* and *FloodFill* to fill a region or a polygon with cross-hatching or other intricate patterns.

Viewports and bit images

The *SetViewPort* procedure makes all output commands operate in a rectangular region onscreen. Plots, lines, figures—all graphics output—are viewport-relative until the viewport is changed. Other routines are provided to clear a viewport and read the current viewport definitions. If clipping is active, all graphics output is clipped to the current port. Note that the CP is never clipped.

GetPixel and *PutPixel* are provided for reading and plotting pixels. *GetImage* and *PutImage* can be used to save and restore rectangular regions onscreen. They support the full complement of *BitBlt* operations (copy, **xor**, **or**, **and**, **not**).

Paging and colors

There are many other routines that support palettes, colors, multiple graphic pages (EGA, VGA, and Hercules only), and so on.

Error handling

Internal errors in the *Graph* unit are returned by the function *GraphResult*. *GraphResult* returns an error code that reports the status of the last graphics operation. Find the error return codes under GraphResult Errors in Chapter 1, "Library reference," in the *Programmer's Reference*.

The following routines set *GraphResult*:

Bar	*ImageSize*	*SetFillPattern*
Bar3D	*InitGraph*	*SetFillStyle*
ClearViewPort	*InstallUserDriver*	*SetGraphBufSize*
CloseGraph	*InstallUserFont*	*SetGraphMode*
DetectGraph	*PieSlice*	*SetLineStyle*
DrawPoly	*RegisterBGIdriver*	*SetPalette*
FillPoly	*RegisterBGIfont*	*SetTextJustify*
FloodFill	*SetAllPalette*	*SetTextStyle*
GetGraphMode		

GraphResult is reset to zero after it has been called. Therefore, the user should store the value of *GraphResult* into a temporary variable and then test it.

Getting started

Here's a simple graphics program:

```
program GrafTest;
uses
  Graph;
const
  S = 'Borland Graphics Interface (BGI)';
var
  GraphDriver: Integer;
  GraphMode: Integer;
  ErrorCode: Integer;
  Size: Word;
begin
  GraphDriver := Detect;                    { Set flag: do detection }
  InitGraph(GraphDriver, GraphMode, 'C:\TP\BGI');
  ErrorCode := GraphResult;
  if ErrorCode <> GrOk then                                { Error? }
    begin
      Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
      Writeln('Program aborted...');
      Halt(1);
    end;
  Rectangle(0, 0, GetMaxX, GetMaxY);          { Draw full sceen box }
  SetTextJustify(CenterText, CenterText);            { Center text }
  Size := 3;
  repeat
    SetTextStyle(DefaultFont, HorizDir, Size);
    Dec(Size);
  until  (Size = 0) or (TextWidth(S) < GetMaxX);
  if Size <> 0 then
    OutTextXY(GetMaxX div 2, GetMaxY div 2, S);  { Center of screen }
  Readln;
  CloseGraph;
end. { GrafTest }
```

The program begins with a call to *InitGraph*, which autodetects the hardware and loads the appropriate graphics driver. For the program to run correctly, the driver and fonts must be in the same directory as the executable program, or the program must specify an explicit directory. In this example, the directory is C:\TP\BGI. If the program fails to recognize graphics hardware or an error occurs during initialization, the program displays an error message and terminates. Otherwise, the program draws a box along the edge of the screen and displays text in the center of the screen.

☞ Neither the AT&T 400 line card nor the IBM 8514 graphics adapter is autodetected. You can still use these drivers by overriding autodetection and passing *InitGraph* the driver code and a valid graphics mode. To use the AT&T driver, for example, replace the ninth and tenth lines in the preceding example with the following three lines of code:

```
GraphDriver := ATT400;
GraphMode := ATT400Hi;
InitGraph(GraphDriver, GraphMode, 'C:\TP\BGI');
```

This instructs the graphics system to load the AT&T 400 line driver located in C:\TP\BGI and set the graphics mode to 640 by 400.

Here's another example that demonstrates how to switch back and forth between graphics and text modes:

```
program GrafTst2;
uses
  Graph;
var
  GraphDriver: Integer;
  GraphMode: Integer;
  ErrorCode: Integer;
begin
  GraphDriver := Detect;                    { Set flag: do detection }
  InitGraph(GraphDriver, GraphMode, 'C:\TP\BGI');
  ErrorCode := GraphResult;
  if ErrorCode <> grOk then                                { Error? }
  begin
    Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
    Writeln('Program aborted...');
    Halt(1);
  end;
  OutText('In Graphics mode. Press <RETURN>');
  Readln;
  RestoreCRTMode;
  Write('Now in text mode. Press <RETURN>');
  Readln;
  SetGraphMode(GraphMode);
  OutText('Back in Graphics mode. Press <RETURN>');
  Readln;
  CloseGraph;
end. { GrafTst2 }
```

Note that the *SetGraphMode* call near the end of the example resets all the graphics parameters (palette, current pointer, foreground, and background colors, and so on) to the default values.

The call to *CloseGraph* restores the video mode that was detected initially by *InitGraph* and frees the heap memory that was used to hold the graphics driver.

Heap management routines

Two heap management routines are used by the *Graph* unit: *GraphGetMem* and *GraphFreeMem*. *GraphGetMem* allocates memory for graphics device drivers, stroked fonts, and a scan buffer. *GraphFreeMem* deallocates the memory allocated to the drivers. The standard routines take the following form:

```
procedure GraphGetMem(var P: Pointer; Size: Word);
{ Allocate memory for graphics }

procedure GraphFreeMem(var P: Pointer; Size: Word);
{ Deallocate memory for graphics }
```

Two pointers are defined by *Graph* that, by default, point to the two standard routines described here. The pointers are defined as follows:

```
var
  GraphGetMemPtr: Pointer;    { Pointer to memory allocation routine }
  GraphFreeMemPtr: Pointer  { Pointer to memory deallocation routine }
```

The *Graph* unit calls the heap management routines referenced by *GraphGetMemPtr* and *GraphFreeMemPtr* to allocate and deallocate memory for three different purposes:

- A multi-purpose graphics buffer whose size can be set by a call to *SetGraphBufSize* (default equals 4K)
- A device driver that is loaded by *InitGraph* (*.BGI files)
- A stroked font file that is loaded by *SetTextStyle* (*.CHR files)

The graphics buffer is always allocated on the heap. The device driver is allocated on the heap unless your program loads or links one in and calls *RegisterBGIdriver*. The font file is allocated on the heap when you select a stroked font using *SetTextStyle*—unless your program loads or links one in and calls *RegisterBGIfont*.

When the *Graph* unit is initialized, these pointers point to the standard graphics allocation and deallocation routines that are defined in the implementation section of the *Graph* unit. You can insert your own memory-management routines by assigning these pointers the address of your routines. The user-defined routines must have the same parameter lists as the standard routines and must be *far* procedures. The following is an example of user-defined allocation and deallocation routines; notice the use of *MyExitProc* to automatically call *CloseGraph* when the program terminates:

```
program UserHeapManagement;
{ Illustrates how the user can steal the heap }
{ management routines used by the Graph unit. }

uses
  Graph;

var
  GraphDriver, GraphMode: Integer;
  ErrorCode: Integer;                { Stores GraphResult return code }
  PreGraphExitProc: Pointer;               { Saves original exit proc }

procedure MyGetMem(var P: Pointer; Size: Word); far;
{ Allocate memory for graphics device drivers, fonts, and scan buffer }
begin
  GetMem(P, Size)
end; { MyGetMem }

procedure MyFreeMem(var P: Pointer; Size: Word); far;
{ Deallocate memory for graphics device drivers, fonts, and scan
  buffer }
begin
  if P <> nil then                         { Don't free nil pointers! }
  begin
    FreeMem(P, Size);
    P := nil;
  end;
end; { MyFreeMem }

procedure MyExitProc; far;
{ Always gets called when program terminates }
begin
  ExitProc := PreGraphExitProc;          { Restore original exit proc }
  CloseGraph;                                     { Do heap clean up }
end; { MyExitProc }

begin                                      { Install clean-up routine }
  PreGraphExitProc := ExitProc;
  ExitProc := @MyExitProc;

  GraphGetMemPtr := @MyGetMem;             { Control memory allocation }
  GraphFreeMemPtr := @MyFreeMem;         { Control memory deallocation }
```

```
    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, '');
    ErrorCode := GraphResult;
    if ErrorCode <> grOk then
    begin
      Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
      Readln;
      Halt(1);
    end;
    Line(0, 0, GetMaxX, GetMaxY);
    OutTextXY(1, 1, 'Press <Return>:');
    Readln;
  end.  { UserHeapManagement }
```

Graph procedures and functions

The *Graph* unit provides many procedures and functions for use in your programs:

Table 17.2: Graph unit procedures and functions

Arc	Draws a circular arc from start angle to end angle using (*x*,*y*) as the center point.
Bar	Draws a bar using the current fill style and color.
Bar3D	Draws a 3-D bar using the current fill style and color.
Circle	Draws a circle using (*x*,*y*) as the center point.
ClearDevice	Clears the currently selected output device and homes the current pointer.
ClearViewPort	Clears the current viewport.
CloseGraph	Shuts down the graphics system.
DetectGraph	Checks the hardware and determines which graphics driver and mode to use.
DrawPoly	Draws the outline of a polygon using the current line style and color.
Ellipse	Draws an elliptical arc from start angle to end angle, using (*x*,*y*) as the center point.
FillEllipse	Draws a filled ellipse using (*x*,*y*) as a center point and *XRadius* and *YRadius* as the horizontal and vertical axes.
FillPoly	Fills a polygon, using the scan converter.
FloodFill	Fills a bounded region using the current fill pattern and fill color.
GetArcCoords	Allows the user to inquire about the coordinates of the last *Arc* command.
GetAspectRatio	Returns the effective resolution of the graphics screen from which the aspect ratio (*Xasp:Yasp*) can be computed.
GetBkColor	Returns the current background color.
GetColor	Returns the current drawing color.

