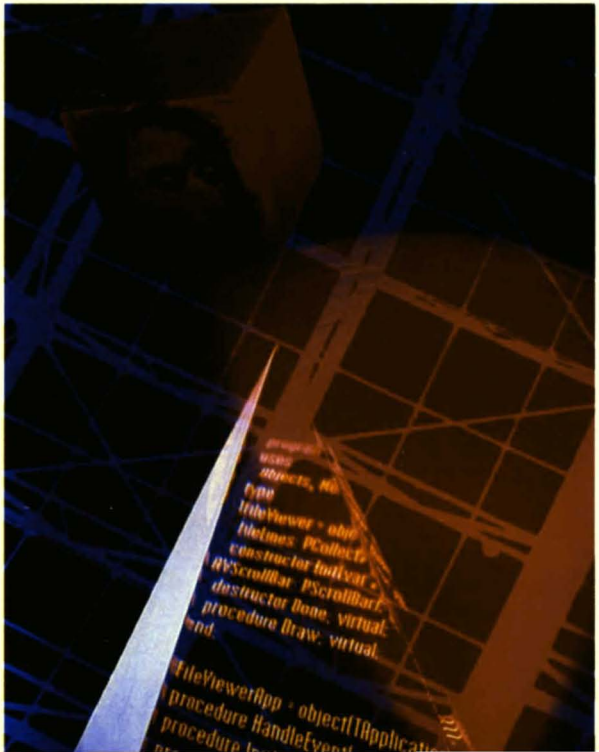
TURBO PASCAL®

6.0

PROGRAMMER'S GUIDE



BORLAND

Turbo Pascal®

Version 6.0

Programmer's Guide

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

PRINTED IN THE USA.
R2 10 9 8 7 6 5 4 3 2 1

C O N T E N T S

T A B L E S

F I G U R E S

I N T R O D U C T I O N

The User's Guide provides an overview of the entire Turbo Pascal documentation set. Read the introduction in that book for information on how to most effectively use the Turbo Pascal manuals.

This manual contains materials for the advanced programmer. If you already know how to program well (whether in Pascal or another language), this manual is for you. It provides a language reference, information on the standard libraries, and programming information on memory and control issues, objects, floating point, overlays, video functions, assembly language interfacing, and the run-time and compile-time error messages.

Read the *User's Guide* if

1. You have never programmed in any language.
2. You have programmed, but not in Pascal, and you would like an introduction to the Pascal language.
3. You have programmed in Pascal but are not familiar with Borland's IDE (integrated development environment).
4. You are looking for information on how to install Turbo Pascal.

The *User's Guide* also contains reference information on Turbo Pascal's IDE (including the editor), the project manager, and the command-line compilers.

The *Library Reference* contains an alphabetical listing of all of Turbo Pascal's procedures and functions.

What's in this manual

This book is split into four parts: language grammar, the standard libraries, advanced programming issues, and interfacing with assembly language.

The first part of this manual, "The Turbo Pascal standard," offers technical information on the following features of the language:

- Chapter 1: "Tokens and constants"
- Chapter 2: "Blocks, locality, and scope"
- Chapter 3: "Types"
- Chapter 4: "Variables"
- Chapter 5: "Typed constants"
- Chapter 6: "Expressions"
- Chapter 7: "Statements"
- Chapter 8: "Procedures and functions"
- Chapter 9: "Programs and units"

The second part contains information about all the standard libraries: the *System, Dos, Graph* (in conjunction with the BGI), *Overlay,* and *Crt* units, along with 8087 information.

The third part provides further technical information for advanced users:

- Chapter 16: "Memory issues"
- Chapter 17: "Objects"
- Chapter 18: "Control issues"
- Chapter 19: "Input and output issues"
- Chapter 20: "Automatic optimizations"
- Chapter 21: "Compiler directives"

And the remaining fourth part discusses the issues involved with using Turbo Pascal with assembly language.

The two appendixes provide reference materials and list all the compiler and run-time error messages generated by Turbo Pascal.

P A R T

1

The Turbo Pascal standard

C H A P T E R

1

Tokens and constants

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

Separators cannot be part of tokens except in string constants.

A Pascal program is made up of tokens and separators, where 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.

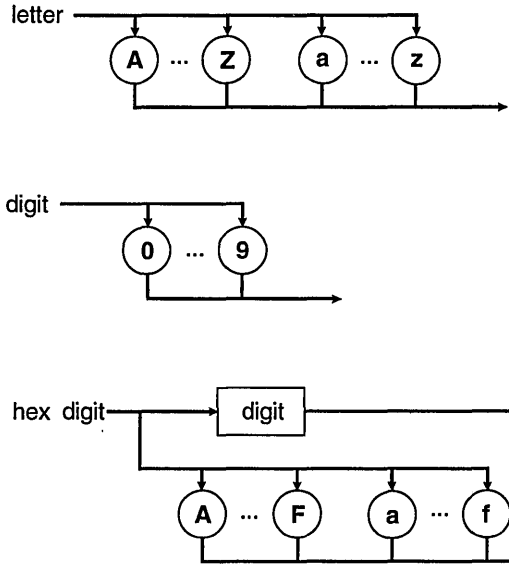
Special symbols and reserved words

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).

What follows are *syntax diagrams* for letter, digit, and hex digit. To read a syntax diagram, follow the arrows. Alternative paths are often possible; paths that begin at the left and end with an arrow on the right are valid. A path traverses boxes that hold the names of elements used to construct that portion of the syntax.

The names in rectangular boxes stand for actual constructions. Those in circular boxes—reserved words, operators, and punctuation—are the actual terms to be used in the program.



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

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

These character pairs are also special symbols:

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

Some special symbols are also operators. A left bracket ([) is equivalent to the character pair of left parenthesis and a period—(.. Similarly, a right bracket (]) is equivalent to the character pair of a period and a right parenthesis—.).

Following are Turbo Pascal's reserved words:

Table 1.1
Turbo Pascal reserved words

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

Reserved words appear in lowercase **boldface** throughout this manual. Turbo Pascal isn't case sensitive, however, so you can use either uppercase or lowercase letters in your programs.

The following are Turbo Pascal's standard directives. Unlike reserved words, these may be redefined by the user. However, this is not advised.

Table 1.2
Turbo Pascal standard directives

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

Note that **private** is a reserved word only within objects.

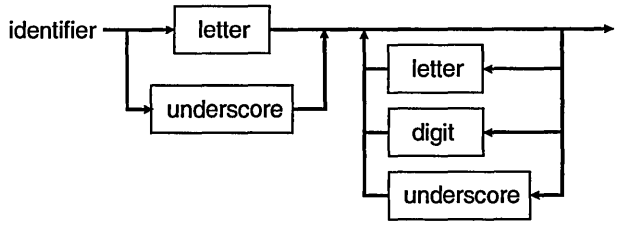
Identifiers

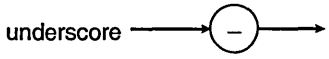
Identifiers denote constants, types, variables, procedures, functions, units, programs, and fields in records.

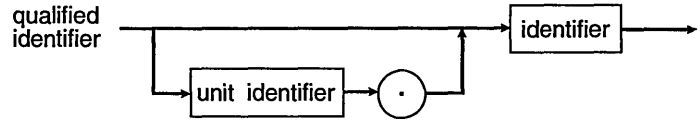
An identifier can be of any length, but only the first 63 characters are significant. An identifier must begin with a letter or an underscore character and cannot contain spaces. Letters, digits, and underscore characters (ASCII $5F) are allowed after the first character. Like reserved words, identifiers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

Units are described in Chapter 3 of the User's Guide and Chapter 9 of this manual.

When several instances of the same identifier exist, you may need to qualify the identifier by a *unit identifier* in order to select a specific instance. For example, to qualify the identifier *Ident* by the unit identifier *UnitName*, you would write *UnitName.Ident*. The combined identifier is called a *qualified identifier*.







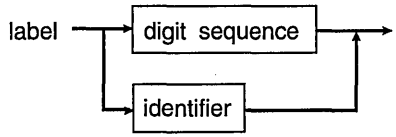
Here are some examples of identifiers:

```
Writeln
Exit
Real2String
System.MemAvail
Dos.Exec
Crt.Window
```

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

Labels

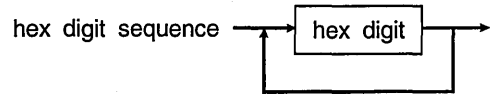
A *label* is a digit sequence in the range 0 to 9999. Leading zeros are not significant. Labels are used with **goto** statements.

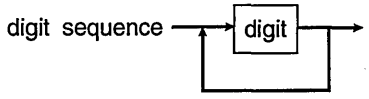


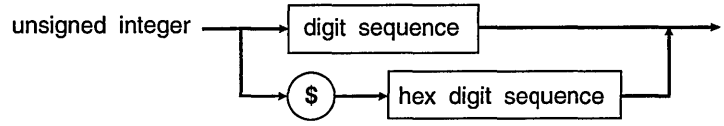
As an extension to standard Pascal, Turbo Pascal also allows identifiers to function as labels.

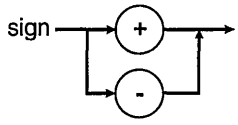
Numbers

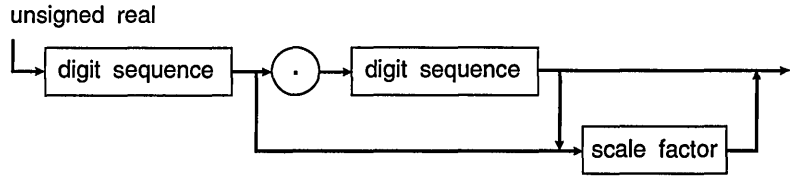
Ordinary decimal notation is used for numbers that are constants of type integer and real. A hexadecimal integer constant uses a dollar sign ($) as a prefix. Engineering notation (E or e, followed by an exponent) is read as "times ten to the power of" in real types. For example, 7E-2 means 7×10^{-2};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

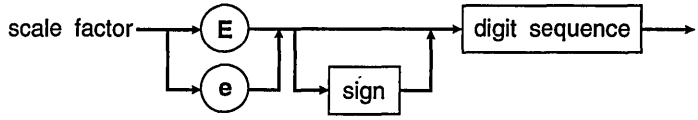


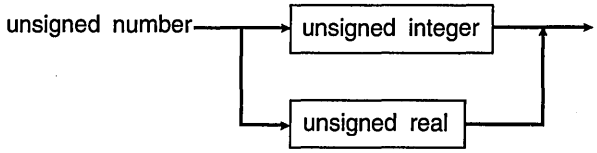


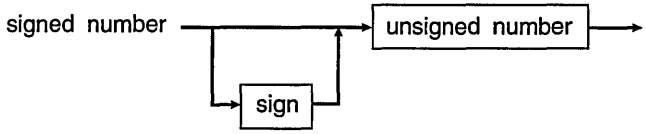












Numbers with decimals or exponents denote real-type constants. Other decimal numbers denote integer-type constants; they must be within the range –2,147,483,648 to 2,147,483,647.

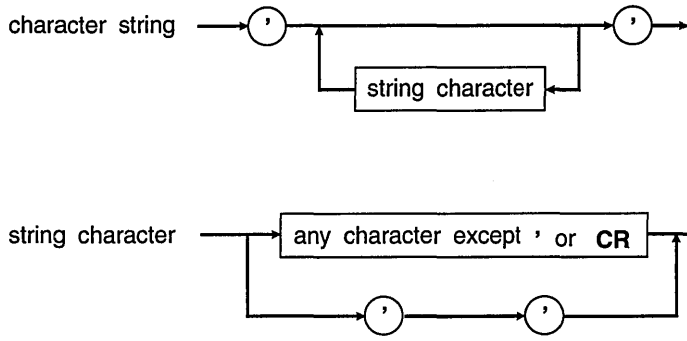
Hexadecimal numbers denote integer-type constants; they must be within the range $00000000 to $FFFFFFFF. The resulting value's sign is implied by the hexadecimal notation.

Character strings

A character string is a sequence of zero or more characters from the extended ASCII character set (Appendix B), written on one

line in the program and enclosed by apostrophes. A character string with nothing between the apostrophes is a *null string*. Two sequential apostrophes in a character string denote a single character, an apostrophe. The length attribute of a character string is the actual number of characters within the apostrophes.

As an extension to standard Pascal, Turbo Pascal allows control characters to be embedded 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.



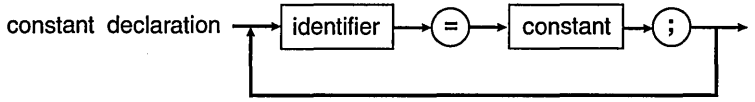
A character string of length zero (the null string) is compatible only with string types. A character string of length one is compatible with any Char and string type. A character string of length *N*, where *N* is greater than or equal to 2, is compatible with any string type and with packed arrays of *N* characters.

Here are some examples of character strings:

```
'TURBO'
'You''ll see'
''''
';'
' '
#13#10
'Line 1'#13'Line2'
#7#7'Wake up!'#7#7
```

Constant declarations

A constant declaration declares an identifier that marks a constant within the block containing the declaration. A constant identifier cannot be included in its own declaration.