Table 17.2: Graph unit procedures and functions (continued)

GetDefaultPalette	Returns the default hardware palette in a record of *PaletteType*.
GetDriverName	Returns a string containing the name of the current driver.
GetFillPattern	Returns the last fill pattern set by a call to *SetFillPattern*.
GetFillSettings	Allows the user to inquire about the current fill pattern and color as set by *SetFillStyle* or *SetFillPattern*.
GetGraphMode	Returns the current graphics mode.
GetImage	Saves a bit image of the specified region into a buffer.
GetLineSettings	Returns the current line style, line pattern, and line thickness as set by *SetLineStyle*.
GetMaxColor	Returns the highest color that can be passed to *SetColor*.
GetMaxMode	Returns the maximum mode number for the currently loaded driver.
GetMaxX	Returns the rightmost column (*x* resolution) of the current graphics driver and mode.
GetMaxY	Returns the bottommost row (*y* resolution) of the current graphics driver and mode.
GetModeName	Returns a string containing the name of the specified graphics mode.
GetModeRange	Returns the lowest and highest valid graphics mode for a given driver.
GetPaletteSize	Returns the size of the palette color lookup table.
GetPixel	Gets the pixel value at (*x*,*y*).
GetPalette	Returns the current palette and its size.
GetTextSettings	Returns the current text font, direction, size, and justification as set by *SetTextStyle* and *SetTextJustify*.
GetViewSettings	Allows the user to inquire about the current viewport and clipping parameters.
GetX	Returns the x-coordinate of the current position (CP).
GetY	Returns the y-coordinate of the current position (CP).
GraphDefaults	Homes the current pointer (CP) and resets the graphics system.
GraphErrorMsg	Returns an error message string for the specified *ErrorCode*.
GraphResult	Returns an error code for the last graphics operation.
ImageSize	Returns the number of bytes required to store a rectangular region of the screen.
InstallUserDriver	Installs a vendor-added device driver to the BGI device driver table.
InstallUserFont	Installs a new font file that isn't built into the BGI system.
InitGraph	Initializes the graphics system and puts the hardware into graphics mode.
Line	Draws a line from the (x1, y1) to (x2, y2).
LineRel	Draws a line to a point that is a relative distance from the current pointer (CP).
LineTo	Draws a line from the current pointer to (*x*,*y*).

Table 17.2: Graph unit procedures and functions (continued)

MoveRel	Moves the current pointer (CP) a relative distance from its current position.
MoveTo	Moves the current graphics pointer (CP) to (*x*,*y*).
OutText	Sends a string to the output device at the current pointer.
OutTextXY	Sends a string to the output device.
PieSlice	Draws and fills a pie slice, using (*x*,*y*) as the center point and drawing from start angle to end angle.
PutImage	Puts a bit image onto the screen.
PutPixel	Plots a pixel at (*x*,*y*).
Rectangle	Draws a rectangle using the current line style and color.
RegisterBGIdriver	Registers a valid BGI driver with the graphics system.
RegisterBGIfont	Registers a valid BGI font with the graphics system.
RestoreCrtMode	Restores the original screen mode before graphics is initialized.
Sector	Draws and fills an elliptical sector.
SetActivePage	Sets the active page for graphics output.
SetAllPalette	Changes all palette colors as specified.
SetAspectRatio	Changes the default aspect ratio.
SetBkColor	Sets the current background color using the palette.
SetColor	Sets the current drawing color using the palette.
SetFillPattern	Selects a user-defined fill pattern.
SetFillStyle	Sets the fill pattern and color.
SetGraphBufSize	Lets you change the size of the buffer used for scan and flood fills.
SetGraphMode	Sets the system to graphics mode and clears the screen.
SetLineStyle	Sets the current line width and style.
SetPalette	Changes one palette color as specified by *ColorNum* and *Color*.
SetRGBPalette	Lets you modify palette entries for the IBM 8514 and the VGA drivers.
SetTextJustify	Sets text justification values used by *OutText* and *OutTextXY*.
SetTextStyle	Sets the current text font, style, and character magnification factor.
SetUserCharSize	Lets you change the character width and height for stroked fonts.
SetViewPort	Sets the current output viewport or window for graphics output.
SetVisualPage	Sets the visual graphics page number.
SetWriteMode	Sets the writing mode (copy or **xor**) for lines drawn by *DrawPoly*, *Line*, *LineRel*, *LineTo*, and *Rectangle*.
TextHeight	Returns the height of a string in pixels.
TextWidth	Returns the width of a string in pixels.

For a detailed description of each procedure or function, refer to Chapter 1, "Library reference," in the *Programmer's Reference.*

Graph unit constants, types, and variables

The *Graph* unit defines several constants, types, and variables that your programs can use.

Constants

The *Graph* constants can be grouped by their function. To learn more about these constants, see Chapter 1, "Library reference," in the *Programmer's Reference*. Look up the constant under the group it belongs to. This table will help you identify the group you want:

Table 17.3
Graph unit constant groups

Constant group	**Description**
Driver and mode	Constants that specify video drivers and modes; used with *InitGraph, DetectGraph,* and *GetModeRange.*
grXXXX	Constants that identify the type of error returned from *GraphResult.*
Color	Constants that specify colors; used with *SetPalette* and *SetAllPalette.*
Color for *SetRGBPalette*	Constants used with *SetRGBPalette* to select standard EGA colors on an IBM 8514.
Line style	Constants used to determine a line style and thickness; used with *GetLineSettings* and *SetLineStyle.*
Font control	Constants that identify fonts; used with *GetTextSettings* and *SetTextStyle.*
Justification	Constants that control horizontal and vertical justification; used with *SetTextJustify.*
Clipping	Constants that control clipping; used with *SetViewPort.*
Bar	Constants that control the drawing of a 3-D top on a bar; used with *Bar3D.*
Fill pattern	Constants that determine the pattern used to fill an area; used with *GetFillSettings* and *SetFillStyle.*

Table 17.3: Graph unit constant groups (continued)

BitBlt operators	Operators (copy, **xor**, **or**, **and**, and **not**) used with *PutImage* and, *SetWriteMode*.
MaxColors	The constant that defines the maximum number of colors used with *GetPalette*, *GetDefaultPalette*, and *SetAllPalette*.

For example, to find the constant you need to change the background screen color to green, look under Color Constants in Chapter 1, "Library reference," in the *Programmer's Reference*.

Types

The *Graph* unit defines these types:

Table 17.4
Graph unit types

Type	Description
PaletteType	The record that defines the size and colors of the palette; used by *GetPalette*, *GetDefaultPalette*, and *SetAllPalette*.
LineSettingsType	The record that defines the style, pattern, and thickness of a line; used by *GetLineSettings*.
TextSettingsType	The record that defines the text; used by *GetTextSettings*.
FillSettingsType	The record that defines the pattern and color used to fill an area; used by *GetFillSettings*.
FillPatternType	The record that defines a user-defined fill pattern; used by *GetFillPattern* and *SetFillPattern*.
PointType	A type defined for your convenience.
ViewPortType	A record that reports the status of the current viewport; used by *GetViewSettings*.
ArcCoordsType	A record that retrieves information about the last call to *Arc* or *Ellipse*; used by *GetArcCoords*.

Variables

The *Graph* unit has two variables you can use: *GraphGetMemPtr* and *GraphFreeMemPtr*. They are used by heap-management routines. Read about them in Chapter 1, "Library reference," in the *Programmer's Reference*.

C H A P T E R

18

Using overlays

Overlays are parts of a program that share a common memory area. Only the parts of the program that are required for a given function reside in memory at the same time; they can overwrite each other during execution.

Overlays can significantly reduce a program's total run-time memory requirements. In fact, with overlays you can execute programs that are much larger than the total available memory because only parts of the program reside in memory at any given time.

Turbo Pascal manages overlays at the unit level; this is the smallest part of a program that can be made into an overlay. When an overlaid program is compiled, Turbo Pascal generates an overlay file (extension .OVR) in addition to the executable file (extension .EXE). The .EXE file contains the static (nonoverlaid) parts of the program, and the .OVR file contains all the overlaid units that will be swapped in and out of memory during program execution.

Except for a few programming rules, an overlaid unit is identical to a nonoverlaid unit. In fact, as long as you observe these rules, you don't even need to recompile a unit to make it into an overlay. The decision of whether or not a to overlay a unit is made by the program that uses the unit.

When an overlay is loaded into memory, it's placed in the overlay buffer, which resides in memory between the stack segment and the heap. By default, the size of the overlay buffer is as small as possible, but it can be easily increased at run time by allocating additional space from the heap. Like the data segment and the minimum heap size, the default overlay buffer size is allocated when the .EXE is loaded. If enough memory isn't available, an error message will be displayed by DOS ("Program too big to fit in memory") or by the IDE ("Not enough memory to run program").

One very important option of the overlay manager is the ability to load the overlay file into expanded memory when sufficient space is available. Turbo Pascal supports version 3.2 or later of the Lotus/Intel/Microsoft Expanded Memory Specification (EMS) for this purpose. Once placed into EMS, the overlay file is closed, and subsequent overlay loads are reduced to fast in-memory transfers.

The overlay manager

Turbo Pascal's overlay manager is implemented by the *Overlay* standard unit. The buffer-management techniques used by the *Overlay* unit are very advanced, and always guarantee optimal performance in the available memory. For example, the overlay manager always keeps as many overlays as possible in the overlay buffer to reduce the chance of having to read an overlay from disk. Once an overlay is loaded, a call to one of its routines executes just as fast as a call to a nonoverlaid routine. Also, when the overlay manager needs to dispose of an overlay to make room for another, it attempts to first dispose of overlays that are inactive (ones that have no active routines at that time).

To implement its advanced overlay-management techniques, Turbo Pascal requires that you observe two important rules when writing overlaid programs:

- All overlaid units must include a **{$O+}** directive, which causes the compiler to ensure that the generated code can be overlaid.
- Whenever a call is made to an overlaid procedure or function, you must ensure that all currently active procedures and functions use the far call model.

Both rules are explained further in a section entitled "Designing overlaid programs," beginning on page 197. For now, just note that you can easily satisfy these requirements by placing a **{$O+,F+}** compiler directive at the beginning of all overlaid units, and a {**$F+**} compiler directive at the beginning of all other units and the main program.

☞ Failing to observe the far call requirement in an overlaid program causes unpredictable and possibly catastrophic results when the program is executed.

The {**$O** *unitname*} compiler directive is used in a program to indicate which units to overlay. This directive must be placed after the program's **uses** clause, and the **uses** clause must name the *Overlay* standard unit before any of the overlaid units. Here is an example:

```
program Editor;

{$F+}              { Force FAR calls for all procedures & functions }

uses
  Overlay, Crt, Dos, EdInOut, EdFormat, EdPrint, EdFind, EdMain;

{$O EdInOut}
{$O EdFormat}
{$O EdPrint}
{$O EdFind}
{$O EdMain}
```

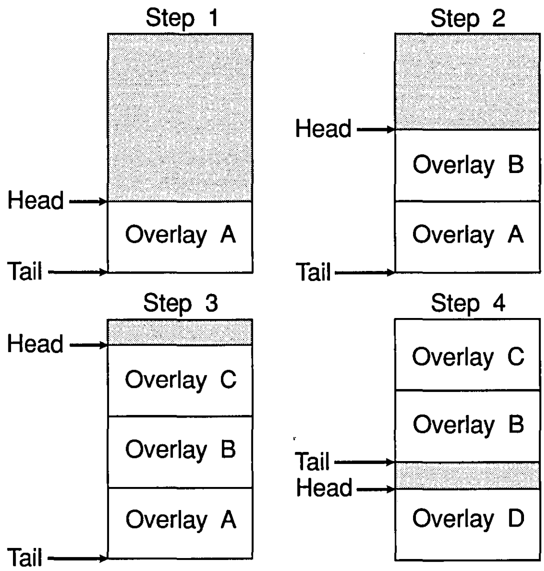
☞ The compiler reports an error if you attempt to overlay a unit that wasn't compiled in the {**$O+**} state. Of the standard units, the only one that can be overlaid is *Dos*; the other standard units, can't be overlaid. Also, programs containing overlaid units must be compiled to disk; the compiler reports an error if you attempt to compile such programs to memory.