As an extension to standard Pascal, Turbo Pascal allows 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
```

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

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

Since 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 in Chapter 5).
- function calls (except those noted in the following text)
- the address operator (@) (except in constant address expressions, as described in Chapter 5)

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

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

The following standard functions are allowed in constant expressions:

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

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(ErrorStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

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 summarized in Chapter 21.

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.

C H A P T E R

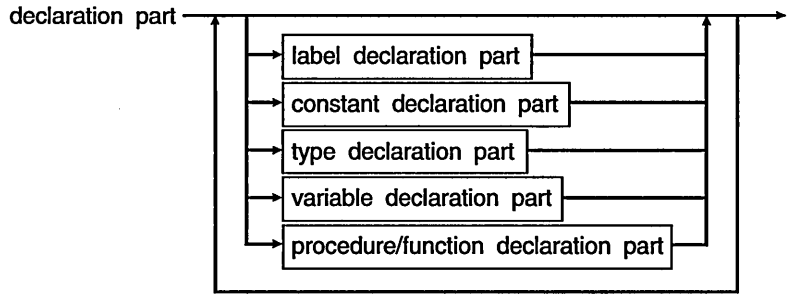
2

Blocks, locality, and scope

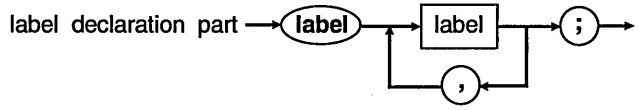
A block is made up of declarations, which are written and combined in any order, and statements. Each block is part of a procedure declaration, a function declaration, or a program or unit. All identifiers and labels declared in the declaration part are local to the block.

Syntax

The overall syntax of any block follows this format:

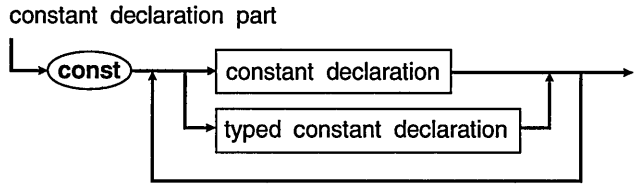


The *label declaration part* is where labels that mark statements in the corresponding statement part are declared. Each label must mark only one statement.

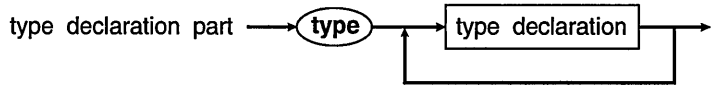


The digit sequence used for a label must be in the range 0 to 9999.

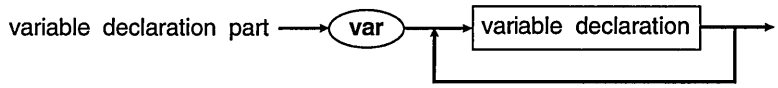
The *constant declaration part* consists of constant declarations local to the block.



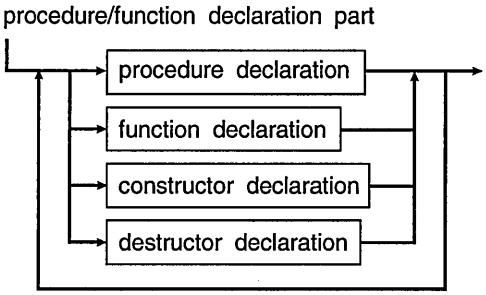
The *type declaration part* includes all type declarations local to the block.



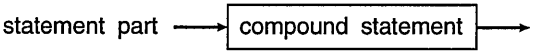
The *variable declaration part* is composed of variable declarations local to the block.



The *procedure and function declaration part* comprises 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. The scope of an identifier or label encompasses its declaration to the end of the current block, including all blocks enclosed by the current block; some exceptions follow:

- **Redeclaration in an enclosed block:** Suppose that *Exterior* is a block that encloses another block, *Interior*. If *Exterior* and *Interior* both have an identifier with the same name (for example, *J*)

then *Interior* can only access the *J* it declared, and similarly *Exterior* can only access the *J* it declared.

- **Position of declaration within its block:** Identifiers and labels cannot be used until after they are declared. An identifier or label's declaration must come before any occurrence of that identifier or label in the program text, with one exception.

  The base type of a pointer type can be an identifier that has not yet been declared. However, the identifier must eventually be declared in the same type declaration part that the pointer type occurs in.

- **Redeclaration within a block:** An identifier or label can only be declared *once* in the outer level of a given block. The only exception to this is when it is declared within a contained block or is in a record's field list.

  A record field identifier is declared within a record type and is significant only in combination with a reference to a variable of that record type. So, you can redeclare a field identifier (with the same spelling) within the same block but not at the same level within the same record type. However, an identifier that has been declared can be redeclared as a field identifier in the same block.

  The scope of an object component's identifier extends over the domain of the object type. See page 35 for further explanation.

Scope of interface and standard identifiers

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 entire program. 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 will select the instance declared by the last unit in the **uses** clause. However, by writing a qualified 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,

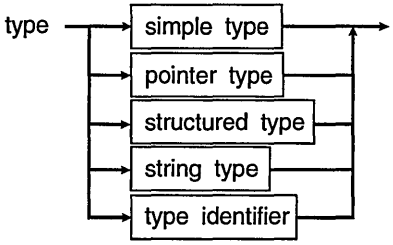
these standard objects are defined in a unit called *System*, which is used by any program or unit before the units named in the **uses** clause. This suggests that any unit or program can redeclare the standard identifiers, but a specific reference can still be made through a qualified identifier, for example, *System.Integer* or *System.Writeln*.

C H A P T E R

3

Types

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



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

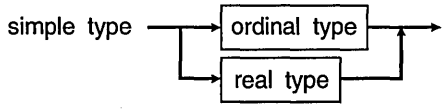
There are six major classes of types:

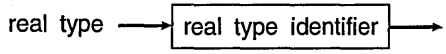
- simple types
- string types
- structured types
- pointer types
- procedural types
- object types

Each of these classes is described in the following sections.

Simple types

Simple types define ordered sets of values.





See Chapter 1 for how to denote constant type integer and real values.

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

Ordinal types

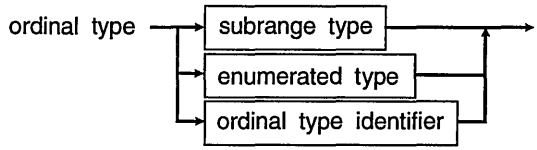
Ordinal types are a subset of simple types. All simple types other than real types are ordinal types, which are set off by four characteristics:

- 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 type Integer 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. A type Integer value's ordinality 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, *Pred* produces an 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, *Succ* produces an error.

The syntax of an ordinal type follows:



Turbo Pascal has seven predefined ordinal types: Integer, Shortint, Longint, Byte, Word, Boolean, 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 3.1
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 type Integer 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 instance, 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 4 and 6.

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

Boolean types

Type Boolean values are denoted by the predefined constant identifiers False and True. Because Boolean is an enumerated type, these relationships hold:

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

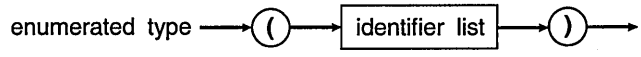
Char type

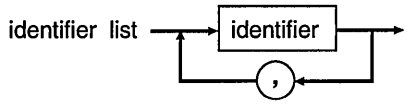
This type's set of values are characters, ordered according to the extended ASCII character set (Appendix B). The function call *Ord(Ch)*, where *Ch* is a Char value, returns *Ch*'s ordinality.

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 in which the identifiers are enumerated.





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

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

An example of an enumerated type follows:

```
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*.

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 is an enumerated type, such as the *color* type described previously. However, the intent of the declaration of *scale* is to define a subrange type. The solution is to either reorganize the first subrange expression so that it does not start with a parenthesis, or to set another constant equal to the value of the expression, and then 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 (see the next table).

Table 3.2
Real data types

Type	Range	Significant digits	Size in bytes
Real	$2.9 \times 10^{-39} .. 1.7 \times 10^{38}$	11-12	6
Single	$1.5 \times 10^{-45} .. 3.4 \times 10^{38}$	7-8	4
Double	$5.0 \times 10^{-324} .. 1.7 \times 10^{308}$	15-16	8
Extended	$3.4 \times 10^{-4932} .. 1.1 \times 10^{4932}$	19-20	10
Comp	$-2^{63}+1 .. 2^{63}-1$	19-20	8

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

Turbo Pascal supports two models of code generation for performing real-type operations: *software* floating point and *8087* floating point. The appropriate model is selected through the **$N** compiler directive. If no 8087 is present, enabling the **$E** compiler directive will provide full 8087 emulation in software.

Software floating point

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

8087 floating point

In the {**$N+**} state, the code generated performs all real type calculations using 8087 instructions. This state permits the use of all five real types.

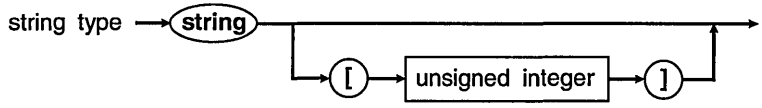
For further details on 8087 floating-point code generation and software emulation, refer to Chapter 14, "Using the 8087."

Turbo Pascal includes a run-time library that will automatically *emulate* an 8087 in software if one is not present; the **$E** compiler directive is used to determine whether or not the 8087 emulator should be included in a program.

String types

Type string operators are described in "String operator" and "Relational operators" in Chapter 6. Type string standard procedures and functions are described in "String procedures and functions" on page 127.

A type string 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 only be equal to other null strings, and they hold the least string values.

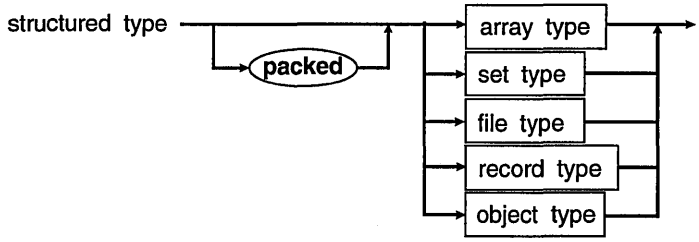
See the section "Arrays, strings, and indexes" in Chapter 4.

Characters in a string can be accessed as components of an array.

Structured types

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

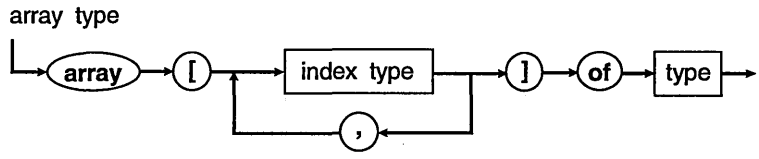
A structured type, characterized by its structuring method and by its component type(s), holds more than one value. If a component type is structured, the resulting structured type has more than one level of structuring. A structured type can have unlimited levels of structuring.

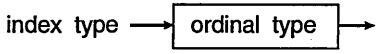


The word **packed** in a structured type's declaration tells the compiler to compress data storage, even at the cost of diminished access to a component of a variable of this type. The word **packed** has no effect in Turbo Pascal; instead packing occurs automatically whenever possible.

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; the number of elements is therefore the number of values in each index type. The number of dimensions is unlimited.

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 instance,

```
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" in Chapter 4.

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

An array type of the form

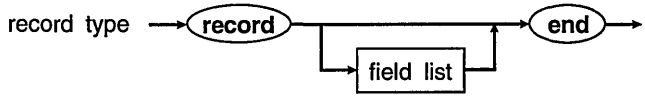
```
packed array[M..N] of Char
```

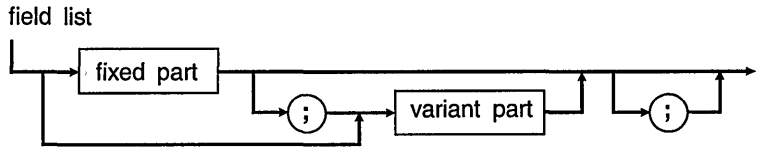
See "Identical and compatible types" later in this chapter.

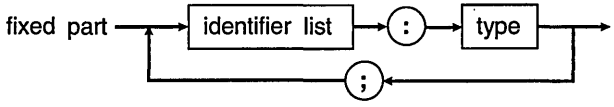
where *M* is less than *N* is called a packed string type (the word **packed** can be omitted because it has no effect in Turbo Pascal). A packed string type has certain properties not shared by other array types .

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.







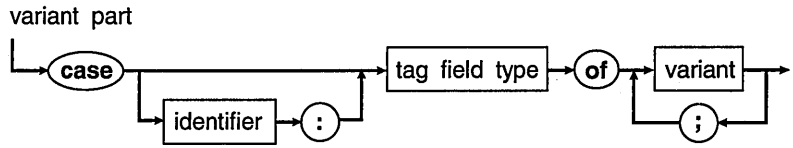
The fixed part of a record type sets out the list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

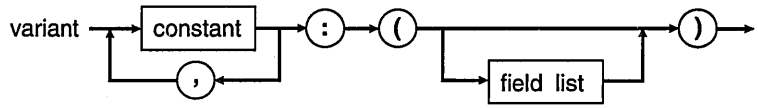
The following is an example of a record type:

```
type
  DateRec = 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
  Person = record
```

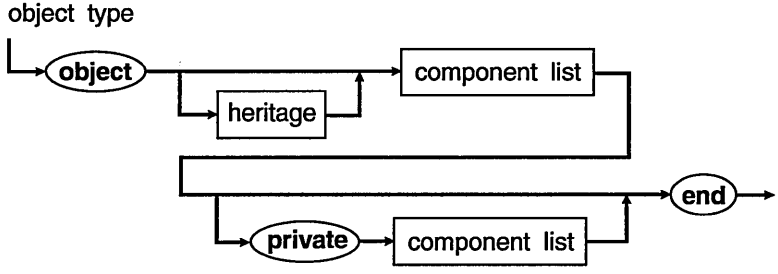
```
    FirstName, LastName: string[40];
    BirthDate: Date;
    case Citizen: Boolean of
      True: (BirthPlace: string[40]);
      False: (Country: string[20];
        EntryPort: string[20];
        EntryDate: Date;
        ExitDate: Date);
  end;

  Polygon = record
    X, Y: Real;
    case Kind: Figure of
      Rectangle: (Height, Width: Real);
      Triangle: (Size1, Side2, Angle: Real);
      Circle: (Radius: Real);
  end;
```

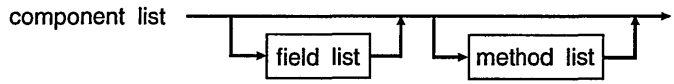
Object types

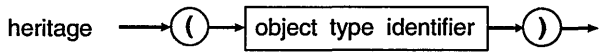
An object type is a structure consisting of a fixed number of components. Each component is either a *field*, which contains data of a particular type, or a *method*, which performs an operation on the object. Analogous to a variable declaration, the declaration of a field specifies the data type of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

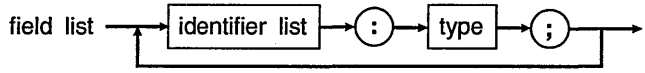
An object type can *inherit* components from another object type. If *T2* inherits from *T1*, then *T2* is a *descendant* of *T1*, and *T1* is an *ancestor* of *T2*.

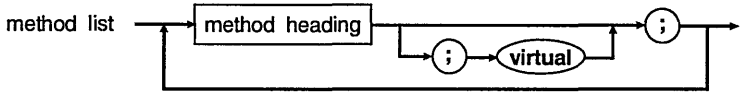
Inheritance is transitive, that is, if *T3* inherits from *T2*, and *T2* inherits from *T1*, then *T3* also inherits from *T1*. The *domain* of an object type consists of itself and all its descendants.

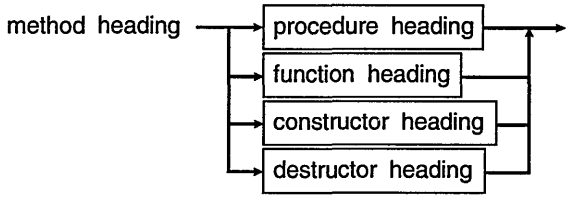












The following code shows examples of object type declarations. These declarations are referred to by other examples throughout this chapter.

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

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

```
  procedure Intersect(var R: Rect);
  procedure Union(var R: Rect);
  function Contains(P: Point): Boolean;
end;

StringPtr = ^String;

FieldPtr = ^Field;

Field = object
  X, Y, Len: Integer;
  Name: StringPtr;
  constructor Copy(var F: Field);
  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;
end;

StrFieldPtr = ^StrField;

StrField = object(Field)
  Value: StringPtr;
  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;

NumFieldPtr = ^NumField;

NumField = object(Field)
  Value, Min, Max: Longint;
  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;

ZipFieldPtr = ^ZipField;

ZipField = object(NumField)
  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. Thus, an object type cannot be declared in a variable declaration part or within a procedure, function, or method block.

Components and scope

The component type of a file type cannot be an object type, or any structured type with an object type component.

The scope of a component identifier extends over the domain of its object type. Furthermore, 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.

The scope of a component identifier declared in the **private** section of an object type declaration 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 others **private** components without making the **private** components known to other's modules.

Methods

The declaration of a method within an object type corresponds to a **forward** declaration of that method. Thus, 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.

Within an object type declaration, a method heading can specify parameters of the object type being declared, even though the declaration is not yet complete. This is illustrated by the *Copy*, *Intersect*, and *Union* methods of the *Rect* type in the previous example.

Virtual methods

Methods are by default *static*, but can, with the exception of constructor methods, be made *virtual* through the inclusion of a **virtual** directive in the method declaration. The compiler resolves calls to static methods at compile time, whereas calls to virtual methods are resolved at run time. The latter is sometimes referred to 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. Thus, 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 is introduced, or until the method identifier is again overridden.

An override of a static method is free to change the method heading in 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. Furthermore, the override must again include a **virtual** directive.

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* standard procedure to a pointer variable of an object type. The resulting object is called an *instance* of the object type.

```
var
  F: Field;
  Z: ZipField;
  FP: FieldPtr;
  ZP: ZipFieldPtr;
```

Given these variable declarations, *F* is an instance of *Field*, and *Z* is an instance of *ZipField*. Likewise, after applying *New* to *FP* and *ZP*, *FP* points to an instance of *Field*, and *ZP* points to an instance of *ZipField*.

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: StrField;
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* would cause this example to fail.

Important! Assignment to an instance of an object type does *not* entail initialization of the instance.

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

```
var
  Comment: array[1..5] of StrField;
  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* standard procedures. Here's an example:

```
var
  SP: StrFieldPtr;
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 *ZipFieldPtr* can be assigned to pointers of type *ZipFieldPtr*, *NumFieldPtr*, and *FieldPtr*, and during execution of a program, a pointer of type *FieldPtr* might be either **nil** or point to an instance of *Field*, *StrField*, *NumField*, or *ZipField*, or any other instance of a descendant of *Field*.

These pointer assignment compatibility rules also apply to object type variable parameters. For example, the *Field.Copy* method might be passed an instance of *Field*, *StrField*, *NumField*, *ZipField*, or any other instance of a descendant of *Field*.

A method is activated through a method designator of the form *Instance.Method*, where *Instance* is an instance of an object type, and *Method* is a method of that object type.

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

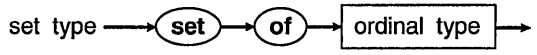
For virtual methods, the *actual* (run-time) type of *Instance* governs the selection. For example, the designator *FP^.Edit* might activate *Field.Edit*, *StrField.Edit*, *NumField.Edit*, or *ZipField.Edit*, depending on the actual type of the instance pointed to by *FP*.

In general, there is no way of determining which method will be activated by a virtual method designator. You can develop a routine (such as a forms editor input routine) that activates *FP^.Edit*, and later, without modifying that routine, apply it to an instance of a new, unforeseen descendant type of *Field*. When extensibility of this sort is desired, you should employ an object type with an open-ended set of descendant types, rather than a record type with a closed set of variants.

Set types

A set type's range of values is the power set of a particular ordinal type (the base type). 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.



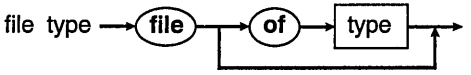
Set-type operators are described in the section entitled "Set operators" in Chapter 6. "Set constructors" in the same chapter shows how to construct set values.

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. For these reasons, the base type of a set cannot be Shortint, Integer, Longint, or Word.

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 or any structured type with a file-type component. The number of components is not set by the file-type declaration.

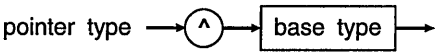


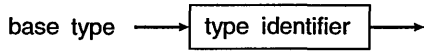
If the word **of** and the component type are omitted, the type denotes an untyped file. Untyped files are low-level I/O channels primarily used for direct access to any disk file regardless of its internal format.

The standard file type *text* signifies a file containing characters organized into lines. Text files use special input/output (I/O) procedures, which are discussed in Chapter 19, "Input and output issues."

Pointer types

A pointer type defines a set of values that point to dynamic variables of a specified type called the *base* type. A type Pointer 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. The *New* procedure 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, including variables that already have identifiers. The *Ptr* function points the pointer variable to a specific memory address.

The reserved word **nil** denotes a pointer-valued constant that does not point to anything.

See Chapter 4's section entitled "Pointers and dynamic variables" for the syntax of referencing the dynamic variable pointed to by a pointer variable.

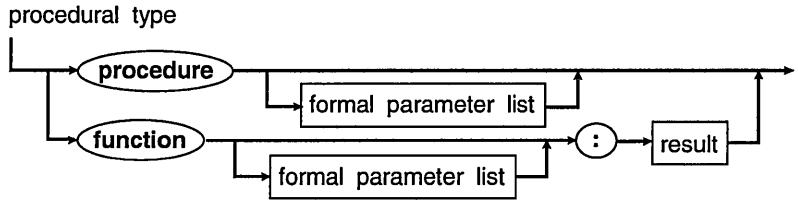
The predefined type Pointer denotes an untyped pointer, that is, a pointer that does not point to any specific type. Variables of type Pointer cannot be dereferenced; writing the pointer symbol ^ after such a variable is an error. Like the value denoted by the word **nil**, values of type Pointer are compatible with all other pointer types.

Procedural types

For a complete discussion of procedural types, refer to the "Procedural types" section on page 108.

Standard Pascal regards procedures and functions strictly 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 objects 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.



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 meaning of the declaration.

Turbo Pascal does not let you declare functions that return procedural type values; a function result value must be a string, Real, Integer, Char, Boolean, Pointer, or a user-defined enumeration.

Identical and compatible types

Two types may 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* does not 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, since **set of** Integer is not 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 a string type, packed string type, or Char type.
- One type is Pointer and the other is any pointer type.
- 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 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.
- An object type T_2 is assignment compatible with an object type T_1 if T_2 is 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 or run-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, and functions that declare types have a type declaration part. An example of this follows:

```
type
  Range = Integer;
  Number = Integer;
  Color = (Red, Green, Blue);
  CharVal = Ord('A')..Ord('Z');
  TestIndex = 1..100;
  TestValue = -99..99;
  TestList = array[TestIndex] of TestValue;
  TestListPtr = ^TestList;
  Date = object
    Year: Integer;
    Month: 1..12;
    Day: 1..31;
    procedure SetDate(D, M, Y: Integer);
    function ShowDate: String;
  end;
  MeasureData = record
    When: Date;
    Count: TestIndex;
    Data: TestListPtr;
  end;
  MeasureList = array[1..50] of MeasureData;
  Name = string[80];
  Sex = (Male, Female);
  Person = ^PersonData;
  PersonData = record
    Name, FirstName: Name;
    Age: Integer;
    Married: Boolean;
    Father, Child, Sibling: Person;
    case S: Sex of
      Male: (Bearded: Boolean);
      Female: (Pregnant: Boolean);
```

```
end;
PersonBuf = array[0..SizeOf(PersonData)-1] of Byte;
People = file of PersonData;
```

In the example, *Range*, *Number*, and Integer are identical types. *TestIndex* 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 *CharVal* and *PersonBuf*.

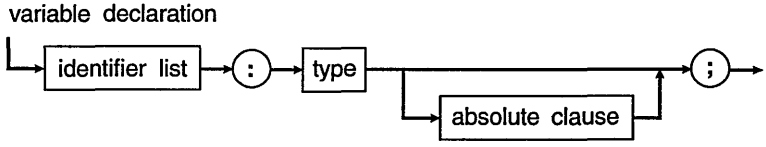
C H A P T E R

4

Variables

Variable declarations

A variable declaration embodies a list of identifiers that designate new variables and their type.



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

When an identifier is specified within the identifier list of a variable declaration, that identifier is a variable identifier for the block in which the declaration occurs. The variable can then be referred to throughout the block, unless the identifier is redeclared in an enclosed block. Redeclaration causes 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 reside in the *data segment*. Variables declared within procedures and functions are called *local variables*, and 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 further details on this subject, see "Pointers and dynamic variables" on page 53.

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.

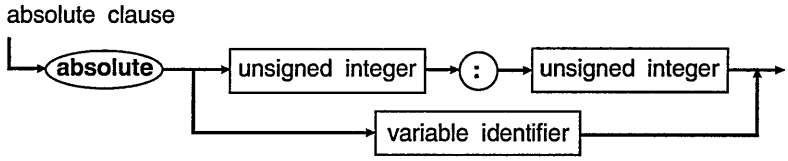
Each time a procedure or function is activated (called), it allocates a set of local variables on the stack. On exit, the local variables are disposed. At any time during the execution of a program, the total size of the local variables allocated by the active procedures and functions cannot exceed the 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 may very well cause a system crash, so don't turn off stack checks unless you are absolutely sure that an overflow will never occur.

Absolute variables

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



Note that 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;
```

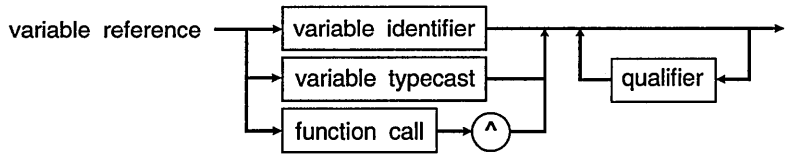
This declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and because the first byte of a string variable contains the dynamic length of the string, *StrLen* will contain the length of *Str*.

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

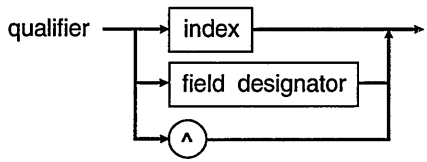
The syntax for a variable reference is



Note that the syntax for a variable reference allows a function call to a pointer function. The resulting pointer is then dereferenced to denote a dynamic variable.

Qualifiers

A variable reference is a variable identifier with 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 (a ^) to differentiate between the pointer field and the dynamic variable it points to.

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

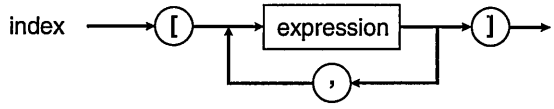
If the variable being pointed to is an array, indexes can be added to denote components of this array.

```
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. Furthermore, 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

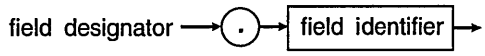
```
Matrix[I, J]
```

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 does not check whether this value is less than the declared size of the string. It is possible to index a string beyond its current dynamic length. The characters thus read are random, and assignments beyond the current length will not affect the actual value of the string variable.

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.



Some examples of a field designator include the following:

```
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 was written before the component reference.

Pointers and dynamic variables

The value of a pointer variable is either **nil** or the address of a value that points to 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 standard procedures *New* and *GetMem*. You can use the @ (address-of) operator and the standard function *Ptr* to create pointer values that are treated as pointers to dynamic variables.

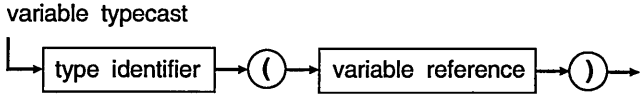
nil does not point to any variable. The results are undefined if you access a dynamic variable when the pointer's value is **nil** or undefined.

Some 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*.



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.

Some examples of variable typecasts follow:

```
type
  ByteRec = record
    Lo, Hi: Byte;
  end;
  WordRec = record
    Low, High: Word;
  end;
  PtrRec  = record
    Ofs, Seg: Word;
  end;
  BytePtr = ^Byte;
var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;
begin
  W := $1234;
  B := ByteRec(W).Lo;
  ByteRec(W).Hi := 0;
  L := $01234567;
  W := WordRec(L).Lo;
  B := ByteRec(WordRec(L).Lo).Hi;
  B := BytePtr(L)^;
  P := Ptr($40,$49);
  W := PtrRec(P).Seg;
  Inc(PtrRec(P).Ofs, 4);
end.
```

Notice the use of the *ByteRec* 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 hand side of an assignment. Also, observe the use of the *WordRec* and *PtrRec* 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: Func;
  P: Pointer;
  N: Integer;
```

you can construct the following assignments:

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

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

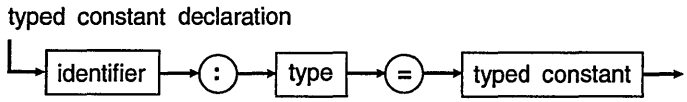
C H A P T E R

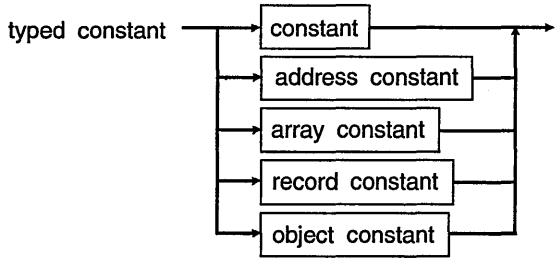
5

Typed constants

See the section entitled "Constant declarations" in Chapter 1.

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





Typed constants can be used exactly like variables of the same type, and 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. Thus, for each entry to a procedure or function, the locally declared typed constants are not reinitialized.

In addition to a normal constant expression, the value of a typed constant may 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 cannot reference local variables or dynamic (heap based) variables, since their addresses cannot be computed at compile-time.

Simple-type constants

Declaring a typed constant as a simple type simply 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 may 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 cannot be interchanged with ordinary constants. For instance, it cannot be used in the declaration of other constants or types.

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

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

String-type constants

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

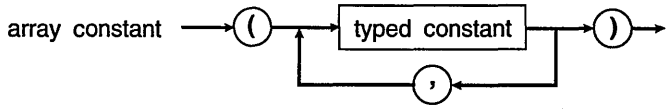
```
const
  Heading: string[7] = 'Section';
  NewLine: string[2] = #13#10;
  TrueStr: string[5] = 'Yes';
  FalseStr: string[5] = 'No';
```

Structured-type constants

The declaration of a structured-type constant specifies the value of each of the structure's components. Turbo Pascal supports the declaration of type array, record, set, and pointer constants; type file constants, and constants of array and record types that contain type file components are not allowed.

Array-type constants

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



An example of an array-type constant follows:

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

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

```
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';
```

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

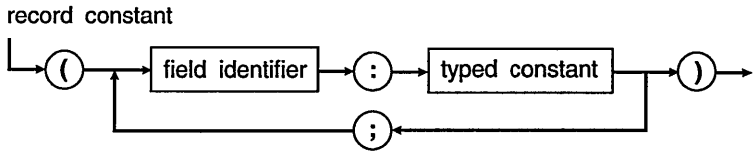
```
type
  Cube = array[0..1, 0..1, 0..1] of Integer;
const
  Maze: Cube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));
```

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.



Some examples of record constants follow:

```
type
  Point = record
    X, Y: Real;
  end;
  Vector = array[0..1] of Point;
  Month = (Jan, Feb, Mar, Apr, May, Jun, Jly, Aug, Sep, Oct,
    Nov, Dec);
  Date = record
    D: 1..31;
    M: Month;
    Y: 1900..1999;
  end;
const
  Origin: Point = (X: 0.0; Y: 0.0);
  Line: Vector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: Date = (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 cannot 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, here are some examples of object-type constants:

```
const
  ZeroPoint: Point = (X: 0; Y: 0);
  ScreenRect: Rect = (A: (X: 0; Y: 0); B: (X: 80; Y: 25));
  CountField: NumField = (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. Some examples follow:

```
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 typically uses a constant address expression to specify the pointer value. Some examples follow:

```
type
  Direction = (Left, Right, Up, Down);
  StringPtr = ^String;
  NodePtr = ^Node;
  Node = record
    Next: NodePtr;
    Symbol: StringPtr;
    Value: Direction;
  end;
const
  S1: string[4] = 'DOWN';
  S2: string[2] = 'UP';
  S3: string[5] = 'RIGHT';
  S4: string[4] = 'LEFT';
  N1: Node = (Next: nil; Symbol: @S1; Value: Down);
  N2: Node = (Next: @N1; Symbol: @S2; Value: Up);
```

```
N3: Node = (Next: @N2; Symbol: @S3; Value: Right);
N4: Node = (Next: @N3; Symbol: @S4; Value: Left);
DirectionTable: NodePtr = @N4;
```

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. An example follows:

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

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

const
  ErrorHandler: ErrorProc = DefaultError;
```

C H A P T E R

6

Expressions

Expressions are made up of *operators* and *operands*. Most Pascal operators are *binary*, that is, 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 (see the following table).

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:

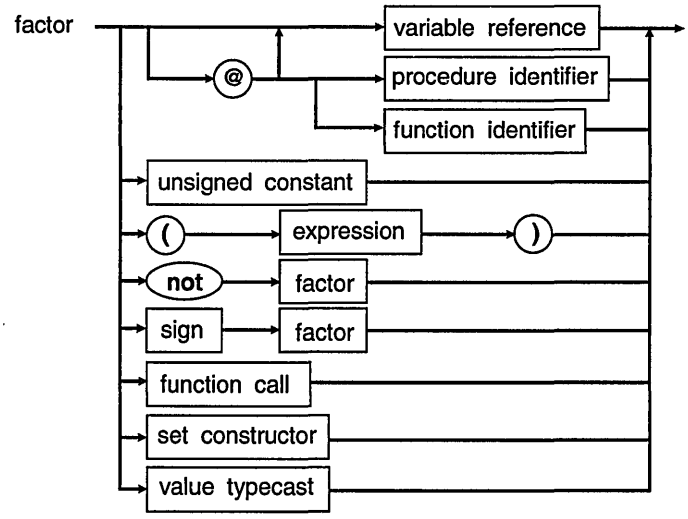
1. An operand between two operators of different precedence is bound to the operator with higher precedence.
2. An operand between two equal operators is bound to the one on its left.
3. Expressions within parentheses are evaluated prior to being treated as a single operand.

Operations with equal precedence are normally performed from left to right, although the compiler may at times 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:



See "Function calls" on page 76.

A function call activates a function and denotes the value returned by the function.

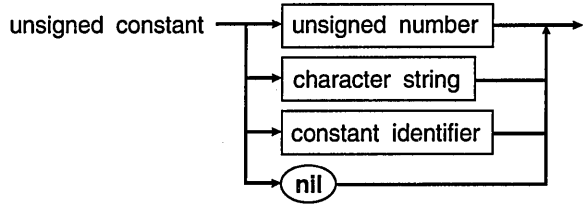
See "Set constructors" on page 77.

A set constructor denotes a value of a set type.

See "Value typecasts" on page 78.

A value typecast changes the type of a value.