Overlay buffer management

The Turbo Pascal overlay buffer is best described as a ring buffer that has a head pointer and a tail pointer. Overlays are always loaded at the head of the buffer, pushing "older" ones toward the tail. When the buffer becomes full (that is, when there isn't enough free space between the head and the tail), overlays are disposed of at the tail to make room for new ones.

Because ordinary memory isn't circular in nature, the actual implementation of the overlay buffer involves a few more steps to make the buffer appear to be a ring. Figure 18.1 illustrates the process. The figure shows a progression of overlays being loaded into an initially empty overlay buffer. Overlay *A* is loaded first, followed by *B*, then *C*, and finally *D*. Shaded areas indicate free buffer space.

Figure 18.1
Loading and disposing of overlays



As you can see, a couple of interesting things happen in the transition from step 3 to step 4. First, the head pointer wraps around to the bottom of the overlay buffer, causing the overlay manager to slide all loaded overlays (and the tail pointer) upward. This sliding is required to keep the area between the head pointer and the tail pointer free. Second, to load overlay *D*, the overlay manager has to dispose of overlay *A* from the tail of the buffer. Overlay *A* in this case is the least recently loaded overlay and, therefore, the best choice for disposal when something has to go. The overlay manager continues to dispose of overlays at the tail to make room for new ones at the head, and each time the head pointer wraps around, the sliding operation repeats.

Although this is the default mode of operation for Turbo Pascal's overlay manager, you can use an optional optimization of the overlay-management algorithm.

Imagine that overlay *A* contains a number of frequently used routines. Even though these routines are used all the time, *A* is still thrown out of the overlay buffer occasionally, only to be reloaded again shortly thereafter.

The overlay manager knows nothing about the *frequency* of calls to routines in *A*—only that a call is made to a routine in *A* and *A* isn't in memory, so it has to load *A*. One solution to this problem might be to trap every call to routines in *A* and then, at each call, move *A* to the head of the overlay buffer to reflect its new status as the most recently used overlay. Intercepting calls this way is very costly in terms of execution speed and, in some cases, can slow down the application even more than the additional overlay load operations does.

Turbo Pascal offers a solution that incurs almost no performance overhead and yet successfully identifies frequently used overlays that shouldn't be unloaded: When an overlay gets close to the tail of the overlay buffer, it's put on "probation."

If, during this probationary period, a call is made to a routine in the overlay, it's "reprieved," and isn't disposed of when it reaches the tail of the overlay buffer. Instead, it's moved to the head of the buffer and gets another free ride around the overlay buffer ring. On the other hand, if no calls are made to an overlay during its probationary period, thereby indicating infrequent use, the overlay is disposed of when it reaches the tail of the overlay buffer.

The net effect of the probation/reprieval scheme is that frequently used overlays are kept in the overlay buffer at the cost of intercepting just *one* call every time the overlay gets close to the tail of the overlay buffer.

Two overlay-manager routines, *OvrSetRetry* and *OvrGetRetry*, control the probation/reprieval mechanism. *OvrSetRetry* sets the size of the area in the overlay buffer to keep on probation and *OvrGetRetry* returns the current setting.

If an overlay falls within the last *OvrGetRetry* bytes before the overlay buffer tail, it's automatically put on probation. Any free space in the overlay buffer is considered part of the probation area.

Overlay procedures and functions

The *Overlay* unit defines a few procedures and functions; find their definitions as well as more details in Chapter 1, "Library reference," in the *Programmer's Reference.*

Table 18.1
Overlay unit procedures and functions

Procedure or function	Description
OvrClearBuf	Clears the overlay buffer.
OvrGetBuf	Returns the current size of the overlay buffer.
OvrGetRetry	Returns the current size of the probation area, the value last set with *OvrSetRetry*.
OvrInit	Initializes the overlay manager and opens the overlay file.
OvrInitEMS	Loads the overlay file into EMS.
OvrSetBuf	Sets the size of the overlay buffer.
OvrSetRetry	Sets the size of the "probation area" in the overlay buffer.

Variables and constants

The *Overlay* unit defines five variables:

Table 18.2
Overlay unit variables

Variable	Description
OvrFileMode	Determines the access code to pass to DOS when the overlay file is opened.
OvrLoadCount	The variable incremented each time an overlay is loaded.
OvrReadBuf	The procedure variable that lets you intercept overlay load operations.
OvrResult	The variable that holds the result code when an *Overlay* procedure executes.
OvrTrapCount	The variable incremented each time an overlaid routine is intercepted by the overlay manager.

Find the values of these variables in Chapter 1, "Library reference," in the *Programmer's Reference*.

Result codes

Errors in the *Overlay* unit are reported through the *OvrResult* variable. Look up "ovrXXXX constants" in Chapter 1, "Library reference," in the *Programmer's Reference* to find *OvrResult* values.

Designing overlaid programs

This section provides some important information on designing programs with overlays. Look it over carefully, because a number of the issues discussed are vital to well-behaved overlaid applications.

Overlay code generation

Turbo Pascal allows a unit to be overlaid only if it was compiled with {**$O+**}. In this state, the code generator takes special precautions when passing string and set constant parameters from one overlaid procedure or function to another. For example, if *UnitA* contains a procedure with the following header:

```
procedure WriteStr(S: string);
```

and if *UnitB* contains the statement

```
WriteStr('Hello world...');
```

then Turbo Pascal places the string constant 'Hello world...' in *UnitB*'s code segment, and passes a pointer to it to the *WriteStr* procedure. If both units are overlaid, this doesn't work because, at the call to *WriteStr*, *UnitB*'s code segment can be overwritten by *UnitA*'s and the string pointer becomes invalid. The {**$O+**} directive is used to avoid such problems; whenever Turbo Pascal detects a call from one unit compiled with {**$O+**} to another unit compiled with {**$O+**}, the compiler copies all code-segment-based constants into stack temporaries before passing pointers to them.

The use of {**$O+**} in a unit doesn't force you to overlay that unit. It just instructs Turbo Pascal to ensure that the unit can be overlaid, if so desired. If you develop units that you plan to use in overlaid as well as nonoverlaid applications, compiling them with {**$O+**} ensures that you can do both with just one version of the unit.

The far call requirement

At any call to an overlaid procedure or function in another module, you *must* guarantee that all currently active procedures or functions use the far call model.

This is best illustrated by example: Assume that *OvrA* is a procedure in an overlaid unit, and that *MainB* and *MainC* are procedures in the main program. If the main program calls

MainC, which calls *MainB*, which then calls *OvrA*, then at the call to *OvrA*, *MainB* and *MainC* are active (they have not yet returned) and they are required to use the far call model. Being declared in the main program, *MainB* and *MainC* would normally use the near call model. In this case, a {**$F+**} compiler directive must be used to force the far call model into effect.

The easiest way to satisfy the far call requirement is to place a {**$F+**} directive at the beginning of the main program and each unit. Alternatively, you can change the default **$F** setting to {**$F+**} using a **/$F+** command-line directive or the Force Far Calls check box in the Options | Compiler dialog box. Compared to the cost of mixing near and far calls, using far calls exclusively costs little—one extra word of stack space per active procedure and one extra byte per call.

Initializing the overlay manager

Here we'll take a look at some examples of how to initialize the overlay manager. Place the initialization code before the first call to an overlaid routine. Typically you would do this at the beginning of the program's statement part.

The following code shows just how little you need to initialize the overlay manager:

```
begin
  OvrInit('EDITOR.OVR');
end;
```

No error checks are made. If there isn't enough memory for the overlay buffer or if the overlay file was not found, run-time error 208 ("Overlay manager not installed") occurs when you attempt to call an overlaid routine.

Here's another simple example that expands on the previous one:

```
begin
  OvrInit('EDITOR.OVR');
  OvrInitEMS;
end;
```

In this case, provided there is enough memory for the overlay buffer and that the overlay file can be located, the overlay manager checks to see if EMS memory is available. If it is, it loads the overlay file into EMS.

The initial overlay buffer size is as small as possible, or in other words, just big enough to contain the largest overlay. This might be adequate for some applications, but imagine a situation where a particular function of a program is implemented through two or more units, each of which is overlaid. If the total size of those units is larger than the largest overlay, a substantial amount of swapping occurs if the units make frequent calls to each other.

The solution is to increase the size of the overlay buffer so that enough memory is available at any given time to contain all overlays that make frequent calls to each other. The following code demonstrates the use of *OvrSetBuf* to increase the overlay buffer size:

```
const
  OvrMaxSize = 80000;
begin
  OvrInit('EDITOR.OVR');
  OvrInitEMS;
  OvrSetBuf(OvrMaxSize);
end;
```

There is no general formula for determining the ideal overlay buffer size. Only an intimate knowledge of the application and a bit of experimenting results in a suitable value.

☞ Using *OvrInitEMS* to place the overlay file in EMS doesn't eliminate the need for an overlay buffer. Overlays must still be copied from EMS into "normal" memory in the overlay buffer before they can be executed, but because such in-memory transfers are significantly faster than disk reads, there is less need to increase the size of the overlay buffer.

Remember, *OvrSetBuf* expands the overlay buffer by shrinking the heap. Therefore, the heap must be empty or *OvrSetBuf* has no effect. If you are using the *Graph* unit, make sure you call *OvrSetBuf* *before* you call *InitGraph*, which allocates memory on the heap.

Here's a rather elaborate example of overlay-manager initialization with full error-checking:

```
const
  OvrMaxSize = 80000;
var
  OvrName: string[79];
  Size: Longint;
```

```
begin
  OvrName := 'EDITOR.OVR';
  repeat
    OvrInit(OvrName);
    if OvrResult = OvrNotFound then

    begin
      Writeln('Overlay file not found: ', OvrName, '.');
      Write('Enter correct overlay file name: ');
      Readln(OvrName);
    end;
  until OvrResult <> OvrNotFound;
  if OvrResult <> OvrOk then
  begin
    Writeln('Overlay-manager error.');
    Halt(1);
  end;
  OvrInitEMS;
  if OvrResult <> OvrOk then
  begin
    case OvrResult of
      ovrIOError: Write('Overlay file I/O error');
      ovrNoEMSDriver: Write('EMS driver not installed');
      ovrNoEMSMemory: Write('Not enough EMS memory');
    end;
    Write('. Press Enter...');
    Readln;
  end;
  OvrSetBuf(OvrMaxSize);
end;
```

First, if the default overlay file name isn't correct, the user is repeatedly prompted for a correct file name.