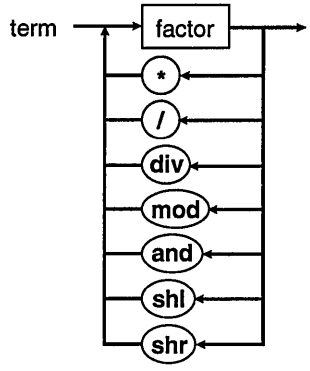
An unsigned constant has the following syntax:



Some examples of factors include the following:

```
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 }
```

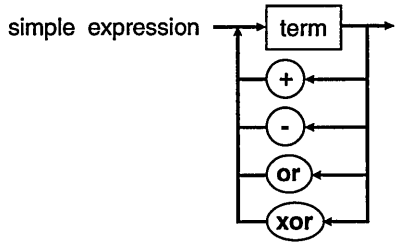
Terms apply the multiplying operators to factors:



Here are some examples of terms:

```
X * Y
Z / (1 - Z)
Done or Error
(X <= Y) and (Y < Z)
```

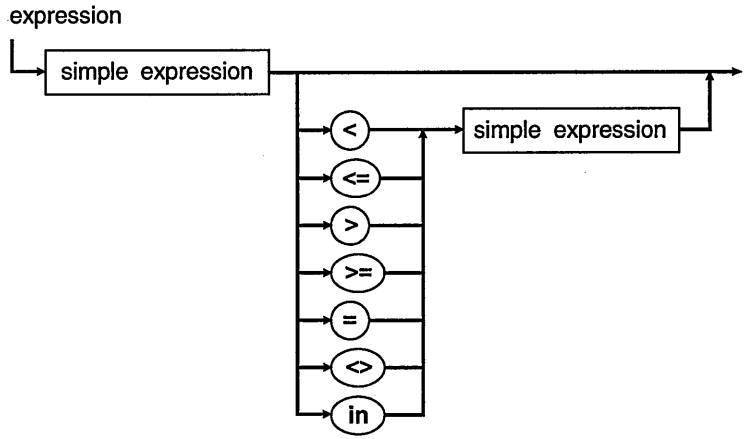
Simple expressions apply adding operators and signs to terms:



Here are some examples of simple expressions:

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

An expression applies the relational operators to simple expressions:



Here are some examples of expressions:

```
X = 1.5
Done <> Error
(I < J) = (J < K)
C in Hue1
```

Operators

The operators are classified as arithmetic operators, logical operators, string 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 6.3
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.

See the section "Integer types" in Chapter 3 for a definition of common types.

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. An 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. An 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*. An 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	integer type
and	bitwise and	integer type	integer type
or	bitwise or	integer type	integer type
xor	bitwise xor	integer type	integer type
shl	shift left	integer type	integer type
shr	shift right	integer type	integer type

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 or to the right by *J* bits. The type of the result 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	Boolean
and	logical and	Boolean	Boolean
or	logical or	Boolean	Boolean
xor	logical xor	Boolean	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, since 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 instance:

```
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-**} (unless changed using the **O**ptions | **C**ompiler menu), and in this state short-circuit evaluation code is generated. In the {**$B+**} state, complete evaluation code is generated.

Since standard Pascal does not specify which model should be used for Boolean expression evaluation, programs depending on either model being in effect are not truly portable. However, sacrificing portability is often worth gaining the 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 is truncated after character 255.

Set operators

The types of operands for set operations are shown in Table 6.7.

Table 6.7
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 6.8.

Table 6.8
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, or packed string types	Boolean
>	greater than	compatible simple, string, or packed string types	Boolean
<=	less or equal	compatible simple, string, or packed string types	Boolean
>=	greater or equal	compatible simple, string, or packed string types	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 of =, <>, <, >, >=, 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, and 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 is 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.

When it compares pointers, Turbo Pascal simply compares the segment and offset parts. Because of the segment mapping scheme of the 80x86 processors, two logically different pointers can in fact point to the same physical memory location. For instance, *Ptr*($0040,$0049) and *Ptr*($0000,$0449) are two pointers to the same physical address. Pointers returned by the standard procedures *New* and *GetMem* are always normalized (offset part in the range $0000 to $000F), and will therefore always compare correctly. When creating pointers with the *Ptr* standard function, special care must be taken if such pointers are to be compared.

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

A pointer to a variable can be created with the @ operator. Table 6.9 shows the operand and result types.

Table 6.9
Pointer operation

Operator	Operation	Operand types	Result type
@	Pointer formation	Variable reference or procedure or function identifier	Pointer (same as **nil**)

Special rules apply to use of the @ operator with a procedural variable. For further details, see "Procedural types" on page 108.

@ is a unary operator that takes a variable reference or a procedure or function identifier as its operand, and returns a pointer to the operand. The type of the value is the same as the type of **nil**, therefore it can be assigned to any pointer variable.

@ with a variable

The use of @ with an ordinary variable (not a parameter) is uncomplicated. Given the declarations

```
type
  TwoChar = array[0..1] of Char;
var
  Int: Integer;
  TwoCharPtr: ^TwoChar;
```

then the statement

```
TwoCharPtr := @Int;
```

causes *TwoCharPtr* to point to *Int*. *TwoCharPtr*^ becomes a re-interpretation of the value of *Int*, as though it were an `array[0..1] of Char`.

@ with a value parameter

Applying @ to a formal value parameter results in a pointer to the stack location containing the actual value. Suppose *Foo* is a formal value parameter in a procedure and *FooPtr* is a pointer variable. If the procedure executes the statement

```
FooPtr := @Foo;
```

then *FooPtr*^ references *Foo*'s value. However, *FooPtr*^ does not reference *Foo* itself, rather it references the value that was taken from *Foo* and stored on the stack.

@ with a variable parameter

Applying @ to a formal variable parameter results in a pointer to the actual parameter (the pointer is taken from the stack). Suppose *One* is a formal variable parameter of a procedure, *Two* is a variable passed to the procedure as *One*'s actual parameter, and *OnePtr* is a pointer variable. If the procedure executes the statement

```
OnePtr := @One;
```

then *OnePtr* is a pointer to *Two*, and *OnePtr^* is a reference to *Two* itself.

@ with a procedure or function

You can apply @ to a procedure or a function to produce a pointer to its entry point. Turbo Pascal does not give you a mechanism for using such a pointer. The only use for a procedure pointer is to pass it to an assembly language routine or to use it in an **inline** statement. See "Turbo Assembler and Turbo Pascal" on page 308 for information on interfacing Turbo Assembler and Turbo Pascal.

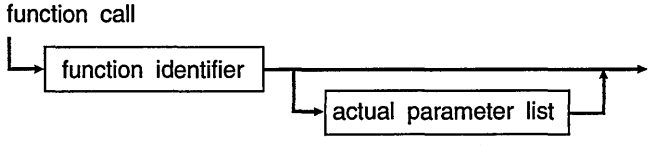
@ with a method

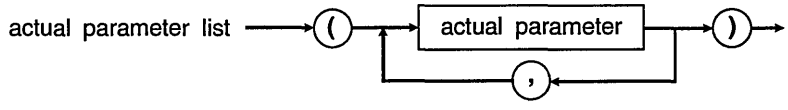
You can apply @ to a qualified method identifier to produce a pointer to the method's entry point.

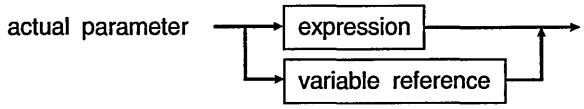
Function calls

A function call activates the function specified by the function identifier. Any identifier declared to denote a function is a function identifier.

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 set forth in Chapter 19, "Input and output issues."







Some examples of function calls follow:

A function can also be invoked via a procedural variable. For further details, refer to the "Procedural types" section on page 108.

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

The syntax of a function call has been extended to allow a method designator or a qualified method identifier denoting a function to replace the function identifier.

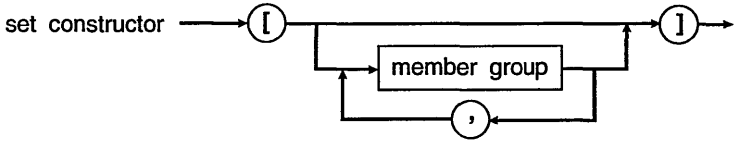
The discussion of extensions to procedure statements in the section, "Procedure statements" in Chapter 7 also applies to function calls.

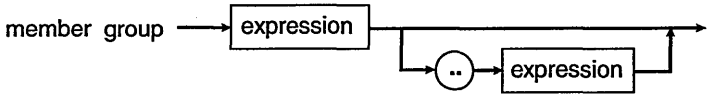
See "Extended syntax" in Chapter 21 for details.

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* does not 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.

Some examples of set constructors follow:

```
[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 53.

The syntax of a value typecast is almost identical to that of a variable typecast. However, value typecasts operate on values, not on variables, and can therefore not participate in variable references; that is, a value typecast may not be followed by qualifiers. In particular, value typecasts cannot appear on the left-hand side of an assignment statement.

Some examples of value typecasts include the following:

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

```
Longint(@Buffer)
BytePtr(Ptr($40, $49))
```

Procedural types in expressions

In general, the use of a procedural variable in a statement or an expression denotes a call of the procedure or function stored in the variable. There is however one exception: When Turbo Pascal sees a procedural variable on the left-hand side of an assignment statement, it knows that the right-hand 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 cannot 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 whether it should call *F* and *ReadInt*, and then compare the returned values.

```
if F = ReadInt then
  WriteLn('Equal');
```

However, standard Pascal syntax 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. Thus, *@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*.

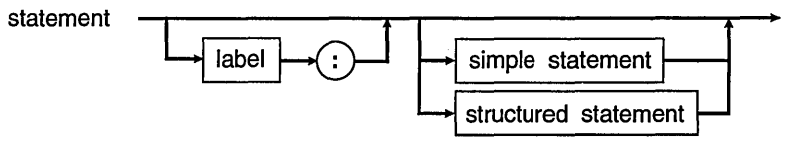
To get the memory address of a procedural variable, rather than the address stored in it, a double address (**@@**) operator must be used. For example, where *@P* means convert *P* into an untyped pointer variable, *@@P* means return the physical address of the variable *P*.

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.

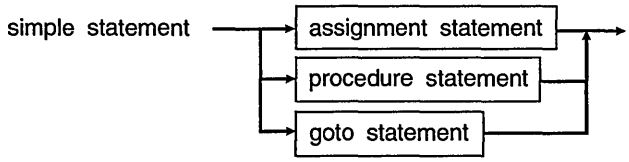


As you saw in Chapter 1, a label is either a digit sequence in the range 0 to 9999 or an identifier.

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

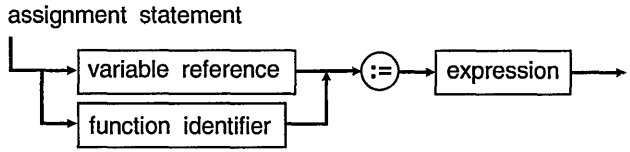
Simple statements

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



Assignment statements

Assignment statements either replace the current value of a variable with a new value specified by an expression or specify an expression whose value is to be returned by a function.



See the section "Type compatibility" on page 42.

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

Some examples of assignment statements follow:

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

Object type assignments

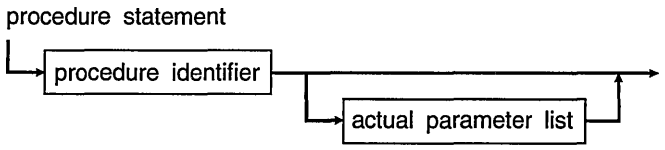
The rules of object type assignment compatibility allow an instance of an object type to be assigned an instance of any of its descendant types. Such an assignment constitutes a *projection* of the descendant onto the space spanned by its ancestor. For example, given an instance *F* of type *Field*, and an instance *Z* of type *ZipField*, the assignment *F* := *Z* will copy only the fields *X*, *Y*, *Len*, and *Name*.

Object types are discussed in more detail in Chapter 5.

Assignment to an instance of an object type does *not* entail initialization of the instance. Referring to the preceding example, the assignment *F* := *Z* does not mean that a constructor call for *F* can be omitted.

Procedure statements

A **procedure** statement specifies the activation of the procedure denoted by the procedure identifier. 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.



A procedure can also be invoked via a procedural variable. For further details, refer to the "Procedural types" section on page 108.

Some examples of procedure statements follow:

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

Method, constructor, and destructor calls

The syntax of a procedure statement has been extended to allow a method designator denoting a procedure, constructor, or destructor to replace the procedure identifier.

The instance denoted by the method designator serves two purposes. First, in the case of a virtual method, the *actual* (run time) type of the instance determines which implementation of the method is activated. Second, the instance itself becomes an implicit actual parameter of the method; it corresponds to a formal variable parameter named *Self* that possesses the type corresponding to the activated method.

Within a method, a procedure statement allows a qualified method identifier to denote activation of a specific method. The object type given in the qualified identifier must be the same as the method's object type, or an ancestor of it. This type of activation is called a *qualified activation*.

The implicit *Self* parameter of a qualified activation becomes the *Self* of the method containing the call. A qualified activation never

employs the virtual method dispatch mechanism—the call is always static and always invokes the specified method.

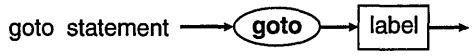
A qualified activation is generally used within an override method to activate the overridden method. Referring to the types declared earlier, here are some examples of qualified activations:

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

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

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

Goto statements

A **goto** statement transfers program execution to the statement prefixed by the label referenced in the **goto** statement. The syntax diagram of a **goto** statement follows:

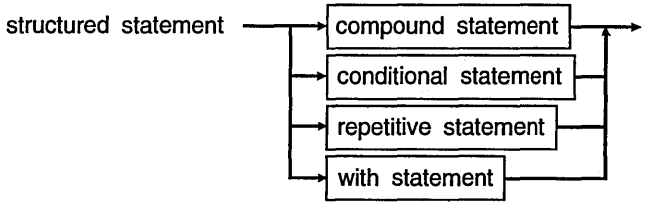


The following rules should be observed when using **goto** statements:

- The label referenced by a **goto** statement must be in the same block as the **goto** statement. In other words, it is 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 will not indicate an error.

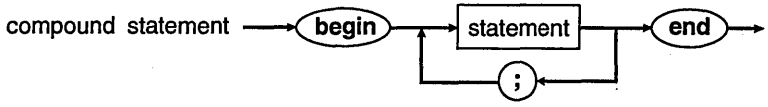
Structured statements

Structured statements are constructs composed of other statements that are to be executed in sequence (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.

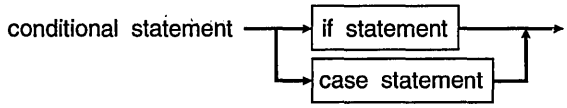


Here's an example of a compound statement:

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

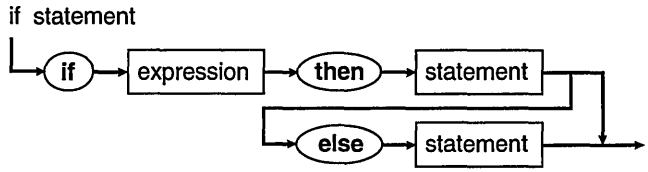
Conditional statements

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 is not present, nothing is executed.

The syntactic ambiguity arising from the construct

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

is resolved by interpreting the construct as follows:

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

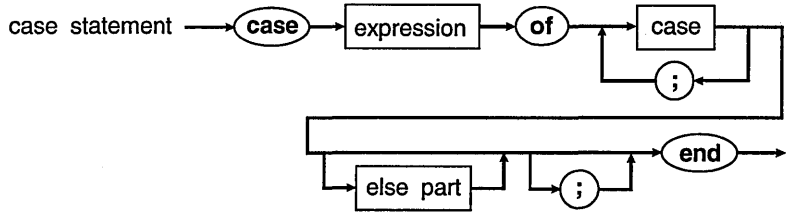
In general, an **else** is associated with the closest **if** not already associated with an **else**.

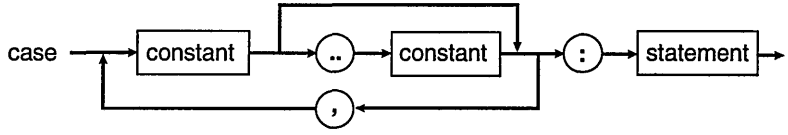
Two examples of **if** statements follow:

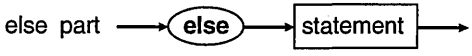
```
if X < 1.5 then
  Z := X + Y
else
  Z := 1.5;

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. Thus, 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, nothing is executed.

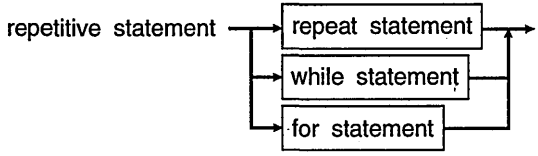
Examples of **case** statements include

```
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

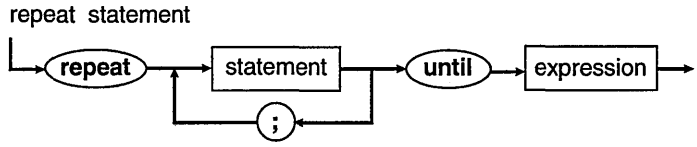
Repetitive statements specify certain statements to be executed repeatedly.



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

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.