Next, a check is made for other errors that might have occurred during initialization. If an error is detected, the program halts because errors in *OvrInit* are fatal. (If they are ignored, a run-time error occurs upon the first call to an overlaid routine.)

Assuming successful initialization, a call to *OvrInitEMS* is made to load the overlay file into EMS if possible. In case of error, a diagnostic message is displayed, but the program isn't halted. Instead, it continues to read overlays from disk.

Finally, *OvrSetBuf* is called to set the overlay buffer size to a suitable value, determined through analysis and experimentation with the particular application. Errors from *OvrSetBuf* are ignored, although *OvrResult* might return an error code of –3 (*OvrNoMemory*). If there isn't enough memory, the overlay manager continues to use the minimum buffer that was allocated when the program started.

Initialization sections

Like static units, overlaid units can have an initialization section. Although overlaid initialization code is no different from normal overlaid code, the overlay manager must be initialized first so it can load and execute overlaid units.

Referring to the earlier *Editor* program, assume that the *EdInOut* and *EdMain* units have initialization code. This requires that *OvrInit* is called before *EdInOut's* initialization code. The only way to do that is to create an additional nonoverlaid unit that goes before *EdInOut* and calls *OvrInit* in its initialization section:

```
unit EdInit;
interface
implementation
uses Overlay;
const
  OvrMaxSize = 80000;
begin
  OvrInit('EDITOR.OVR');
  OvrInitEMS;
  OvrSetBuf(OvrMaxSize);
end.
```

The *EdInit* unit must be listed in the program's **uses** clause before any of the overlaid units:

```
program Editor;

{$F+}

uses Overlay, Crt, Dos, EdInit, EdInOut, EdFormat, EdPrint, EdFind,
     EdMain;

{$O EdInOut}
{$O EdFormat}
{$O EdPrint}
{$O EdFind}
{$O EdMain}
```

In general, although initialization code in overlaid units is indeed possible, you should avoid it for a number of reasons.

First, the initialization code, even though it's executed only once, is a part of the overlay, and occupies overlay-buffer space whenever the overlay is loaded. Second, if a number of overlaid units have initialization code, each of them has to be read into memory when the program starts.

A much better approach is to gather all the initialization code into an overlaid initialization unit, which is called once at the beginning of the program, and then never referenced again.

What not to overlay

Certain units can't be overlaid. In particular, don't try to overlay the following:

- Units compiled in the {**$O-**} state. The compiler reports an error if you attempt to overlay a unit that wasn't compiled with {**$O+**}. Such nonoverlay units include *System*, *Overlay*, *Crt*, *Graph*, *Turbo3*, and *Graph3*.
- Units that contain interrupt handlers. Due to the non-reentrant nature of the DOS operating system, units that implement **interrupt** procedures should not be overlaid. An example of such a unit is the *Crt* standard unit, which implements a *Ctrl+Break* interrupt handler.
- BGI drivers or fonts registered with calls to *RegisterBGIdriver* or *RegisterBGIfont*.

Calling overlaid routines via procedure pointers is fully supported by Turbo Pascal's overlay manager. Examples of the use of procedure pointers include exit procedures and text-file device drivers.

The overlay manager also supports passing overlaid procedures and functions as procedural parameters and assigning overlaid procedures and functions to procedural type variables.

Debugging overlays

Most debuggers have very limited overlay debugging capabilities, if any at all. This isn't so with Turbo Pascal and Turbo Debugger. The integrated debugger fully supports single-stepping and breakpoints in overlays in a manner completely transparent to you. By using overlays, you can easily engineer and debug huge applications—all from inside the IDE or by using Turbo Debugger.

External routines in overlays

Like normal Pascal procedures and functions, **external** assembly language routines must observe certain programming rules to work correctly with the overlay manager.

If an assembly language routine makes calls to *any* overlaid procedures or functions, the assembly language routine must use the far model, and it must set up a stack frame using the BP register. For example, assuming that *OtherProc* is an overlaid procedure in another unit, and that the assembly language routine *ExternProc* calls it, then *ExternProc* must use the FAR model and set up a stack frame. For example,

```
ExternProc      PROC    FAR

        PUSH    BP              ;Save BP
        MOV     BP,SP           ;Set up stack frame
        SUB     SP,LocalSize    ;Allocate local variables
          :
        CALL    OtherProc       ;Call another overlaid unit
          :
        MOV     SP,BP           ;Dispose of local variables
        POP     BP              ;Restore BP
        RET     ParamSize       ;Return

ExternProc      ENDP
```

LocalSize is the size of the local variables and *ParamSize* is the size of the parameters. If *LocalSize* is zero, you can omit the two lines to allocate and dispose of local variables.

These requirements are the same if *ExternProc* makes *indirect* references to overlaid procedures or functions. For example, if *OtherProc* makes calls to overlaid procedures or functions, but isn't itself overlaid, *ExternProc* must still use the FAR model and still has to set up a stack frame.

When an assembly language routine doesn't make any direct or indirect references to overlaid procedures or functions, there are no special requirements; the assembly language routine is free to use the near model and it doesn't have to set up a stack frame.

Overlaid assembly language routines should *not* create variables in the code segment, because any modifications made to an overlaid code segment are lost when the overlay is disposed of. Likewise, pointers to objects based in an overlaid code segment can't be expected to remain valid across calls to other overlays, because the overlay manager freely moves around and disposes of overlaid code segments.

Installing an overlay-read function

The *OvrReadBuf* procedure variable lets you intercept overlay load operations. For example, you can implement error handling or check that a removable disk is present. Whenever the overlay manager needs to read an overlay, it calls the function whose address is stored in *OvrReadBuf*. If the function returns zero, the overlay manager assumes that the operation was successful; if the function result is nonzero, the compiler generates run-time error 209. The *OvrSeg* parameter indicates what overlay to load, but as you'll see later, you won't need to access this information.

Don't attempt to call any overlaid routines from within your overlay-read function—such calls crash the system.

To install your own overlay-read function, you must first save the previous value of *OvrReadBuf* in a variable of type *OvrReadFunc*, and then assign your overlay-read function to *OvrReadBuf*. Within your read function, you should call the saved read function to perform the actual load operation. Any validations you want to perform, such as checking that a removable disk is present, should go before the call to the saved read function, and any error checking should go after the call.

The code to install an overlay-read function should go right after the call to *OvrInit*; at this point, *OvrReadBuf* contains the address of the default disk read function.

If you also call *OvrInitEMS*, it uses your read function to read overlays from disk into EMS memory, and if no errors occur, it stores the address of the default EMS read function in *OvrReadBuf*. If you also wish to override the EMS read function, simply repeat the installation process after the call to *OvrInitEMS*.

The default disk-read function returns zero if it succeeds, or a DOS error code if it fails. Likewise, the default EMS-read function returns 0 if it succeeds, or an EMS error code (ranging from $80 through $FF) if it fails. For details on DOS error codes, refer to the *Programmer's Reference*. For details on EMS error codes, refer to your Expanded Memory Specification documentation.

The following code fragment demonstrates how to write and install an overlay-read function. The new overlay-read function repeatedly calls the saved overlay-read function until no errors occur. Any errors are passed to the *DOSError* or *EMSError* procedures (not shown here) so that they can present the error to the user. Notice how the *OvrSeg* parameter is just passed on to the saved overlay-read function and never directly handled by the new overlay-read function.

```
uses Overlay;
var
  SaveOvrRead: OvrReadFunc;
  UsingEMS: Boolean;

function MyOvrRead(OvrSeg: Word): Integer; far;
var
  E: Integer;
begin
  repeat
    E := SaveOvrRead(OvrSeg);
    if E <> 0 then
      if UsingEMS then
        EMSError(E) else DOSError(E);
  until E = 0;
  MyOvrRead := 0;
end;

begin
  OvrInit('MYPROG.OVR');
  SaveOvrRead := OvrReadBuf;                        { Save disk default }
  OvrReadBuf := MyOvrRead;                               { Install ours }
  UsingEMS := False;
  OvrInitEMS;
  if OvrResult = OvrOK then
  begin
    SaveOvrRead := OvrReadBuf;                       { Save EMS default }
    OvrReadBuf := MyOvrRead;                             { Install ours }
    UsingEMS := True;
  end;
    :
end.
```

Overlays in .EXE files

Turbo Pascal allows you to store your overlays at the end of your application's .EXE file rather than in a separate .OVR file. To attach an .OVR file to the end of an .EXE file, use the DOS COPY command with a **/B** command-line switch, for example,

```
COPY/B MYPROG.EXE + MYPROG.OVR
```

You must make sure that the .EXE file was compiled *without* Turbo Debugger debug information. In the IDE, make sure the Standalone option isn't checked in Options | Debugger. With the command-line version of the compiler, don't specify a **/V** switch.

To read overlays from the end of an .EXE file instead of from a separate .OVR file, simply specify the .EXE file name in the call to *OvrInit*. If you are running under DOS 3.x or greater, you can use the *ParamStr* standard function to obtain the name of the .EXE file; for example,

```
OvrInit(ParamStr(0));
```

P A R T

3

Inside Turbo Pascal

C H A P T E R

19

Memory issues

This chapter describes in detail the ways Turbo Pascal programs use memory. We'll look at the memory map of a Turbo Pascal application, internal data formats, the heap manager, and direct memory access.

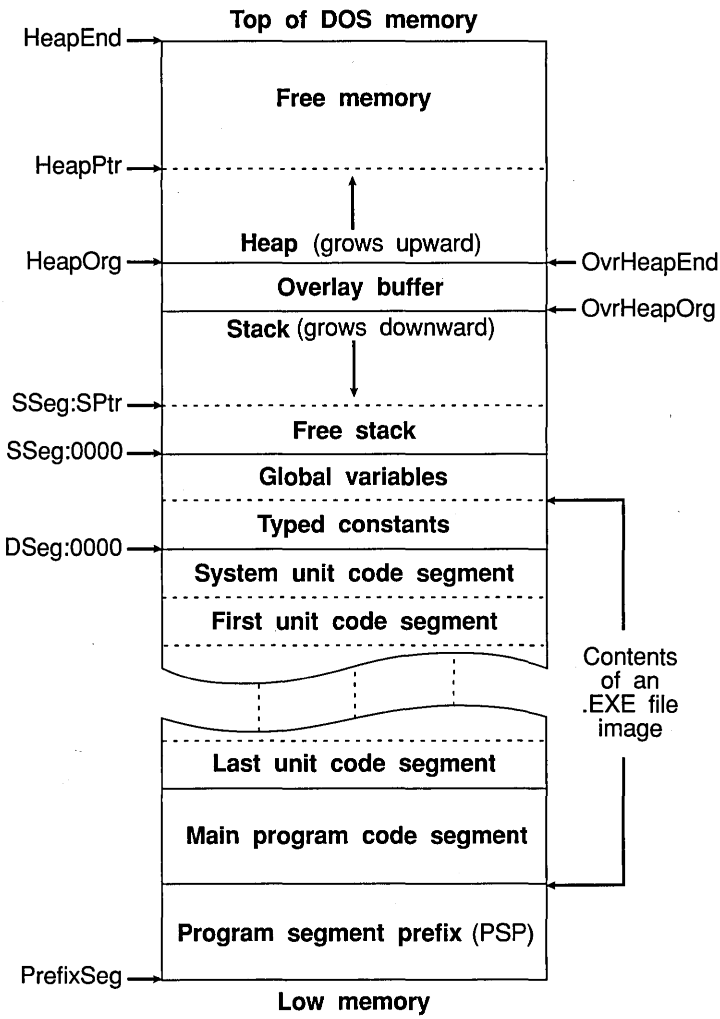
The Turbo Pascal memory map

Figure 19.1 depicts the memory map of a Turbo Pascal program.

The Program Segment Prefix (PSP) is a 256-byte area built by DOS when the .EXE file is loaded. The segment address of PSP is stored in the predeclared variable *PrefixSeg*.

Each module, which includes the main program and each unit, has its own code segment. The main program occupies the first code segment; the code segments that follow it are occupied by the units (in reverse order from how they are listed in the **uses** clause), and the last code segment is occupied by the *System* unit. 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.

Figure 19.1
Turbo Pascal memory map



The data segment (addressed through DS) contains all typed constants followed by all global variables. The DS register is never changed during program execution. The size of the data segment can't exceed 64K.

On entry to the program, the stack segment register (SS) and the stack pointer (SP) are loaded so that SS:SP points to the first byte past the stack segment. The SS register is never changed during program execution, but SP can move downward until it reaches the bottom of the segment. The size of the stack segment can't exceed 64K; the default size is 16K, but this can be changed with a **$M** compiler directive.

The *Overlay* standard unit uses the overlay buffer to store overlaid code. The default size of the overlay buffer corresponds to the size of the largest overlay in the program; if the program has no overlays, the size of the overlay buffer is zero. The size of the overlay buffer can be increased through a call to the *OvrSetBuf* routine in the *Overlay* unit; in that case, the size of the heap is decreased accordingly, by moving *HeapOrg* upwards.

The heap stores *dynamic variables*, that is, variables allocated through calls to the *New* and *GetMem* standard procedures. It occupies all or some of the free memory left when a program is executed. The actual size of the heap depends on the minimum and maximum heap values, which can be set with the **$M** compiler directive. Its size is guaranteed to be at least the minimum heap size and never more than the maximum heap size. If the minimum amount of memory isn't available, the program doesn't execute. The default heap minimum is 0 bytes, and the default heap maximum is 640K; this means that by default the heap occupies all remaining memory.

As you might expect, the heap manager (which is part of Turbo Pascal's run-time library) manages the heap. It's described in detail in the following section.

The heap manager

The heap is a stack-like structure that grows from low memory in the heap segment. The bottom of the heap is stored in the variable *HeapOrg*, and the top of the heap, corresponding to the bottom of free memory, is stored in the variable *HeapPtr*. Each time a dynamic variable is allocated on the heap (via *New* or *GetMem*), the heap manager moves *HeapPtr* upward by the size of the variable, in effect stacking the dynamic variables on top of each other.

HeapPtr is always normalized after each operation, forcing the offset part into the range $0000 to $000F. The maximum size of a single variable that can be allocated on the heap is 65,519 bytes (corresponding to $10000 minus $000F), because every variable must be completely contained in a single segment.

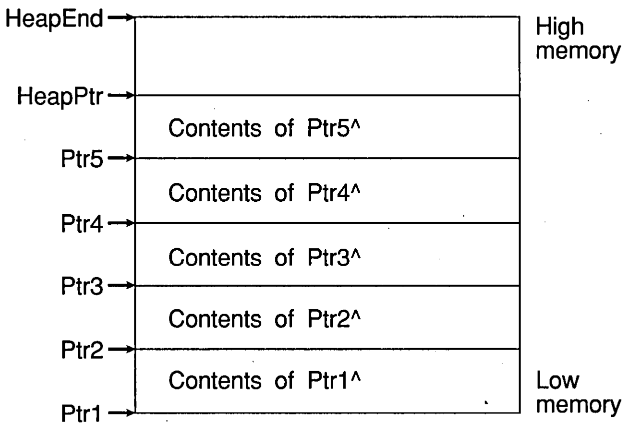
Disposal methods

The dynamic variables stored on the heap are disposed of in one of two ways: (1) through *Dispose* or *FreeMem* or (2) through *Mark* and *Release*. The simplest scheme is that of *Mark* and *Release*; for example, if the following statements are executed:

```
New(Ptr1);
New(Ptr2);
Mark(P);
New(Ptr3);
New(Ptr4);
New(Ptr5);
```

the layout of the heap will then look like this figure:

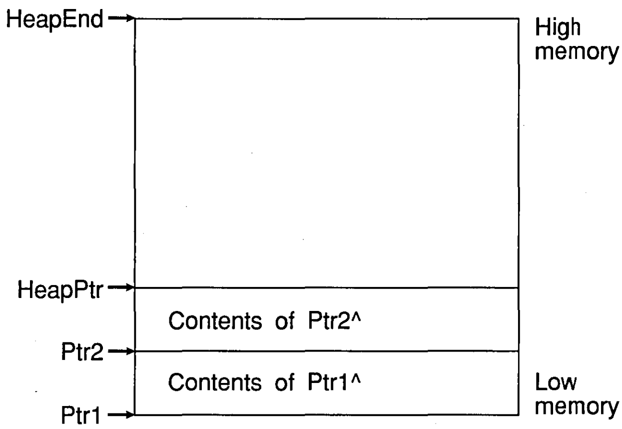
Figure 19.2
Disposal method using Mark and Release



Executing Release(HeapOrg) completely disposes of the entire heap because HeapOrg points to the bottom of the heap.

The *Mark(P)* statement marks the state of the heap just before *Ptr3* is allocated (by storing the current *HeapPtr* in *P*). If the statement *Release(P)* is executed, the heap layout becomes like that of Figure 19.3, effectively disposing of all pointers allocated since the call to *Mark*.

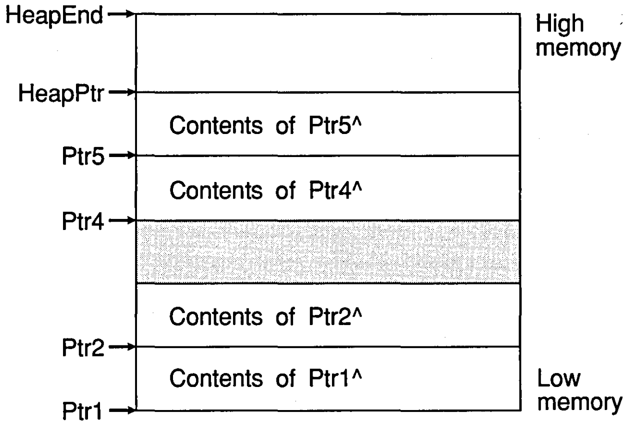
Figure 19.3
Heap layout with Release(P) executed



For applications that dispose of pointers in exactly the reverse order of allocation, the *Mark* and *Release* procedures are very efficient. Yet most programs tend to allocate and dispose of pointers in a more random manner, requiring the more sophisticated management technique implemented by *Dispose* and *FreeMem*. These procedures allow an application to dispose of any pointer at any time.

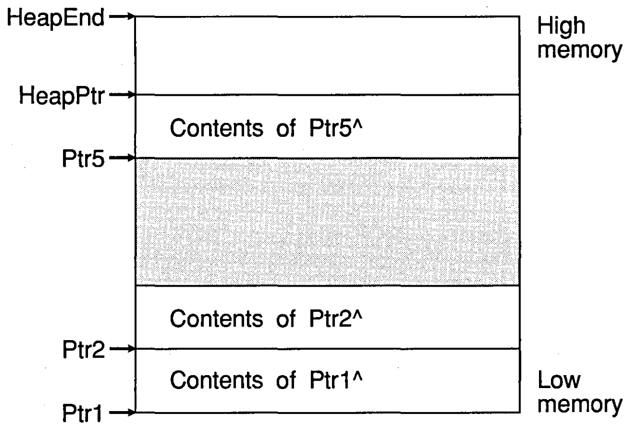
When a dynamic variable that isn't the topmost variable on the heap is disposed of through *Dispose* or *FreeMem*, the heap becomes fragmented. Assuming that the same statement sequence has been executed, then after executing *Dispose*(*Ptr3*), a "hole" is created in the middle of the heap (see Figure 19.4).

Figure 19.4
Creating a "hole" in the heap



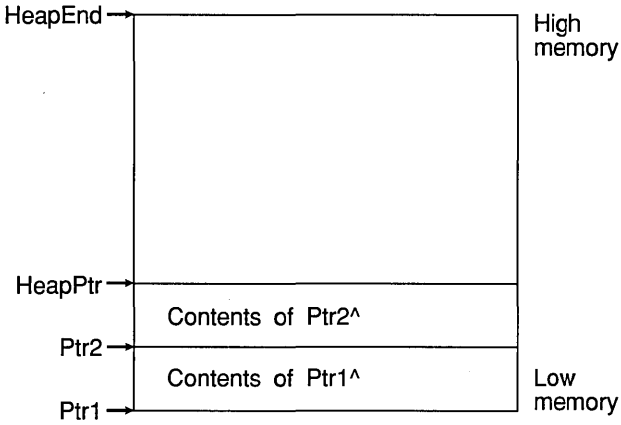
If *New*(*Ptr3*) had been executed now, it would again occupy the same memory area. On the other hand, executing *Dispose*(*Ptr4*) enlarges the free block, because *Ptr3* and *Ptr4* were neighboring blocks (see Figure 19.5).

Figure 19.5
Enlarging the free block



Finally, executing *Dispose*(*Ptr5*) first creates an even bigger free block, and then lowers *HeapPtr*. This, in effect, releases the free block, because the last valid pointer is now *Ptr2* (see Figure 19.6).