Examples of **repeat** statements follow:

```
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 is not executed at all.

Examples of **while** statements include:

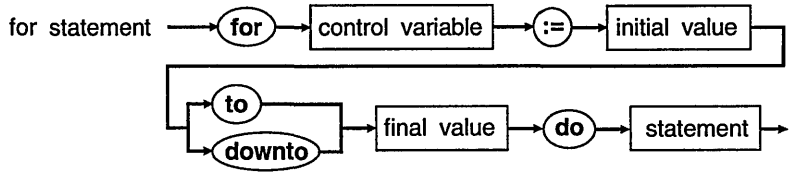
```
while Data[I] <> X do I := I + 1;
```

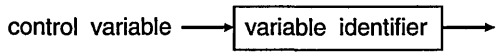
```
while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;
```

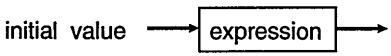
```
while not Eof(InFile) do
begin
  Readln(InFile, Line);
  Process(Line);
end;
```

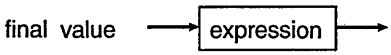
For statements

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









The control variable must be a variable identifier (without any qualifier) that signifies a variable declared to be local 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 is not 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 is not executed.

It's an error if the contained statement alters the value of the control variable. 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;
```

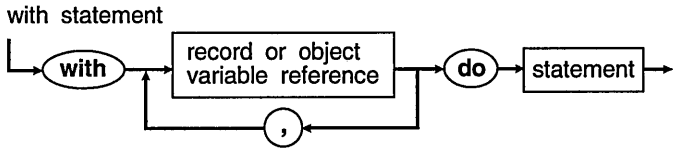
where *Temp1* and *Temp2* are auxiliary variables of the host type of the variable *V* and don't occur elsewhere in the program.

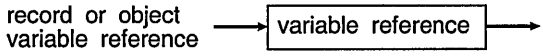
Examples of **for** statements follow:

```
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, methods, constructor, and destructor 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:





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

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

This is equivalent to

```
if Date.Month = 12 then
begin
```

```
    Date.Month := 1;
    Date.Year := Date.Year + 1
  end
  else
    Date.Month := Date.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 is always interpreted as such, even if a variable with the same name is also accessible. Suppose the following declarations have been made:

```
type
  Point = record
    X, Y: Integer;
  end;
var
  X: Point;
  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 *Point*, 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 is 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

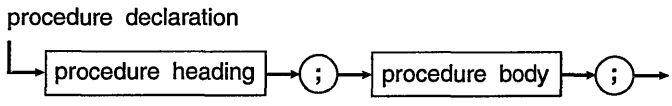
Procedures and functions

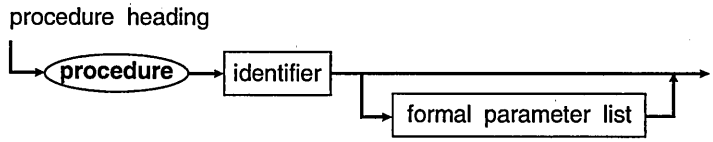
Procedures and functions allow you to nest additional blocks in the main program block. Each procedure or function declaration has a heading followed by a block. A procedure is activated by a procedure statement; a function is activated by the evaluation of an expression that contains its call and returns a value to that expression.

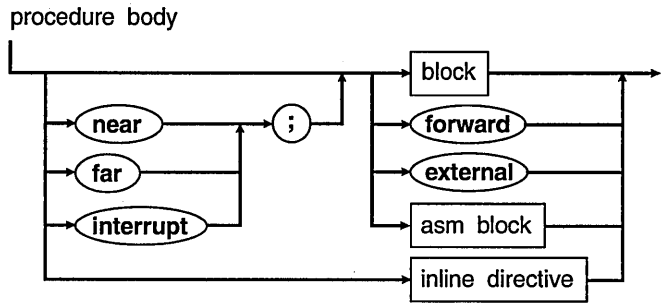
This chapter discusses the different types of procedure and function declarations and their parameters.

Procedure declarations

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







The syntax for a formal parameter list is shown in the section "Parameters" on page 105.

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 any actual parameters required. 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 full in Chapter 18, "Control issues."

Turbo Pascal supports two *procedure call models*: near and far. In terms of code size and execution speed, the near call model is the more efficient, but it carries the restriction that near procedures can only be called from within the module in which they are declared. Far procedures, on the other hand, can be called from any module, but the code for a far call is slightly less efficient.

Turbo Pascal will automatically select the correct call model based on a procedure's declaration: Procedures declared in the **interface** part of a unit use the far call model—they can be called from other modules. Procedures 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 specific purposes, a procedure may be required to use the far call model. For example, in an overlaid application, all procedures and functions are generally required to be far; likewise, 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 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

Interrupt procedures are described in full in Chapter 18, "Control issues."

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

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

Instead of the block in a procedure or function declaration, you can write a **forward, external,** or **inline** declaration.

Forward declarations

A procedure 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—a procedure declaration that uses the same procedure identifier but omits the formal parameter list and includes a block. 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. Mutual recursion is thus possible.

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

An example of a **forward** declaration follows:

```
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 defining declaration can be an **external** or **assembler** declaration; however, it cannot be a **near, far,** or **inline** declaration or another **forward** declaration. Likewise, the defining declaration cannot specify a **near, far,** or **interrupt** directive.

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

External declarations

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

External declarations allow you to 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.

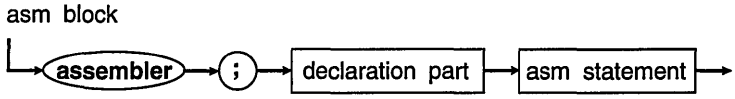
Examples of **external** procedure declarations follow:

```
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 further details on assembler procedures and functions, refer to Chapter 22, "The inline assembler."

Assembler declarations let you write entire procedures and functions in inline assembler.



Inline declarations

Inline procedures are described in full in Chapter 22, "The inline assembler."

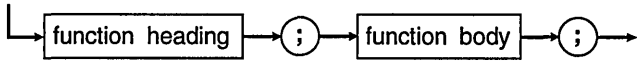
The **inline** directive permits you to write machine code instructions instead of the block. When a normal procedure is called, the compiler generates code that pushes the procedure's parameters onto the stack, and then generates a **CALL** instruction to call the procedure. When you "call" an **inline** procedure, the compiler generates code from the inline directive instead of the **CALL**. Thus, an **inline** procedure 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 }
```

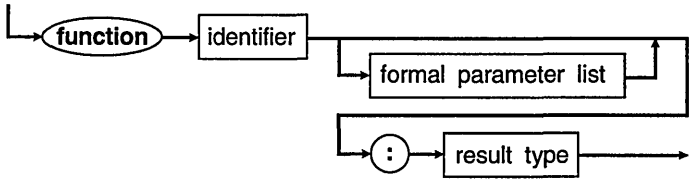
Function declarations

A **function** declaration defines a part of the program that computes and returns a value.

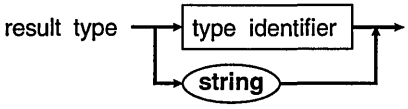
function declaration



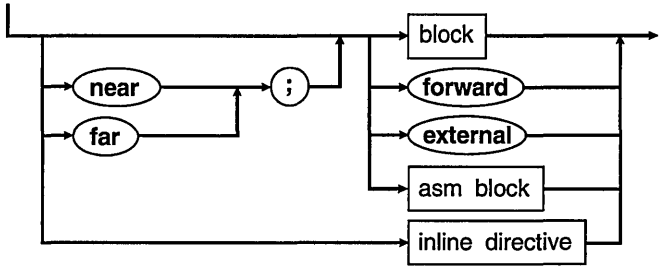
function heading



result type



function body



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 any actual

parameters 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 is not executed, the value returned by the function is unspecified.

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**; however, **interrupt** functions are *not* allowed.

Method declarations

The declaration of a method within an object type corresponds to a **forward** declaration of that method. Thus, 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 declaration, with the exception that the procedure or function identifier in this case is a qualified 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.

A method's defining declaration can optionally repeat the formal parameter list of the method heading in the object type. The defining declaration's method heading must in that case 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. Within the method block, *Self* represents the instance whose method component was designated to activate the method. Thus, 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:

```
procedure Rect.Intersect(var R: Rect);
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 Field.Display;
begin
  GotoXY(X, Y);
  Write(Name^, ' ', GetStr);
end;

function NumField.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 instantiated objects. Typically, the initialization is based on values passed as parameters to the constructor. Constructors cannot be virtual, because the virtual method dispatch mechanism depends on a constructor first having initialized the object.

Here are some examples of constructors:

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

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

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

The first action of a constructor of a descendant type, such as the preceding *StrField.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 any pointer fields in the object.

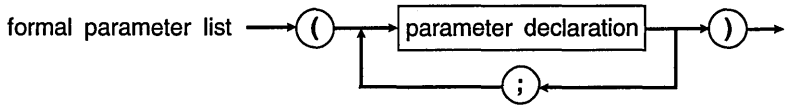
Here are some examples of destructors:

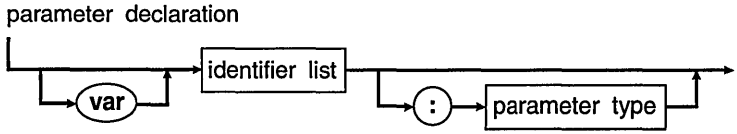
```
destructor Field.Done;
begin
  FreeMem(Name, Length(Name^) + 1);
end;

destructor StrField.Done;
begin
  FreeMem(Value, Len);
  Field.Done;
end;
```

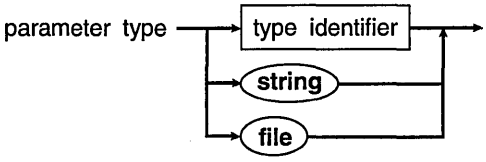
A destructor of a descendant type, such as the preceding *StrField.Done*, typically first disposes the pointer fields introduced in the descendant, and then, as its last action, calls the corresponding destructor of its immediate ancestor to dispose any inherited pointer fields of the object.

Parameters

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







There are three kinds of parameters: *value, variable,* and *untyped variable.* They 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 **var** and followed by a type is a list of variable parameters.
- A parameter group preceded by **var** and not followed by a type is a list of untyped variable parameters.

Value parameters

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

Variable parameters

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

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The type of the actual parameter must be identical to the type of the formal variable parameter (you can bypass this restriction through untyped variable parameters). If the formal parameter type is **string**, it is given the length attribute 255, and the actual variable parameter must be a string type with a length attribute of 255.

File types can only be passed as variable parameters.

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

Objects

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, the *Field.Copy* method might be passed an instance of *Field*, *StrField*, *NumField*, *ZipField*, or any other instance of a descendant of *Field*.

Untyped variable parameters

When a formal parameter is an untyped variable parameter, the corresponding actual parameter may be any variable reference, regardless of its type.

Within the procedure or function, the untyped variable 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.

An example of untyped variable parameters follows:

```
function Equal(var Source, Dest; Size: Word): Boolean;
type
  Bytes = array[0..MaxInt] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Size) and (Bytes(Dest)[N] <> Bytes(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
  Vector = array[1..10] of Integer;
  Point  = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: Vector;
  N: Integer;
  P: Point;
```

then the function calls

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

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

Procedural types

Procedural types are defined in Chapter 3, "Types."

As an extension to standard Pascal, Turbo Pascal allows procedures and functions to be treated as objects that can be assigned to variables and passed as parameters; *procedural types* make this possible.

Procedural variables

Once a procedural type has been defined, it becomes possible to declare variables of that type. Such variables are called *procedural variables*. For example, given the preceding type declarations, the following variables can be declared:

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

Like an integer variable that can be assigned an integer value, a procedural variable can be assigned a *procedural value*. Such a value can of course be another procedural variable, but it can also be a procedure or a function identifier. In this context, 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 procedure and function declarations,

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

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

The variables *P* and *F* declared previously can now be assigned values:

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

Following these assignments, the call `P(I, J)` is equivalent to `Swap(I, J)`, and `F(X)` is equivalent to `Tan(X)`.

As in any other assignment operation, the variable on the left and the value on the right must be assignment-compatible. To be considered assignment-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. As mentioned previously, parameter names are of no significance when it comes to procedural-type compatibility.

In addition to being of a compatible type, a procedure or function must satisfy the following requirements if it is to be assigned to a procedural variable:

- It must be declared with a **far** directive or compiled in the {**$F+**} state.
- It cannot be
  - a standard procedure or function
  - a nested procedure or function
  - an **inline** procedure or function
  - an **interrupt** procedure or function

Standard procedures and functions are the procedures and functions declared by the *System* unit, such as *Writeln*, *Readln*, *Chr*, and *Ord*. To use a standard procedure or function with a procedural variable, you will have to write a "shell" around it. For example, given the procedural type

```
type
  IntProc = procedure(N: Integer);
```

the following is an assignment-compatible procedure to write an integer:

```
procedure WriteInt(Number: Integer); far;
begin
  Write(Number);
end;
```

Nested procedures and function cannot be used with procedural variables. A procedure or function is nested when it is declared within another procedure or function. In the following example, *Inner* is nested within *Outer,* and *Inner* cannot therefore be assigned to a procedural variable.

```
program Nested;
procedure Outer;
procedure Inner;
begin
  Writeln('Inner is nested');
end;
begin
  Inner;
end;
begin
  Outer;
end.
```

The use of procedural types is not restricted to simple procedural variables. Like any other type, a procedural type can participate in the declaration of a structured type, as demonstrated by the following declarations:

```
type
  GotoProc  = procedure(X, Y: Integer);
  ProcList  = array[1..10] of GotoProc;
  WindowPtr = ^WindowRec;
  WindowRec = record
    Next: WindowPtr;
    Header: string[31];
    Top, Left, Bottom, Right: Integer;
    SetCursor: GotoProc;
  end;
var
  P: ProcList;
  W: WindowPtr;
```

Given the preceding declarations, the following statements are valid procedure calls:

```
P[3](1, 1);
W^.SetCursor(10, 10);
```

When a procedural value is assigned to a procedural variable, what physically takes place is that the address of the procedure is stored in the variable. In fact, a procedural variable is much like a pointer variable, except that instead of pointing to data, it points to a procedure or function. Like a pointer, a procedural variable occupies 4 bytes (two words), containing a memory address. The first word stores the offset part of the address, and the second word stores the segment part.

Procedural-type parameters

Since procedural types are allowed in any context, it is possible to declare procedures or functions that take procedures or functions as parameters. The following program demonstrates the use of a procedural-type parameter to output three tables of different arithmetic functions:

```
program Tables;

type
  Func = function(X, Y: Integer): Integer;

function Add(X, Y: Integer): Integer; far;
begin
  Add := X + Y;
end;

function Multiply(X, Y: Integer): Integer; far;
begin
  Multiply := X * Y;
end;

function Funny(X, Y: Integer): Integer; far;
begin
  Funny := (X + Y) * (X - Y);
end;

procedure PrintTable(W, H: Integer; Operation: Func);
var
  X, Y: Integer;
begin
  for Y := 1 to H do
  begin
    for X := 1 to W do
      Write(Operation(X, Y):5);
    Writeln;
  end;
  Writeln;
end;
```

```
begin
  PrintTable(10, 10, Add);
  PrintTable(10, 10, Multiply);
  PrintTable(10, 10, Funny);
end.
```

When run, the *Tables* program outputs three tables. The second one looks like this:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Procedural-type parameters are particularly useful in situations where a certain common action is to be carried out on a set of procedures or functions. In this case, the *PrintTable* procedure represents the common action to be carried out on the *Add*, *Multiply*, and *Funny* functions.

If a procedure or function is to be passed as a parameter, it must conform to the same type-compatibility rules as in an assignment. Thus, such procedures and functions must be declared with a **far** directive, they cannot be built-in routines, they cannot be nested, and they cannot be declared with the **inline** or **interrupt** attributes.

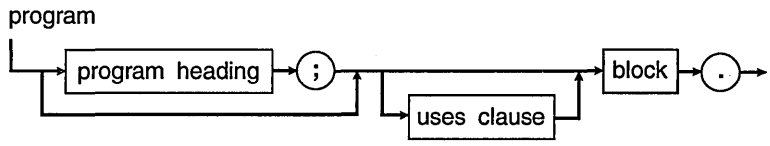
C H A P T E R

9

Programs and units

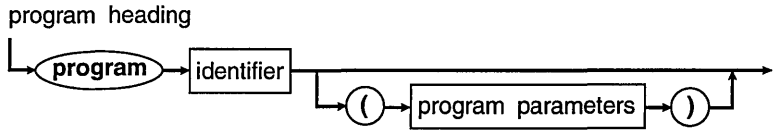
Program syntax

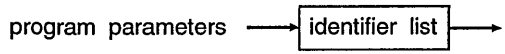
A Turbo Pascal program takes the form of a procedure declaration except for its heading and an optional **uses** clause.



The program heading

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

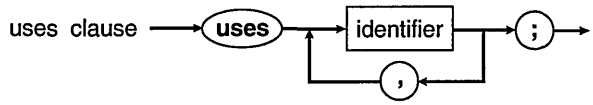




The program heading, if present, is purely decorative and is ignored by the compiler.

The uses clause

The **uses** clause identifies all units used by the program, including units used directly and units used by those units.



The *System* unit is always used automatically. *System* implements all low-level, run-time support routines to support such features as file I/O, string handling, floating point, dynamic memory allocation, and others.

Apart from *System*, Turbo Pascal implements many standard units, such as *Printer*, *Dos*, and *Crt*. These are not used automatically; you must include them in your **uses** clause, for instance,

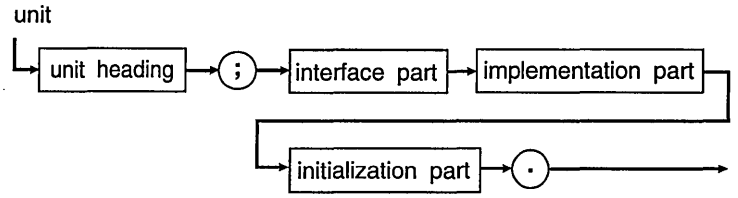
```
uses Dos,Crt;                { Can now access facilities in Dos and Crt }
```

See "The initialization part" on page 117.

The order of the units listed in the **uses** clause determines the order of their initialization.

Unit syntax

Units are the basis of modular programming in Turbo Pascal. They are 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.



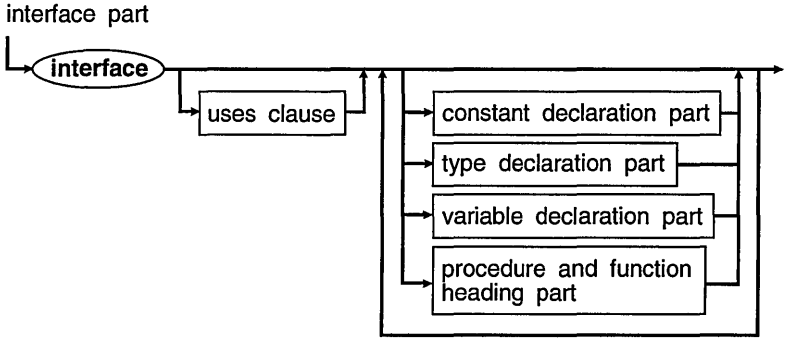
The unit heading

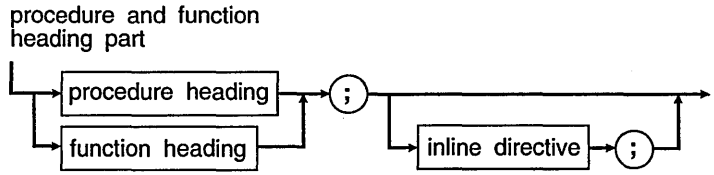
The unit heading specifies the unit's name.

The unit name is used when referring to the unit in a **uses** clause. The name must be unique—two units with the same name cannot be used at the same time.

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.

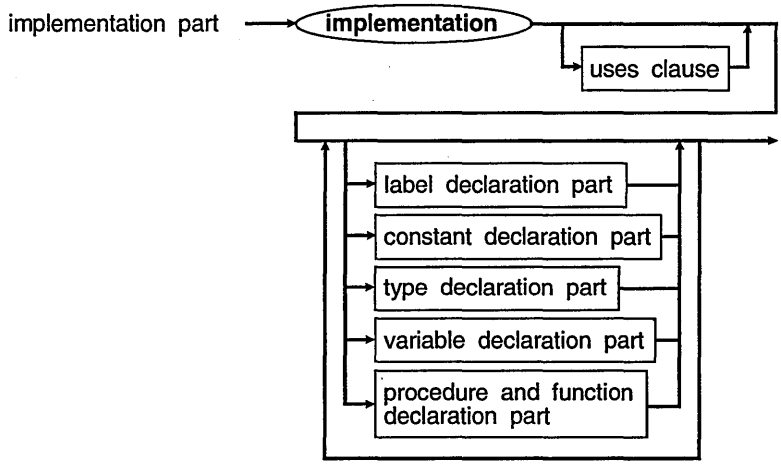




Unless a procedure or function is **inline**, the interface part only lists the procedure or function heading. The block of the procedure or function follows in the implementation part.

The implementation part

The implementation part defines the block of all public procedures and functions. In addition, it declares constants, types, variables, procedures, and functions that are *private*, that is, not available to the host.

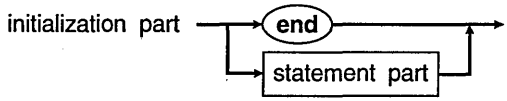


In effect, the procedure and function declarations in the interface part are like **forward** declarations, although the **forward** directive is not specified. Therefore, these procedures and functions can be defined and referenced in any sequence in the implementation part.

Procedure and function headings can be duplicated from the interface part. You don't have to specify the formal parameter list, but if you do, the compiler will issue a compile-time error if the interface and implementation declarations don't match.

The initialization part

The initialization part is the last part of a unit. It consists either of the reserved word **end** (in which case the unit has no initialization code) or of a statement part to be executed in order 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 example:

```
program Prog;
uses Unit2;
const a = b;
begin
end.

unit Unit2;
interface
uses Unit1;
const b = c;
implementation
end.

unit Unit1;
interface
const c = 1;
implementation
const d = 2;
end.
```

In the previous example, *Unit2* is directly dependent on *Unit1*, and *Prog* is directly dependent on *Unit2*. Furthermore, *Prog* is indirectly dependent on *Unit1* (through *Unit2*), even though none of the identifiers declared in *Unit1* are available to *Prog*.

In order to compile a module, Turbo Pascal must be able to locate all units upon which the module depends (directly or indirectly). So, to compile *Prog* above, the compiler must be able to locate both *Unit1* and *Unit2*, or else an error occurs.

When changes are made in the interface part of a unit, other units using the unit must be recompiled. However, if changes are only made 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; changing the implementation part (for example, $d = 1$) doesn't require recompilation of *Unit2*.

When a unit is compiled, Turbo Pascal computes a *unit version number*, which is basically a checksum of the interface part. 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 do not match, indicating that a change was made in the interface part of *Unit1* since *Unit2* was compiled, the compiler shows an error or recompiles *Unit2*, depending on the mode of compilation.

Circular unit references

Placing a **uses** clause in the implementation section of a unit allows you to further hide the inner details of the unit, since units used in the implementation section are not visible to users of the unit. More importantly, however, it also enables you to construct mutually dependent units.

The following program demonstrates how two units can "use" each other. The main program, *Circular*, uses a unit named *Display*. *Display* contains one routine in its interface section, *WriteXY*, which takes three parameters: an (x, y) coordinate pair and a text message to display. If the (x, y) coordinates are onscreen, *WriteXY* moves the cursor to (x, y) and displays the message there; otherwise, it calls a simple error-handling routine.

So far, there's nothing fancy here—*WriteXY* is taking the place of *Write*. Here's where the circular unit reference enters in: How is

the error-handling routine going to display its error message? By using *WriteXY* again. Thus you have *WriteXY*, which calls the error-handling routine *ShowError*, which in turn calls *WriteXY* to put an error message onscreen. If your head is spinning in circles, let's look at the source code to this example, so you can see that it's really not that tricky.

The main program, *Circular*, clears the screen and makes three calls to *WriteXY*:

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

uses
  Crt, Display;

begin
  ClrScr;
  WriteXY(1, 1, 'Upper left corner of screen');
  WriteXY(100, 100, 'Way off the screen');
  WriteXY(81 - Length('Back to reality'), 15, 'Back to reality');
end.
```

Look at the (*x*, *y*) coordinates of the second call to *WriteXY*. It's hard to display text at (100, 100) on an 80x25 line screen. Next, let's see how *WriteXY* works. Here's the source to the *Display* unit, which contains the *WriteXY* procedure. If the (*x*, *y*) coordinates are valid, it displays the message; otherwise, *WriteXY* displays an error message:

```
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 *ShowError* procedure called by *WriteXY* is declared in the following code in the *Error* unit. *ShowError* always displays its error message on the 25th line of the screen:

```
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.
```

Notice that the **uses** clause in the **implementation** sections of both *Display* and *Error* refer to each other. These two units can refer to each other in their **implementation** sections because Turbo Pascal can compile complete **interface** sections for both. In other words, the Turbo Pascal compiler will accept a reference to partially compiled unit *A* in the **implementation** section of unit *B*, as long as both *A* and *B*'s **interface** sections do not depend upon each other (and thus follow Pascal's strict rules for declaration order).

Sharing other declarations

What if you want to modify *WriteXY* and *ShowError* to take an additional parameter that specifies a rectangular window onscreen:

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

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

Remember these two procedures are in separate units. Even if you declared *WindData* in the **interface** of one, there would be no legal way to make that declaration available to the **interface** of the other. The solution is to declare a third module 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.
```

In addition to modifying the code to *WriteXY* and *ShowError* to make use of the new parameter, the **interface** sections of both the *Display* and *Error* units can now "use" *WindData*. This approach is legal because unit *WindData* has no dependencies in its **uses** clause, and units *Display* and *Error* refer to each other only in their respective **implementation** sections.

PART 2

The standard libraries

C H A P T E R

10

The System unit

The *System* unit is Turbo Pascal's run-time library. It implements low-level, run-time support routines for all built-in features, such as file I/O, string handling, 8087 emulation, floating point, overlay management, and dynamic memory allocation. The *System* unit is used automatically by any unit or program, and need never be referred to in a **uses** clause.

Standard procedures and functions

For more detailed information, refer to Chapter 1 in the Library Reference.

This section briefly describes all the standard (built-in) procedures and functions in Turbo Pascal, except for the I/O procedures and functions discussed in the next section beginning on page 129.

Standard procedures and functions are predeclared. Since all predeclared entities act as if they were declared in a block surrounding the program, no conflict arises from a declaration that redefines the same identifier within the program.

Flow control procedures

Procedure	Description
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.

Dynamic allocation procedures

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" of Chapter 16, "Memory issues."

Procedure	Description
Dispose	Disposes a dynamic variable.
FreeMem	Disposes 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.
Mark	Records the state of the heap in a pointer variable.
New	Creates a new dynamic variable and sets a pointer variable to point to it.
Release	Returns the heap to a given state.

Dynamic allocation functions

Function	Description
MaxAvail	Returns the size of the largest contiguous free block in the heap, corresponding to 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.

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 type Real value to a type Longint value.
Trunc	Truncates a type Real value to a type Longint value.

Arithmetic functions

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.

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

Ordinal procedures

Procedure	Description
Dec	Decrements a variable.
Inc	Increments a variable.

Ordinal functions

Function	Description
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

Procedure	Description
Delete	Deletes a substring from a string.
Insert	Inserts a substring into a string.
Str	Converts a numeric value to its string representation.
Val	Converts the string value to its numeric representation.

String functions

Function	Description
Concat	Concatenates a sequence of strings.
Copy	Returns a substring of a string.
Length	Returns the dynamic length of a string.
Pos	Searches for a substring in a string.

Pointer and address functions

Function	Description
Addr	Returns the address of a specified object.
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 procedures

Procedure	Description
FillChar	Fills a specified number of contiguous bytes with a specified value.
Move	Copies a specified number of contiguous bytes from a source range to a destination range.
Randomize	Initializes the built-in random generator with a random value.

Miscellaneous functions

Function	Description
Hi	Returns the high-order byte of the argument.
Lo	Returns the low-order byte of the argument.
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.
SizeOf	Returns the number of bytes occupied by the argument.
Swap	Swaps the high- and low-order bytes of the argument.
UpCase	Converts a character to uppercase.

File input and output

This section briefly describes the standard (or built-in) input and output (I/O) procedures and functions of Turbo Pascal. For more detailed information, refer to Chapter 19.

An introduction to file I/O

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 "Structured types" in Chapter 3.

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. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

The standard text-file variables *Input* and *Output* are opened automatically when program execution begins. *Input* is a read-only file associated with the keyboard and *Output* is a write-only file associated with the display.

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. However, 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 closing a file completely, 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 does not cause the program to halt. To check the result of an I/O operation, you must instead call the standard function *IOResult*.

I/O functions

Function	Description
Eof	Returns the end-of-file status of a file.
FilePos	Returns the current file position of a file. Not used for text files.
FileSize	Returns the current size of a file. Not used for text files.
IOResult	Returns an integer value that is the status of the last I/O function performed.

I/O procedures

Procedure	Description
Assign	Assigns the name of an external file to a file variable.
ChDir	Changes the current directory.
Close	Closes an open file.
Erase	Erases an external file.
GetDir	Returns the current directory of a specified drive.
MkDir	Creates a subdirectory.
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 file to a specified component. Not used with text files.
Truncate	Truncates the file size at the current file position. Not used with text files.

Text files

This section summarizes input and output using file variables of the standard type *Text*. Note that in Turbo Pascal the type *Text* is distinct from the type **file of** Char.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a linefeed character).

For text files, there are special forms of *Read* and *Write* that allow you to read and write values that are not of type Char. Such values are automatically translated to and from their character representation. For example, *Read(F, I)*, where *I* is a type Integer variable, will read a sequence of digits, interpret that sequence as a decimal integer, and store it in *I*.