Figure 19.6
Releasing the free block



The heap is now in the same state as it would be after executing *Release*(*P*), as shown in Figure 19.3. The free blocks created and destroyed in the process were tracked for possible reuse, however.

The free list

The addresses and sizes of the free blocks generated by *Dispose* and *FreeMem* operations are kept on a *free list*. Whenever a dynamic variable is allocated, the free list is checked before the heap is expanded. If a free block of adequate size exists (it's greater than or equal to the size of the requested block size), it's used.

☞ The *Release* procedure always clears the free list, therefore causing the heap manager to "forget" about any free blocks that might exist below the heap pointer. If you mix calls to *Mark* and *Release* with calls to *Dispose* and *FreeMem*, you must ensure that no such free blocks exist.

The *FreeList* variable in the *System* unit points to the first free block in the heap. This block contains a pointer to the next free block, which contains a pointer to the following free block, and so on. The last free block contains a pointer to the top of the heap (that is, to the location given by *HeapPtr*). If there are no free blocks on the free list, *FreeList* will be equal to *HeapPtr*.

The format of the first eight bytes of a free block are given by the *TFreeRec* type as follows:

```
type
  PFreeRec = ^TFreeRec;
  TFreeRec = record
    Next: PFreeRec;
    Size: Pointer;
  end;
```

The *Next* field points to the next free block, or to the same location as *HeapPtr* if the block is the last free block. The *Size* field encodes the size of the free block. The value in *Size* isn't a normal 32-bit value; rather, it's a "normalized" pointer value with a count of free paragraphs (16-byte blocks) in the high word, and a count of free bytes (between 0 and 15) in the low word. The following *BlockSize* function converts a *Size* field value to a normal *Longint* value:

```
function BlockSize(Size: Pointer): Longint;
type
  PtrRec = record Lo, Hi: Word end;
begin
  BlockSize := Longint(PtrRec(Size).Hi) * 16 + PtrRec(Size).Lo;
end;
```

To guarantee that there will always be room for a *TFreeRec* at the beginning of a free block, the heap manager rounds the size of *every* block allocated by *New* or *GetMem* upwards to an 8-byte boundary. Eight bytes are allocated for blocks of size 1..8, 16 bytes are allocated for blocks of size 9..16, and so on. This might seem an excessive waste of memory at first, and it would be if every block was just 1 byte in size. Blocks are usually larger, however, and so the relative size of the unused space is less.

The 8-byte granularity factor ensures that a number of random allocations and deallocations of blocks of varying small sizes, such as would be typical for variable-length line records in a text-processing program, don't heavily fragment the heap. For example, say a 50-byte block is allocated and disposed of, thereby becoming an entry on the free list. The block would have been rounded to 56 bytes (7*8), and a later request to allocate anywhere from 49 to 56 bytes would completely reuse the block, instead of leaving 1 to 7 bytes of free (but most likely unusable) space, which would fragment the heap.

The HeapError variable

The *HeapError* variable allows you to install a heap-error function, which is called whenever the heap manager can't complete an allocation request. *HeapError* is a pointer that points to a function with the following header:

```
function HeapFunc(Size: Word): Integer; far;
```

Note that the **far** directive forces the heap-error function to use the FAR call model.

The heap-error function is installed by assigning its address to the *HeapError* variable:

```
HeapError := @HeapFunc;
```

The heap-error function is called whenever a call to *New* or *GetMem* can't complete the request. The *Size* parameter contains the size of the block that couldn't be allocated, and the heap-error function should attempt to free a block of at least that size.

Depending on its success, the heap-error function should return 0, 1, or 2. A return of 0 indicates failure, causing a run-time error to occur immediately. A return of 1 also indicates failure, but instead of a run-time error, it causes *New* or *GetMem* to return a **nil** pointer. Finally, a return of 2 indicates success and causes a retry (which could also cause another call to the heap-error function).

The standard heap-error function always returns 0 and causes a run-time error whenever a call to *New* or *GetMem* can't be completed. For many applications, however, the simple heap-error function that follows is more appropriate:

```
function HeapFunc(Size: Word): Integer; far;
begin
  HeapFunc := 1;
end;
```

When installed, this function causes *New* or *GetMem* to return **nil** when they can't complete the request, instead of aborting the program.

☞ A call to the heap-error function with a *Size* parameter of 0 means that the heap manager has just expanded the heap by moving *HeapPtr* upwards. This occurs whenever there are no free blocks on the free list, or when all free blocks are too small for the allocation request. Such a call doesn't indicate an error condition, because there was still adequate room for expansion between *HeapPtr* and *HeapEnd*. Instead, the call indicates that the unused area above *HeapPtr* has shrunk, and the heap manager ignores the return value.

Internal data formats

The next several pages discuss the internal data formats of Turbo 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 (*Integer*), 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* type is stored as a *Byte*, a *WordBool* type is stored as a *Word*, and a *LongBool* type is stored as a *Longint*.

A *Boolean* type 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 256 or fewer values; otherwise, it's 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

$$+/- \text{ significand} \times 2^{\text{exponent}}$$

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:

width in bits

1	39	8
s	f	e
msb	lsb	msb lsb

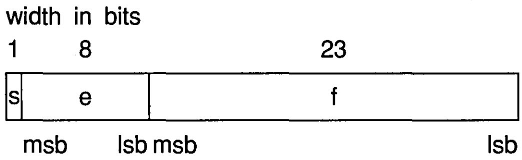
The value *v* of the number is determined by the following:

$$\textbf{if } 0 < e <= 255, \textbf{ then } v = (-1)^{s} * 2^{(e-129)} * (1.f).$$
$$\textbf{if } e = 0, \textbf{ then } v = 0.$$

The *Real* type can't store denormals, NaNs, or 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:

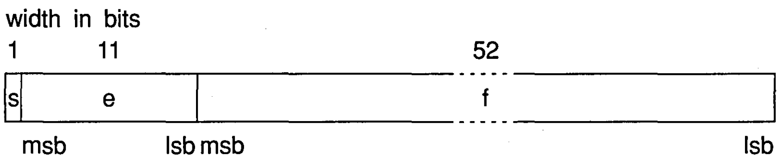


The value *v* of the number is determined by the following:

```
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:

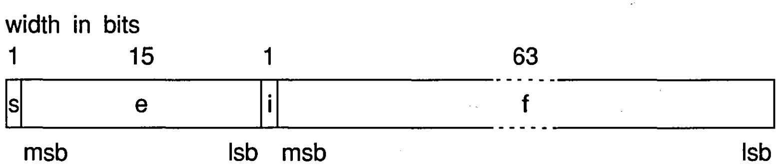


The value *v* of the number is determined by the following:

```
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:

width in bits

1	63
s	d
msb	lsb

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.

Object types

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 sequence of variables. Any fields inherited from an ancestor type are stored before the new fields defined in the descendant type.

If an object type defines virtual methods, constructors, or destructors, the compiler allocates an extra field in the object type. This 16-bit field, called the *virtual method table (VMT) field,* is used to store the offset of the object type's VMT in the data segment. The VMT field immediately follows after the ordinary fields in the object type. When an object type inherits virtual methods, constructors, or destructors, it also inherits a VMT field, so an additional one isn't allocated.

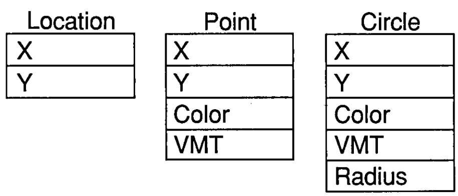
Initialization of the VMT field of an instance is handled by the object type's constructor(s). A program never explicitly initializes or accesses the VMT field.

The following examples illustrate the internal data formats of object types:

```
type
  PLocation = ^TLocation;
  TLocation = object
    X, Y: Integer;
    procedure Init(PX, PY: Integer);
    function GetX: Integer;
    function GetY: Integer;
  end;

  PPoint = ^TPoint;
  TPoint = object(TLocation)
    Color: Integer;
    constructor Init(PX, PY, PColor: Integer);
    destructor Done; virtual;
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure MoveTo(PX, PY: Integer); virtual;
  end;

  PCircle = ^TCircle;
  TCircle = object(TPoint)
    Radius: Integer;
    constructor Init(PX, PY, PColor, PRadius: Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Fill; virtual;
  end.
```

Figure 19.7 shows layouts of instances of *TLocation*, *TPoint*, and *TCircle*; each box corresponds to one word of storage.

Figure 19.7
Layouts of instances of TLocation, TPoint, and TCircle



Virtual method tables

Each object type that contains or inherits virtual methods, constructors, or destructors has a VMT associated with it, which is stored in the initialized part of the program's data segment. There is only one VMT per object type (not one per instance), but two distinct object 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 object type instances by the object type's constructor(s) and are never directly manipulated by a program.

The first word of a VMT contains the size of instances of the associated object type; this information is used by constructors and destructors to determine how many bytes to allocate or dispose of, using the extended syntax of the *New* and *Dispose* standard procedures.

The second word of a VMT contains the negative size of instances of the associated object type; this information is used by the virtual method call validation mechanism to detect uninitialized objects (instances for which no constructor call has been made), and to check the consistency of the VMT. When virtual call validation is enabled (using the {**$R+**} compiler directive, which has been expanded to include virtual method checking), the compiler generates a call to a VMT validation routine before each virtual call. The VMT validation routine checks that the first word of the VMT isn't zero, and that the sum of the first and the second word is zero. If either check fails, the compiler generates run-time error 210.

☞ Enabling range checking and virtual method call checking slows down your program and makes it somewhat larger, so use the {**R+**} state only when debugging, and switch to the {**$R-**} state for the final version of the program.

See page 225 for an explanation of dynamic method tables.

The third word of a VMT contains the data segment offset of the object type's dynamic method table (DMT), or zero if the object type has no dynamic methods.

The fourth word of a VMT is reserved and always contains zero.

Finally, starting at offset 8 in the VMT, is a list of 32-bit method pointers, one per virtual method in the object type, in order of declaration. Each slot contains the address of the corresponding virtual method's entry point.

Figure 19.8 shows the layouts of the VMTs of the *TPoint* and *TCircle* types; each small box corresponds to one word of storage, and each large box corresponds to two words of storage.

Figure 19.8
TPoint and TCircle's VMT layouts

TPoint VMT	TCircle VMT
8	10
-8	-10
0	0
0	0
@TPoint.Done	@TPoint.Done
@TPoint.Show	@TCircle.Show
@TPoint.Hide	@TCircle.Hide
@TPoint.MoveTo	@TPoint.MoveTo
	@TCircle.Fill

Notice how *TCircle* inherits the *Done* and *MoveTo* methods from *TPoint*, and how it overrides the *Show* and *Hide* methods.

As mentioned already, an object type's constructors contain special code that stores the offset of the object type's VMT in the instance being initialized. For example, given an instance *P* of type *Pointer*, and an instance *C* of type *TCircle*, a call to *P.Init* automatically stores the offset of *TPoint*'s VMT in *P*'s VMT field, and a call to *C.Init* likewise stores the offset of *TCircle*'s VMT in *C*'s VMT field. This automatic initialization is part of a constructor's entry code, so when control arrives at the **begin** of the constructor's statement part, the VMT field *Self* is already set up. Therefore, if the need arises, a constructor can make calls to virtual methods.

Dynamic method tables

The VMT for an object type contains a four-byte entry (a method pointer) for each virtual method declared in the object type and any of its ancestors. In cases where ancestral type(s) define a large number of virtual methods, the process of creating derived types can use up quite a lot of memory, especially if many derived types are created. Even though the derived types can override only a few of the inherited methods, the VMT of each derived type contains method pointers for all inherited virtual methods, even if they haven't changed.

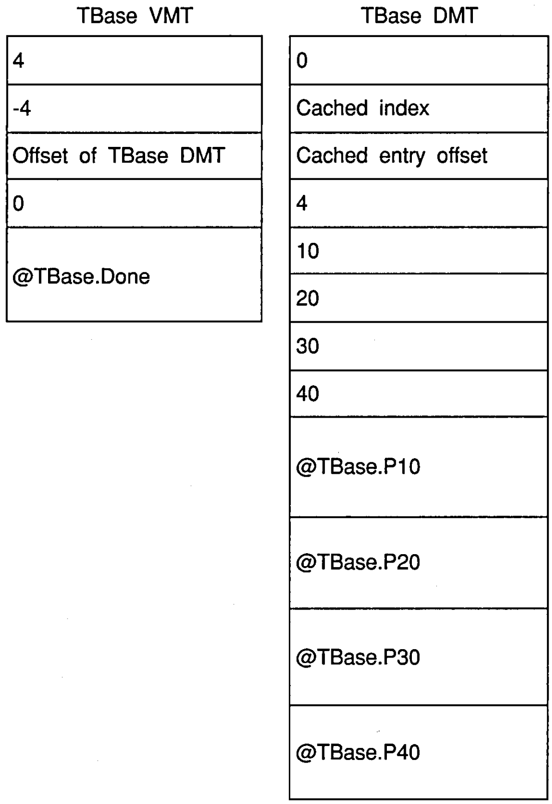
Dynamic methods provide an alternative in such situations. Instead of encoding a pointer for *all* late-bound methods in an object type, a dynamic method table (DMT) encodes only the methods that were *overridden* in the object type. When descendant types override only a few of a large number of inherited late-bound methods, the dynamic method table format uses less space than the format used by VMTs.

The following two object types illustrate DMT formats:

```
type
  TBase = object
    X: Integer;
    constructor Init;
    destructor Done; virtual;
    procedure P10; virtual 10;
    procedure P20; virtual 20;
    procedure P30; virtual 30;
    procedure P40; virtual 40;
  end;

type
  TDerived = object(TBase)
    Y: Integer;
    constructor Init;
    destructor Done; virtual;
    procedure P10; virtual 10;
    procedure P30; virtual 30;
    procedure P50; virtual 50;
  end;
```

Figures 19.9 and 19.10 shows the layouts of the VMTs and DMTs of *TBase* and *TDerived*. Each small box corresponds to one word of storage, and each large box corresponds to two words of storage.

Figure 19.9
TBase's VMT and DMT layouts



An object type has a DMT only if it introduces or overrides dynamic methods. If an object type inherits dynamic methods, but doesn't override any of them or introduce new ones, it simply inherits the DMT of its ancestor.

As is the case for VMTs, DMTs are stored in the initialized part of the application's data segment.

Figure 19.10
TDerived's VMT and DMT layouts

TDerived VMT
6
-6
Offset of TDerived DMT
0
@TDerived.Done

TDerived DMT
Offset of TBase DMT
Cached index
Cached entry offset
3
10
30
50
@TDerived.P10
@TDerived.P30
@TDerived.P50

The first word of a DMT contains the data segment offset of the *parent DMT*, or zero if there is no parent DMT.

The second and third words of a DMT are used to cache dynamic method lookups, as is described on page 239.

The fourth word of a DMT contains the *DMT entry count*. It's immediately followed by a list of words, each of which contain a dynamic method index, and then followed by a list of corresponding method pointers. The length of each list is given by the DMT entry count.

File types

File types are represented as records. Typed files and untyped files occupy 128 bytes, which are laid out in the *Dos* unit as follows:

```
type
  FileRec = 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
  TextBuf = array[0..127] of Char;
  TextRec = record
    Handle: Word;
    Mode: Word;
    BufSize: Word;
    Private: Word;
    BufPos: Word;
    BufEnd: Word;
    BufPtr: ^TextBuf;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..16] of Byte;
    Name: array[0..79] of Char;
    Buffer: TextBuf;
  end;
```

Handle contains the file's handle (when the file is open) as returned by DOS.

The *Mode* field can assume one of the following "magic" 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 Turbo 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 procedural type is stored as a double word, with the offset part of the referenced procedure in the low word and the segment part in the high word.

Direct memory access

Turbo 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 some examples:

```
Mem[$0040:$0049] := 7;
Data := MemW[Seg(V):Ofs(V)];
MemLong := MemL[64:3*4];
```

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*. The third statement moves the *Longint* value stored at $0040:$000C into the variable *MemLong.*

Direct port access

For access to the 80x86 CPU data ports, Turbo 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.

C H A P T E R

20

Control issues

This chapter describes in detail the various ways that Turbo Pascal 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 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 doesn't 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's 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 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.

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 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.

NEAR and FAR calls

The 80x86 family of CPUs support 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.

Turbo Pascal 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 be 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 {**$F-**} state, Turbo Pascal automatically selects the correct model. The default state is {**$F-**}.

Nested procedures and functions

A procedure or function is said to be nested when it's 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. In an overlaid application, however, a {**$F+**}

directive is generally used to force all procedures and functions to be far, including those that are nested.

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:

Nested procedures and functions can't be declared with the external directive, and they can't be procedural parameters.

```
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;
```

Method calling conventions

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter, *Self*, that corresponds to a **var** parameter of the same type as the method's object type. The *Self* parameter is always passed as the last parameter, and always takes the form of a 32-bit pointer to the instance through which the method is called. For example, given a variable *PP* of type *PPoint* as defined on page 223, the call *PP^.MoveTo*(10, 20) is coded as follows:

```
MOV     AX,10               ;Load 10 into AX
PUSH    AX                  ;Pass as PX parameter
MOV     AX,20               ;Load 20 into AX
PUSH    AX                  ;Pass as PY parameter
LES     DI,PP               ;Load PP into ES:DI
PUSH    ES                  ;Pass as Self parameter
PUSH    DI
MOV     DI,ES:[DI+6]        ;Pick up VMT offset from VMT field
CALL    DWORD PTR [DI+20]   ;Call VMT entry for MoveTo
```

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.

Virtual method calls

To call a virtual method, the compiler generates code that picks up the VMT address from the VMT field in the object, and then calls via the slot associated with the method. For example, given a variable *PP* of type *Point* (see page 223), the call *PP^.Show* generates the following code:

```
LES     DI,PP               ;Load PP into ES:DI
PUSH    ES                  ;Pass as Self parameter
PUSH    DI
MOV     DI,ES:[DI+6]        ;Pick up VMT offset from VMT field
CALL    DWORD PTR [DI+12]   ;Call VMT entry for Show
```

The type compatibility rules of object types allow *PP* to point at a *Point* or a *TCircle*, or at any other descendant of *TPoint*. And if you examine the VMTs shown on page 225, you'll see that for a *TPoint*, the entry at offset 12 in the VMT points to *TPoint.Show*; whereas for a *TCircle*, it points to *TCircle.Show*. Therefore, depending upon the *actual* run-time type of *PP*, the CALL instruction calls *TPoint.Show* or *TCircle.Show*, or the *Show* method of any other descendant of *TPoint*.

If *Show* had been a static method, the compiler would have generated this for the call to *PP^.Show*:

```
LES     DI,PP              ;Load PP into ES:DI
PUSH    ES                 ;Pass as Self parameter
PUSH    DI
CALL    TPoint.Show        ;Directly call TPoint.Show
```

Here, no matter what *PP* points to, the code always calls the *TPoint.Show* method.

Dynamic method calls

Dispatching a dynamic method call is somewhat more complicated and time consuming than dispatching a virtual method call. Instead of using a CALL instruction to call through a method pointer at a static offset in the VMT, the object type's DMT and parent DMTs must be *scanned* to find the topmost occurrence of a particular dynamic method index, and then a call must be made through the corresponding method pointer. This process involves far more instructions than can be coded in-line, so the Turbo Pascal run-time library (RTL) contains a dispatch-support routine that is used when making dynamic method calls.

Had the *Show* method of the preceding type *TPoint* been declared as a dynamic method (with a dynamic method index of 200), the call *PP^.Show*, where *PP* is of type *Point*, would generate the following code:

```
LES     DI,PP              ;Load PP into ES:DI
PUSH    ES                 ;Pass as Self parameter
PUSH    DI
MOV     DI,EX:[DI+6]       ;Pick up VMT offset from VMT field
MOV     AX,200             ;Load dynamic method index into AX
CALL    Dispatch           ;Call RTL routine to dispatch call
```

The RTL dispatcher first picks up the DMT offset from the VMT pointed to by the DI register. Then, using the "cached index" field of the DMT, the dispatcher checks if the dynamic method index of the method being called is the same as the last one that was called. If so, it immediately transfers control to the method, by jumping indirectly through the method pointer stored at the offset given by the "cached entry offset" field.

If the dynamic index of the method being called isn't the same as the one stored in the cache, the dispatcher scans the DMT and the parent DMTs (by following the parent links in the DMTs) until it locates an entry with the given dynamic method index. The index and the offset of the corresponding method pointer is then stored in the DMT's cache fields, and control is transferred to the method. If, for some reason, the dispatcher can't find an entry with the given dynamic method index, indicating that the DMTs have somehow been destroyed, it terminates the application with a run-time error 210.

In spite of caching and a highly optimized RTL dispatch support routine, the dispatching of a dynamic method call takes substantially longer than a virtual method call. When the actions performed by the dynamic methods themselves take up a lot of time, however, the amount of space saved by using DMTs might outweigh this penalty.

Constructors and destructors

Constructors and destructors use the same calling conventions as other methods, except that an additional word-sized parameter, called the *VMT* parameter, is passed on the stack just before the *Self* parameter.

For constructors, the VMT parameter contains the VMT offset to store in *Self*'s VMT field to initialize *Self*.