As noted previously there are 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), and 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);
```

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

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

Some of the standard procedures and functions listed in this section need not have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* are assumed by default, depending on whether the procedure or function is input- or output-oriented. For instance, *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 procedures or functions in this section, the file must have been associated with an external file using *Assign*, and opened using *Reset*, *Rewrite*, or *Append*. An error message is generated 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.

Procedures

Procedure	Description
Append	Opens an existing file for appending.
Flush	Flushes the buffer of an output file.
Read	Reads one or more values from a text file into one or more variables.
Readln	Does what a *Read* does and then skips to the beginning of the next line in the file.
SetTextBuf	Assigns an I/O buffer to a text file.
Write	Writes one or more values to a text file.
Writeln	Does the same as a *Write*, and then writes an end-of-line marker to the file.

Functions

Function	Description
Eoln	Returns the end-of-line status of a file.
SeekEof	Returns the end-of-file status of a file.
SeekEoln	Returns the end-of-line status of a file.

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. The preferred record size is 1, because that is the only value that correctly reflects the exact size of any file (no partial records are possible when the record size is 1).

Except for *Read* and *Write*, all typed file standard procedures and functions are also allowed on untyped files. Instead of *Read* and *Write*, two procedures called *BlockRead* and *BlockWrite* are used for high-speed data transfers.

Procedures

Procedure	Description
BlockRead	Reads one or more records into a variable.
BlockWrite	Writes one or more records from a variable.

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. However, for all versions, the following modes are defined:

0: Read only
1: Write only
2: Read/Write

DOS version 3.x defines additional modes, which are primarily concerned with file-sharing on networks. (For further details on these, refer to 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 is not 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.

Each of the DOS devices is described in the next section. Other DOS implementations can provide additional devices, and still others cannot provide all the ones described here.

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 is not 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 is usually referred to as LPT1, for which the synonym PRN 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

In addition to the CRT device, Turbo Pascal allows you to write your own text file device drivers. A full description of this is given in the section "Text file device drivers" in Chapter 19, "Input and output issues."

Text file devices are used to implement devices unsupported by DOS or to make available another set of features other than those provided by a similar DOS device. A good example of a text file device is the CRT device implemented by the *Crt* standard unit. Its main function is to provide an interface to the display and the keyboard, just like the CON device in DOS. However, the CRT device is much faster and supports such invaluable features as color and windows.

Contrary to 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 instance, the *Crt* standard unit implements an *AssignCrt* procedure that associates text files with the CRT device.

Predeclared variables

Besides procedures and functions, the *System* unit provides a number of predeclared variables.

Uninitialized variables

Variable	Type	Description
Input	Text	Input standard file
Output	Text	Output standard file
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

Initialized variables

Variable	Type	Initial value	Description
ErrorAddr	Pointer	**nil**	Run-time error address
ExitCode	Integer	0	Exit code
ExitProc	Pointer	**nil**	Exit procedure
FileMode	Byte	2	File open mode
FreeList	Pointer	**nil**	Free heap block list
HeapEnd	Pointer	**nil**	Heap end
HeapError	Pointer	**nil**	Heap error function
HeapOrg	Pointer	**nil**	Heap origin
HeapPtr	Pointer	**nil**	Heap pointer
InOutRes	Integer	0	I/O result buffer
OvrCodeList	Word	0	Overlay code segment list
OvrDebugPtr	Pointer	**nil**	Overlay debugger hook
OvrDosHandle	Word	0	Overlay DOS handle
OvrEmsHandle	Word	0	Overlay EMS handle
OvrHeapEnd	Word	0	Overlay buffer end
OvrHeapOrg	Word	0	Overlay buffer origin
OvrHeapPtr	Word	0	Overlay buffer pointer
OvrHeapSize	Word	0	Initial overlay buffer size
OvrLoadList	Word	0	Loaded overlays list
PrefixSeg	Word	0	Program segment prefix
RandSeed	Longint	0	Random seed
StackLimit	Word	0	Minimum stack pointer
Test8087	Byte	0	8087 test result

The *Overlay* unit uses *OvrCodeList*, *OvrHeapSize*, *OvrDebugPtr*, *OvrHeapOrg*, *OvrHeapPtr*, *OvrHeapEnd*, *OvrLoadList*, *OvrDosHandle*, and *OvrEmsHandle* to implement Turbo Pascal's overlay manager. The overlay buffer resides between the stack segment and the heap, and *OvrHeapOrg* and *OvrHeapEnd* contain its starting and ending segment addresses. 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 manager uses *HeapOrg*, *HeapPtr*, *HeapEnd*, *FreeList*, and *HeapError* to implement Turbo Pascal's dynamic memory allocation routines. The heap manager is described in full in Chapter 16, "Memory issues."

The *ExitProc*, *ExitCode*, and *ErrorAddr* variables implement exit procedures. This is also described in Chapter 18, "Control issues."

PrefixSeg is a Word variable containing the segment address of the Program Segment Prefix (PSP) created by DOS when the program was executed. For a complete description of the PSP, refer to your DOS manual.

StackLimit contains the offset of the bottom of the stack in the stack segment, corresponding to the lowest value the SP register is allowed to assume before it is considered a stack overflow. By default, *StackLimit* is zero, but in a program compiled with {**$N+,E+**}, the 8087 emulator will set it to 224 to reserve workspace at the bottom of the stack segment if no 8087 is present in the system.

The built-in I/O routines use *InOutRes* to store the value that the next call to the *IOResult* standard function will return.

RandSeed stores the built-in random number generator's seed. By assigning a specific value to *RandSeed*, the *Random* function can be made to generate a specific sequence of random numbers over and over. This is useful is applications that deal with data encryption, statistics, and simulations.

For further details, see page 133.

The *FileMode* variable allows you to change the access mode in which typed and untyped files are opened (by the *Reset* standard procedure).

For further details, refer to Chapter 14, "Using the 8087."

Test8087 stores the result of the coprocessor autodetection logic, which is executed at startup in a program compiled with {**$N+**}.

For further details, see page 129.

Input and *Output* are the standard I/O files required by every Pascal implementation. By default, they refer to the standard input and output files in DOS.

The *System* unit takes over several interrupt vectors. Before installing its own interrupt handling routines, *System* stores the old vectors in the *SaveIntXX* variables.

Note that the *System* unit contains an INT 24 handler for trapping critical errors. In a Turbo Pascal program, a DOS critical error is treated like any other I/O error: In the {**$I+**} state, the program

halts with a run-time error, and, in the {**$I-**} state, *IOResult* returns a nonzero value.

Here's a skeleton program that restores the original vector, and thereby the original critical error-handling logic:

```
program Restore;
uses Dos;
begin
  SetIntVec($24, SaveInt24);
  ...
end.
```

For more information, refer to the entry on SwapVectors in Chapter 1 of the Library Reference.

The *SwapVectors* routine in the *Dos* unit swaps the contents of the *SaveIntXX* variables with the current contents of the interrupt vectors. *SwapVectors* should be called just before and just after a call to the *Exec* routine, to ensure that the *Exec*'d process does not use any interrupt handlers installed by the current process, and vice versa.

C H A P T E R

11

The Dos unit

The *Dos* unit implements a number of very useful operating system and file-handling routines. None of the routines in the *Dos* unit are defined by standard Pascal, so they have been placed in their own module.

For a complete description of DOS operations, refer to the IBM DOS technical manual.

Constants, types, and variables

Each of the constants, types, and variables defined by the *Dos* unit are briefly discussed in this section. For more detailed information, see the descriptions of the procedures and functions that depend on these objects in Chapter 1, "Run-time library," in the *Library Reference.*

Constants

Flag constants

The following constants test individual flag bits in the Flags register after a call to *Intr* or *MsDos*:

Constants	Value
FCarry	$0001
FParity	$0004
FAuxiliary	$0010
FZero	$0040
FSign	$0080
FOverflow	$0800

For instance, if *R* is a register record, the tests

```
R.Flags and FCarry <> 0
R.Flags and FZero = 0
```

are True respectively if the Carry flag is set and if the Zero flag is clear.

File mode constants

The file-handling procedures use these constants when opening and closing disk files. The mode fields of Turbo Pascal's file variables will contain one of the values specified in the following:

Constant	Value
fmClosed	$D7B0
fmInput	$D7B1
fmOutput	$D7B2
fmInOut	$D7B3

File attribute constants

These constants test, set, and clear file attribute bits in connection with the *GetFAttr*, *SetFAttr*, *FindFirst*, and *FindNext* procedures:

Constant	Value
ReadOnly	$01
Hidden	$02
SysFile	$04
VolumeID	$08
Directory	$10
Archive	$20
AnyFile	$3F

The constants are additive, that is, the statement

```
FindFirst('*.*', ReadOnly + Directory, S);
```

will locate all normal files as well as read-only files and subdirectories in the current directory. The *AnyFile* constant is simply the sum of all attributes.

Types

File record types

The record definitions used internally by Turbo Pascal are also declared in the *Dos* unit. *FileRec* is used for both typed and untyped files, while *TextRec* is the internal format of a variable of type text.

```
type
{ Typed and untyped files }
  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;

{ Textfile record }
  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;
```

The Registers type

The *Intr* and *MsDos* procedures use variables of type *Registers* to specify the input register contents and examine the output register contents of a software interrupt.

```
type
  Registers = record
    case Integer of
    0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
    1: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
```

```
end;
```

Notice the use of a variant record to map the 8-bit registers on top of their 16-bit equivalents.

The DateTime type

Variables of *DateTime* type are used in connection with the *UnpackTime* and *PackTime* procedures to examine and construct 4-byte, packed date-and-time values for the *GetFTime*, *SetFTime*, *FindFirst*, and *FindNext* procedures.

```
type
  DateTime = record
    Year,Month,Day,Hour,Min,Sec: Word;
  end;
```

Valid ranges are *Year* 1980..2099, *Month* 1..12, *Day* 1..31, *Hour* 0..23, *Min* 0..59, and *Sec* 0..59.

The SearchRec type

The *FindFirst* and *FindNext* procedures use variables of type *SearchRec* to scan directories.

```
type
  SearchRec = record
    Fill: array[1..21] of Byte;
    Attr: Byte;
    Time: Longint;
    Size: Longint;
    Name: string[12];
  end;
```

The information for each file found by one of these procedures is reported back in a *SearchRec*. The *Attr* field contains the file's attributes (constructed from file attribute constants), *Time* contains its packed date and time (use *UnpackTime* to unpack), *Size* contains its size in bytes, and *Name* contains its name. The *Fill* field is reserved by DOS and should never be modified.

File-handling string types

These string types are are defined by the *Dos* unit to handle file names and paths in connection with the string procedure *FSplit*:

```
ComStr  = string[127];                    { Command-line string }
PathStr = string[79];                   { Full file path string }
DirStr  = string[67];                 { Drive and directory string }
NameStr = string[8];                        { File-name string }
ExtStr  = string[4];                    { File-extension string }
```

Variables

The DosError variable

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

```
var DosError: Integer;
```

The values stored in *DosError* are DOS error codes. A value of 0 indicates no error; other possible error codes include the following:

DOS error code	Meaning
2	File not found
3	Path not found
5	Access denied
6	Invalid handle
8	Not enough memory
10	Invalid environment
11	Invalid format
18	No more files

Procedures and functions

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 written.
GetTime	Returns the current time set in the operating system.
PackTime	Converts a *DateTime* record into a 4-byte, packed date-and-time character Longint used by *SetFTime*. The fields of the *DateTime* record are not range-checked.
SetDate	Sets the current date in the operating system.
SetFTime	Sets the date and time a file was last written.
SetTime	Sets the current time in the operating system.
UnpackTime	Converts a 4-byte, packed date-and-time character Longint returned by *GetFTime*, *FindFirst*, or *FindNext* into an unpacked *DateTime* record.

Interrupt support procedures

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

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.

File-handling procedures

Procedure	Description
FindFirst	Searches the specified (or current) 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*.
FSplit	Splits a file name into its three component parts (directory, file name, and extension).
GetFAttr	Returns the attributes of a file.
SetFAttr	Sets the attributes of a file.

File-handling functions

Function	Description
FExpand	Takes a file name and returns a fully qualified file name (drive, directory, and extension).
FSearch	Searches for a file in a list of directories.

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.

Process-handling functions

Function	Description
DosExitCode	Returns the exit code of a subprocess.

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.

Miscellaneous procedures

Procedure	Description
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.

Miscellaneous functions

Function	Description
DosVersion	Returns the DOS version number.

C H A P T E R

12

The Graph unit

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

Pursuant to the terms of the license agreement, you can 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 TURBO.TPL and the *Graph* unit in GRAPH.TPU. To run a program that uses the *Graph* unit, in addition to your .EXE program, you'll need one or more of the graphics drivers (.BGI files, see the next section). In addition, 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, refer to *RegisterBGIdriver* in Chapter 1 in the *Library 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.

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, which is a new, high-resolution graphics card capable of resolutions up to 1024 x 768 pixels, and a color palette of 256 colors from a list of 256K colors. The driver file name is IBM8514.BGI.

Turbo Pascal cannot 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 x 480 pixels), and IBM8514HI (1024 x 768 pixels). Both mode constants are defined in the interface for GRAPH.TPU.

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. To

allow you to define colors for the IBM 8514, a new routine was added to the BGI library. The routine is defined in GRAPH.TPU as follows:

```
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 may not be used with the IBM 8514 driver (that is, *SetAllPalette, SetPalette, 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.

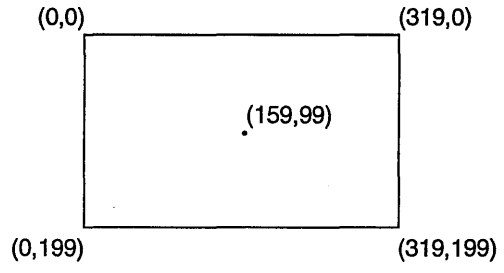
The *FloodFill* routine will not work with the IBM 8514 driver.

These same restrictions apply when also using the VGA in 256-color mode.

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. Thus, 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 12.1
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 is not visible.

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

In text mode, the preceding *Write* statement will leave the cursor in the column immediately following the letter *C*. If the *C* is written in column 80, then the cursor will wrap around to column 1 of the next line. If the *C* is written in column 80 on the 25th line, the entire screen will scroll up one line, and the cursor will be in column 1 of line 25.

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

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

The *MoveTo* command is the equivalent of *GoToXY*. It's 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 bit-mapped font and several "stroked" fonts are included for text output while in graphics mode. A bit-mapped 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 to using a stroked font is apparent when you start to draw large characters. Since a stroked font is defined by vectors, it will still retain good resolution and quality when the font is enlarged.

When a bit-mapped font is enlarged, the matrix is multiplied by a scaling factor and, as the scaling factors becomes larger, the characters' resolution becomes coarser. For small characters, the bit-

mapped font should be 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.

Stroked fonts are each kept in a separate 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 also be linked in or loaded by the user program and "registered" with the *Graph* unit.

A special utility, BINOBJ.EXE, is provided 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 on the distribution disks.)

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 *ViewPort* 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 support routines, including support for multiple graphic pages (EGA, VGA, and Hercules only; especially useful for doing animation), palettes, colors, 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. The error return codes are defined in Table 12.2 on page 160.

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		

Note that *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.

Here's a simple graphics program:

```
1  program GraphTest;
2  uses
3    Graph;
4  var
5    GraphDriver: Integer;
6    GraphMode: Integer;
7    ErrorCode: Integer;
8  begin
9    GraphDriver := Detect;                { Set flag: do detection }
10   InitGraph(GraphDriver, GraphMode, 'C:\DRIVERS');
11   ErrorCode := GraphResult;
12   if ErrorCode <> grOk then                                { Error? }
13   begin
14     Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
15     Writeln('Program aborted...');
16     Halt(1);
17   end;
18   Rectangle(0, 0, GetMaxX, GetMaxY);       { Draw full screen box }
19   SetTextJustify(CenterText, CenterText);          { Center text }
20   SetTextStyle(DefaultFont, HorizDir, 3);
21   OutTextXY(GetMaxX div 2, GetMaxY div 2,      { Center of screen }
22     'Borland Graphics Interface (BGI)');
23   Readln;
24   CloseGraph;
25 end. { GraphTest }
```

The program begins with a call to *InitGraph*, which autodetects the hardware and loads the appropriate graphics driver (located in C:\DRIVERS). If no graphics hardware is recognized or an error occurs during initialization, an error message is displayed and the program terminates. Otherwise, a box is drawn along the edge of the screen and text is displayed 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 autodection and passing *InitGraph* the driver code and a valid graphics mode. To use the AT&T driver, for example, replace lines 9 and 10 in the preceding example with the following three lines of code:

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

This instructs the graphics system to load the AT&T 400 line driver located in C:\DRIVERS 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:

```
1  program GraphTest;
2  uses
3    Graph;
4  var
5    GraphDriver: Integer;
6    GraphMode: Integer;
7    ErrorCode: Integer;
8  begin
9    GraphDriver := Detect;                { Set flag: do detection }
10   InitGraph(GraphDriver, GraphMode, 'C:\DRIVERS');
11   ErrorCode := GraphResult;
12   if ErrorCode <> grOk then                              { Error? }
13   begin
14     Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
15     Writeln('Program aborted...');
16     Halt(1);
17   end;
18   OutText('In Graphics mode. Press <RETURN>');
19   Readln;
20   RestoreCRTMode;
21   Write('Now in text mode. Press <RETURN>');
22   Readln;
23   SetGraphMode(GraphMode);
24   OutText('Back in Graphics mode. Press <RETURN>');
25   Readln;
26   CloseGraph;
27 end. { GraphTest }
```

Note that the *SetGraphMode* call on line 23 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 heap management routines referenced by *GraphGetMemPtr* and *GraphFreeMemPtr* are called by the *Graph* unit 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*, and 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*.