When a constructor is called to allocate a dynamic object using the extended syntax of the *New* standard procedure, a **nil** pointer is passed in the *Self* parameter. The constructor allocates a new dynamic object, the address of which is passed back to the caller in DX:AX when the constructor returns. If the constructor can't allocate the object, a **nil** pointer is returned in DX:AX.

See "Constructor error recovery" on page 106.

Finally, when a constructor is called using a qualified-method identifier (that is, an object type identifier, followed by a period and a method identifier), a value of zero is passed in the VMT parameter. This indicates to the constructor that it should *not* initialize the VMT field of *Self*.

For destructors, a 0 in the VMT parameter indicates a normal call, and a nonzero value indicates that the destructor was called using the extended syntax of the *Dispose* standard procedure. This causes the destructor to deallocate *Self* just before returning (the size of *Self* is found by looking at the first word of *Self*'s VMT).

Entry and exit code

Each Pascal procedure and function begins and ends with standard entry and exit code.

This is the standard entry code:

```
PUSH    BP              ;Save BP
MOV     BP,SP           ;Set up stack frame
SUB     SP,LocalSize    ;Allocate locals (if any)
```

LocalSize is the size of the local variables. The SUB instruction is present only if *LocalSize* isn't 0. If the procedure's call model is near, the parameters start at BP + 4; if it's far, they start at BP + 6.

This is the standard exit code:

```
MOV     SP,BP           ;Deallocate locals (if any)
POP     BP              ;Restore BP
RET     ParamSize       ;Remove parameters and return
```

ParamSize is the size of the parameters. The RET instruction is either a near or far return, depending on the routine's call model.

Register-saving conventions

Procedures and functions should preserve the BP, SP, SS, and DS registers. All other registers can be modified.

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 as 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 program demonstrates a skeleton method of implementing an exit procedure:

```
program Testexit;
var
  ExitSave: Pointer;

procedure MyExit; far;
begin
  ExitProc := ExitSave;            { Always restore old vector first }
    :
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
    :
end.
```

On entry, the program 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* isn't **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 isn't **nil**. In addition, before returning, make sure to set *ErrorAddr* to **nil**, so that the error isn't reported again by other exit procedures.

Once the run-time library has called all exit procedures, it returns to DOS, passing the value stored in *ExitCode* as a return code.

Interrupt handling

The Turbo Pascal run-time library and the code generated by the compiler are fully interruptible. Also, most of the run-time library is reentrant, which allows you to write interrupt service routines in Turbo Pascal.

Writing interrupt procedures

Declare **interrupt** procedures with the **interrupt** directive. Every **interrupt** procedure must specify the following procedure header (or a subset of it, as explained later):

```
procedure IntHandler(Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES,
  BP: Word);
interrupt;
begin
   ⋮
end;
```

As you can see, all the registers are passed as pseudoparameters so you can use and modify them in your code. You can omit some or all of the parameters, starting with *Flags* and moving towards *BP*. It's an error to declare more parameters than are listed in the preceding example, or to omit a specific parameter without also omitting the ones before it (although no error is reported). For example,

```
procedure IntHandler(DI, ES, BP: Word);             { Invalid header }
procedure IntHandler(SI, DI, DS, ES, BP: Word);       { Valid header }
```

On entry, an **interrupt** procedure automatically saves all registers (regardless of the procedure header) and initializes the DS register:

```
PUSH    AX
PUSH    BX
PUSH    CX
PUSH    DX
PUSH    SI
PUSH    DI
PUSH    DS
PUSH    ES
PUSH    BP
MOV     BP,SP
SUB     SP,LocalSize
MOV     AX,SEG DATA
MOV     DS,AX
```

Notice the lack of a STI instruction to enable additional interrupts. You should code this yourself (if required) using an **inline** statement. The exit code restores the registers and executes an interrupt-return instruction:

```
MOV     SP, BP
POP     BP
POP     ES
POP     DS
POP     DI
POP     SI
POP     DX
POP     CX
POP     BX
POP     AX
IRET
```

An **interrupt** procedure can modify its parameters. Changing the declared parameters will modify the corresponding register when the interrupt handler returns. This can be useful when you are using an interrupt handler as a user service, much like the DOS INT 21H services.

Interrupt procedures that handle hardware-generated interrupts should not use any of Turbo Pascal's input and output or dynamic memory allocation routines, because they aren't reentrant. Likewise, no DOS functions can be used because DOS isn't reentrant.

C H A P T E R

21

Optimizing your code

Turbo Pascal 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, Turbo Pascal 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

Turbo Pascal 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

Read more about constant parameters on page 109.

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.

Redundant pointer-load elimination

In certain situations, Turbo Pascal'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 additional 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, this statement:

```
if ((Ch >= 'A') and (Ch <= 'Z')) or
  ((Ch >= 'a') and (Ch <= 'z')) then ... ;
```

is less readable and also less efficient than this:

```
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, Turbo Pascal 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.

For more details, refer to Chapter 2, "Compiler directives," in the Programmer's Reference.

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 any code:

```
if False then
  statement
while False do
  statement
```

Smart linking

When compiling to memory, Turbo Pascal's smart linker is disabled. This explains why some programs become smaller when compiled to disk.

Turbo Pascal'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 := 'Turbo 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.

Smart linking is especially valuable in connection with units that implement procedure/function libraries. An example of such a unit is the *Dos* 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 *Dos*, 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.

P A R T

4

Using Turbo Pascal with assembly language

C H A P T E R

22

The built-in assembler

Turbo Pascal's built-in assembler allows you to write 8086/8087 and 80286/80287 assembler code directly inside your 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 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 Turbo Pascal constructs. For example, most EQU directives correspond to **const**, **var**, and **type** declarations in Turbo Pascal, the PROC directive corresponds to **procedure** and **function** declarations, and the STRUC directive corresponds to Turbo Pascal **record** types. In fact, Turbo Pascal's built-in assembler can be thought of as an assembler language compiler that uses 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
```

AsmStatement is an assembler statement and *Separator* is a semicolon, a new-line, or a 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 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

Only the first 32 characters of an identifier are significant in the built-in assembler.

Labels are defined in assembler as they are in Pascal—by writing a label identifier and a colon before a statement. And as they are in 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 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 generate a near jump and a far jump, respectively, even if *Stop* is a label within reach of a short jump.

Assembler directives

Turbo Pascal'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 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,"Turbo Pascal"                     { 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 Turbo Pascal, the only kind of symbol that can be defined in an built-in assembler statement is a label. All variables must be declared using 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 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 22.1
Built-in assembler reserved words

AH	CS	LOW	SI
AL	CX	MOD	SP
AND	DH	NEAR	SS
AX	DI	NOT	ST
BH	DL	OFFSET	TBYTE
BL	DS	OR	TYPE
BP	DWORD	PTR	WORD
BX	DX	QWORD	XOR
BYTE	ES	SEG	
CH	FAR	SHL	
CL	HIGH	SHR	

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 Pascal and Assembler expressions

The most important difference between 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 Pascal and built-in assembler expressions is the way variables are interpreted. In a 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 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 rather it loads the value of a word stored four bytes beyond *X*. The correct way to add 4 to the contents of *X* is:

```
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 Pascal expressions. 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'
'Turbo Pascal'
"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 22.2
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 22.3
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 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* would use the *@Result* variable if it was 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

Table 22.4 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 22.4
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 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 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; 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 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;
```

all of which 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;
```

Table 22.5 summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Pascal types.

Table 22.5
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. Table 22.6 lists the built-in assembler's expression operators in decreasing order of precedence.

Table 22.6
Summary of built-in asssembler expression operators

Built-in assembler operator precedence is different from Pascal. For example, in a built-in assembler expression, the AND operator has lower precedence than the plus (+) and minus (-) operators, whereas in a Pascal expression, it has higher precedence.

Operator(s)	Comments
&	Identifier override operator
(), [], .	Structure member selector
HIGH, LOW	
+, –	Unary operators
:	Segment override operator
OFFSET, SEG, TYPE, PTR, *, /, MOD, SHL, SHR	
+, –	Binary addition/ subtraction operators
NOT, AND, OR, XOR	Bitwise operators

Table 22.7: 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.

Table 22.7: Definitions of built-in assembler expression operators (continued)

/	**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. Turbo Pascal'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 Turbo Pascal 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 in many ways 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 ordinary functions and **assembler** functions. The first example uses an **asm** statement in an ordinary 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;
```

C H A P T E R

23

Linking assembler code

Procedures and functions written in assembly language can be linked with Turbo Pascal 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 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 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 Pascal program or unit. However, they reside in the same segment as the Pascal globals, and can be accessed through the DS segment register.

All procedures, functions, and variables declared in the 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, Turbo Pascal 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.
- Turbo Pascal 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 Turbo Pascal

Turbo Assembler (TASM) makes it easy to program routines in assembly language and interface them into your Turbo Pascal programs. Turbo Assembler provides simplified segmentation and language support for Pascal programmers.

The .MODEL directive specifies the memory model for an assembler module that uses simplified segmentation. For linking with 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 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 Pascal function definition would look like this:

```
function MyProc(I, J: Char): string; external;
```

For more information about interfacing Turbo Assembler with Turbo Pascal, 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 STR5
TPCW stringer
```

Assembly language methods

Method implementations written in assembly language can be linked with Turbo Pascal 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 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

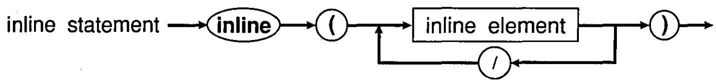
For very short assembly language subroutines, Turbo Pascal's **inline** statements and directives are very convenient. They let you insert machine code instructions directly into the program or unit text instead of through an object file.

Inline statements

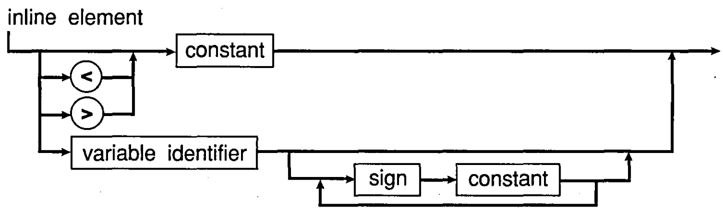
An **inline** statement consists of the reserved word **inline** followed by one or more inline elements, separated by slashes and enclosed in parentheses:

```
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.



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.

Registers BP, SP, SS, and DS must be preserved by inline statements; all other registers can be modified.

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.

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.

Use **inline** directives 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.

I N D E X

D

E

F

H

I

M

N

O

P

Q

R

S

T

U

V

W

X

Z

7.0

TURBO PASCAL®

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-8400. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom ▪ Part #11MN-TPL04-70 ▪ BOR 4680