Upon initialization of the *Graph* unit, 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.  { UserHeapManagment }
```

Graph unit constants, types, and variables

Constants

Use these driver and mode constants with *InitGraph*, *DetectGraph*, and *GetModeRange*:

Table 12.1: Graph unit driver and mode constants

Constant	Value	Meaning
Detect	0	Requests autodetection
CGA	1	
MCGA	2	
EGA	3	
EGA64	4	
EGAMono	5	
IBM8514	6	
HercMono	7	
ATT400	8	
VGA	9	
PC3270	10	
CurrentDriver	–128	Passed to *GetModeRange*
ATT400C0	0	320x200 palette 0: LightGreen, LightRed, Yellow; 1 page
ATT400C1	1	320x200 palette 1: LightCyan, LightMagenta, White; 1 page
ATT400C2	2	320x200 palette 2: Green, Red, Brown; 1 page
ATT400C3	3	320x200 palette 3: Cyan, Magenta, LightGray; 1 page
ATT400Med	4	640x200 1 page
ATT400Hi	5	640x400 1 page
CGAC0	0	320x200 palette 0: LightGreen, LightRed, Yellow; 1 page
CGAC1	1	320x200 palette 1: LightCyan, LightMagenta, White; 1 page
CGAC2	2	320x200 palette 2: Green, Red, Brown; 1 page
CGAC3	3	320x200 palette 3: Cyan, Magenta, LightGray; 1 page
CGAHi	4	640x200 1 page
EGALo	0	640x200 16 color 4 page
EGAHi	1	640x350 16 color 2 page
EGA64Lo	0	640x200 16 color 1 page
EGA64Hi	1	640x350 4 color 1 page
EGAMonoHi	3	640x350 64K on card, 1 page; 256K on card, 2 page
HercMonoHi	0	720x348 2 page
IBM8514Lo	0	640x480 256 colors
IBM8514Hi	1	1024x768 256 colors
MCGAC0	0	320x200 palette 0: LightGreen, LightRed, Yellow; 1 page
MCGAC1	1	320x200 palette 1: LightCyan, LightMagenta, White; 1 page
MCGAC2	2	320x200 palette 2: Green, Red, Brown; 1 page
MCGAC3	3	320x200 palette 3: Cyan, Magenta, LightGray; 1 page
MCGAMed	4	640x200 1 page

Table 12.1: Graph unit driver and mode constants (continued)

MCGAHi	5	640x480 1 page
PC3270Hi	0	720x350 1 page
VGALo	0	640x200 16 color 4 page
VGAMed	1	640x350 16 color 2 page
VGAHi	2	640x480 16 color 1 page

The error values returned by *GraphResult* are shown in the following table:

Table 12.2
GraphResult error values

Constant	Value	Description
grOk	0	No error
grNoInitGraph	–1	(BGI) graphics not installed (use *InitGraph*)
grNotDetected	–2	Graphics hardware not detected
grFileNotFound	–3	Device driver file not found
grInvalidDriver	–4	Invalid device driver file
grNoLoadMem	–5	Not enough memory to load driver
grNoScanMem	–6	Out of memory in scan fill
grNoFloodMem	–7	Out of memory in flood fill
grFontNotFound	–8	Font file not found
grNoFontMem	–9	Not enough memory to load font
grInvalidMode	–10	Invalid graphics mode for selected driver
grError	–11	Graphics error (generic error)
grIOerror	–12	Graphics I/O error
grInvalidFont	–13	Invalid font file
grInvalidFontNum	–14	Invalid font number

SetPalette and SetAllPalette

Use these color constants with *SetPalette* and *SetAllPalette*:

Constant	Value
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
LightGray	7
DarkGray	8
LightBlue	9
LightGreen	10
LightCyan	11
LightRed	12
LightMagenta	13
Yellow	14
White	15

SetRGBPalette

These color constants can be used with *SetRGBPalette* to select the standard EGA colors on an IBM 8514 graphics adapter:

Constant	Value
EGABlack	0 (dark colors)
EGABlue	1
EGAGreen	2
EGACyan	3
EGARed	4
EGAMagenta	5
EGABrown	20
EGALightGray	7
EGADarkGray	56 (light colors)
EGALightBlue	57
EGALightGreen	58
EGALightCyan	59
EGALightRed	60
EGALightMagenta	61
EGAYellow	62
EGAWhite	63

Line style constants

Use these constants with *GetLineSettings* and *SetLineStyle*:

Constant	Value
SolidLn	0
DottedLn	1
CenterLn	2
DashedLn	3
UserBitLn	4 (user-defined line style)
NormWidth	1
ThickWidth	3

Font control constants

Use these constants with *GetTextSettings* and *SetTextStyle*:

Constant	Value
DefaultFont	0 (8x8 bit mapped font)
TriplexFont	1 ("stroked" fonts)
SmallFont	2
SansSerifFont	3
GothicFont	4
HorizDir	0 (left to right)
VertDir	1 (bottom to top)
UserCharSize	0 (user-defined Char size)

Justification constants

These constants control horizontal and vertical justification for *SetTextJustify*:

Constant	Value
LeftText	0
CenterText	1
RightText	2
BottomText	0
CenterText	1 (already defined)
TopText	2

Clipping constants

Use these constants with *SetViewPort* to control clipping. With clipping on, graphics output is clipped at the viewport boundaries:

Constant	Value
ClipOn	True
ClipOff	False

Bar constants

These constants may be used with *Bar3D* to specify whether a 3-D top should be drawn on top of the bar (allows for stacking bars and only drawing a top on the topmost bar):

Constant	Value
TopOn	True
TopOff	False

Fill pattern constants

These fill pattern constants are used by *GetFillSettings* and *SetFillStyle*. Use *SetFillPattern* to define your own fill pattern, then call *SetFillStyle(UserFill, SomeColor)* and make your fill pattern the active style (shown in the following):

Constant	Value
EmptyFill	0 (Fills area in background color)
SolidFill	1 (Fills area in solid fill color)
LineFill	2 (—— fill)
LtSlashFill	3 (/// fill)
SlashFill	4 (/// fill with thick lines)
BkSlashFill	5 (\\\ fill with thick lines)
LtBkSlashFill	6 (\\\ fill)
HatchFill	7 (Light hatch fill)
XHatchFill	8 (Heavy cross hatch fill)

InterleaveFill	9 (Interleaving line fill)
WideDotFill	10 (Widely spaced dot fill)
CloseDotFill	11 (Closely spaced dot fill)
UserFill	12 (User-defined fill)

BitBlt operators

Use these BitBlt operators with both *PutImage* and *SetWriteMode*:

Constant	Value
CopyPut	0 (**mov**)
XORPut	1 (**xor**)

These BitBlt operators are used by PutImage *only:*

OrPut	2 (**or**)
AndPut	3 (**and**)
NotPut	4 (**not**)

Palette constant

This constant is used by *GetPalette, GetDefaultPalette, SetAllPalette,* and defines the *PaletteType* record:

Constant	Value
MaxColors	15

Types

This record is used by *GetPalette, GetDefaultPalette,* and *SetAllPalette*:

```
type
  PaletteType = record
    Size: Byte;
    Colors: array[0..MaxColors] of Shortint;
  end;
```

This record is used by by *GetLineSettings*:

```
type
  LineSettingsType = record
    LineStyle: Word;
    Pattern: Word;
    Thickness: Word;
  end;
```

This record is used by by *GetTextSettings*:

```
type
  TextSettingsType = record
    Font: Word;
```

```
    Direction: Word;
    CharSize: Word;
    Horiz: Word;
    Vert: Word;
  end;
```

This record is used by by *GetFillSettings*:

```
type
  FillSettingsType = record
    Pattern: Word;
    Color: Word;
  end;
```

This record is used by *GetFillPattern* and *SetFillPattern*:

```
type
  FillPatternType = array[1..8] of Byte;  { User-defined fill style }
```

This type is defined for your convenience. Note that both fields are of type Integer (not Word):

```
type
  PointType = record
    X, Y: Integer;
  end;
```

This record is used by *GetViewSettings* to report the status of the current viewport:

```
type
  ViewPortType = record
    x1, y1, x2, y2: Integer;
    Clip: Boolean;
  end;
```

This record is used by by *GetArcCoords* and can be used to retrieve information about the last call to *Arc* or *Ellipse*:

```
type
  ArcCoordsType = record
    X, Y: Integer;
    Xstart, Ystart: Integer;
    Xend, Yend: Integer;
  end;
```

Variables

These variables initially point to the Graph unit's heap management routines. If your program does its own heap management, assign the addresses of your allocation and deallocation routines to *GraphGetMemPtr* and *GraphFreeMemPtr*, respectively:

Variable	Value
GraphGetMemPtr	Pointer (steal heap allocation)
GraphFreeMemPtr	Pointer (steal heap deallocation)

Table 12.3: Graph unit procedures

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.
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*.
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*.
GetModeRange	Returns the lowest and highest valid graphics mode for a given driver.
GetPalette	Returns the current palette and its size.

Table 12.3: Graph unit procedures (continued)

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.
GraphDefaults	Homes the current pointer (CP) and resets the graphics 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*).
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.
RestoreCrtMode	Restores the original screen mode before graphics is initialized.
Sector	Draws and fills an elliptical sector.
SetActivePage	Set 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.

Table 12.3: Graph unit procedures (continued)

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.*

Table 12.4: Graph unit functions

GetBkColor	Returns the current background color.
GetColor	Returns the current drawing color.
GetDefaultPalette	Returns the default hardware palette in a record of *PaletteType.*
GetDriverName	Returns a string containing the name of the current driver.
GetGraphMode	Returns the current graphics mode.
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.
GetPaletteSize	Returns the size of the palette color lookup table.
GetPixel	Gets the pixel value at (*x,y*).
GetX	Returns the x-coordinate of the current position (CP).
GetY	Returns the y-coordinate of the current position (CP).
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 is not built into the BGI system.
RegisterBGIdriver	Registers a valid BGI driver with the graphics system.
RegisterBGIfont	Registers a valid BGI font with the graphics system.
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, "Run-time library," in the *Library Reference.*

C H A P T E R

13

The Overlay unit

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, since 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 (non-overlaid) 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 non-overlaid 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 unit is overlaid is made by the program that uses the unit.

When an overlay is loaded into memory, it is 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 may 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 non-overlaid routine. Furthermore, 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 point in 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.
- At any call to an overlaid procedure or function, you must guarantee 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 179. 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.

Important! Failing to observe the FAR call requirement in an overlaid program will cause 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. An example follows:

```
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}
```

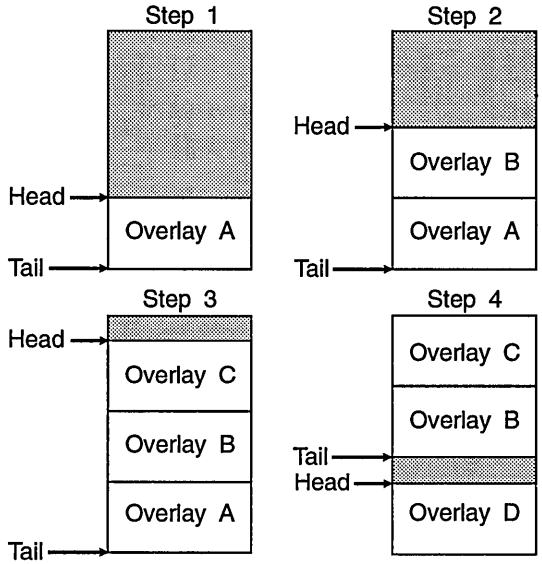
The compiler reports an error if you attempt to overlay a unit that wasn't compiled in the {**$O+**} state. Of the standard units, the only one that can be overlaid is *Dos*; the other standard units, *System*, *Overlay*, *Crt*, *Graph*, *Turbo3*, and *Graph3*, cannot be overlaid. In addition, programs 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 is not enough free space between the head and the tail), overlays are disposed at the tail to make room for new ones.

Since ordinary memory is not circular in nature, the actual implementation of the overlay buffer involves a few more steps in order to make the buffer appear to be a ring. Figure 13.1 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 13.1
Loading and disposing 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 always keep the free area located between the head pointer and the tail pointer. Second, in order to load overlay *D*, the overlay manager has to dispose 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 overlays at the tail to make room for new ones at the head, and each time the head pointer wraps around, the sliding operation is repeated.

This is the default mode of operation for Turbo Pascal 6.0's overlay manager. However, Turbo Pascal also lets you make use of 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* will still occasionally be thrown out of the overlay buffer, only to be reloaded again shortly thereafter. The problem here is that the overlay manager knows nothing about the *frequency* of calls to

routines in *A*—all it knows is that when a call is made to a routine in *A* and *A* is not in memory, 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. Such call interception is unfortunately very costly in terms of execution speed, and may in some cases slow down the application even more than the additional overlay load operations.

Turbo Pascal provides a compromise solution that incurs practically no performance overhead and still maintains a high degree of success in identifying frequently used overlays that shouldn't be unloaded: When an overlay gets close to the tail of the overlay buffer, it is put on "probation." If, during this probationary period, a call is made to a routine in the overlay, it is "reprieved," and will not be disposed when it reaches the tail of the overlay buffer. Instead, it is simply moved to the head of the buffer, and thus gets another free ride around the overlay buffer ring. If, on the other hand, no calls are made to an overlay during its probationary period, indicating less frequent 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 new 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 is automatically put on probation. Any free space in the overlay buffer is considered part of the probation area.

Constants and variables

The constants and variables defined by the *Overlay* unit are briefly discussed in this section.

OvrResult

Before returning, each of the procedures in the *Overlay* unit stores a result code in the *OvrResult* variable.

```
var OvrResult: Integer;
```

The possible return codes are defined in the constant declaration in the next section. In general, a value of zero indicates success.

The *OvrResult* variable resembles the *IOResult* standard function except that *OvrResult* is *not* set to zero once it is accessed. Thus, there is no need to copy *OvrResult* into a local variable before it is examined.

OvrTrapCount

```
var OvrTrapCount: Word;
```

Each time a call to an overlaid routine is intercepted by the overlay manager, either because the overlay is not in memory or because the overlay is on probation, the *OvrTrapCount* variable is incremented. The initial value of *OvrTrapCount* is 0.

OvrLoadCount

```
var OvrLoadCount: Word;
```

Each time an overlay is loaded, the *OvrLoadCount* variable is incremented. The initial value of *OvrLoadCount* is zero.

By examining *OvrTrapCount* and *OvrLoadCount* (for example, in the debugger's Watch window) over identical runs of an application, you can monitor the effect of different probation area sizes (set with *OvrSetRetry)* to find the optimal size for your particular application.

OvrFileMode

```
var OvrFileMode: Byte;
```

The *OvrFileMode* variable determines the access code to pass to DOS when the overlay file is opened. The default *OvrFileMode* is 0, corresponding to read-only access. By assigning a new value to *OvrFileMode* before calling *OvrInit*, you can change the access code; for example, to allow shared access on a network system.

For further details on access code values, refer to your DOS programmer's reference manual.

OvrReadBuf

```
type
  OvrReadFunc = function(OvrSeg: Word): Integer;
var
  OvrReadBuf: OvrReadFunc;
```

The *OvrReadBuf* procedure variable lets you intercept overlay load operations, for example, to implement error handling or to 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, run-time error 209 is generated. The *OvrSeg* parameter indicates what overlay to load, but as you'll see later, you never need to access this information.

Don't attempt to call any overlaid routines from within your overlay read function—such calls will 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* will contain 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 in case of success, or a DOS error code in case of failure. Likewise, the default EMS read function returns 0 in case of success, or an EMS error code (ranging from $80 through $FF) in case of failure. For details on DOS error codes, refer to the "Run-time errors" section in Appendix A of this book. 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.
```

Result codes

Errors in the *Overlay* unit are reported through the *OvrResult* variable. The following result codes are defined as follows:

Table 13.1
OvrResult values

Constant	Value	Meaning
ovrOk	0	Success
ovrError	–1	Overlay manager error
ovrNotFound	–2	Overlay file not found
ovrNoMemory	–3	Not enough memory for overlay buffer
ovrIOError	–4	Overlay file I/O error
ovrNoEMSDriver	–5	EMS driver not installed
ovrNoEMSMemory	–6	Not enough EMS memory

Procedures and functions

The *Overlay* unit defines the procedures *OvrInit*, *OvrInitEMS*, *OvrSetBuf*, *OvrClearBuf*, and *OvrSetRetry*, and the functions *OvrGetBuf* and *OvrGetRetry*. Here's a brief description of each.

OvrInit

```
procedure OvrInit(FileName: string);
```

The OvrInit procedure must be called before any of the other overlay manager procedures.

Initializes the overlay manager and opens the overlay file. If the *FileName* parameter does not specify a drive or a subdirectory, the overlay manager searches for the file in the current directory, in the directory that contains the .EXE file (if running under DOS 3.x), and in the directories specified in the DOS PATH environment variable. Possible error return codes are *ovrError* and *ovrNotFound*. In case of error, the overlay manager remains uninstalled, and an attempt to call an overlaid routine will produce run-time error 208.

OvrInitEMS

```
procedure OvrInitEMS;
```

If possible, loads the overlay file into EMS. If successful, the overlay file is closed, and all subsequent overlay loads are reduced to fast in-memory transfers. Possible error return codes are *ovrError*, *ovrIOError*, *ovrNoEMSDriver*, and *ovrNoEMSMemory*.

The overlay manager will continue to function if *OvrInitEMS* returns an error, but overlays will be read from disk.

Using *OvrInitEMS* to place the overlay file in EMS does not eliminate the need for an overlay buffer. Overlays still have to be copied from EMS into "normal" memory in the overlay buffer before they can be executed. However, since such in-memory transfers are significantly faster than disk reads, the need to increase the size of the overlay buffer becomes less apparent.

OvrSetBuf

```
procedure OvrSetBuf(Size: Longint);
```

Sets the size of the overlay buffer. The specified size must be larger than or equal to the initial size of the overlay buffer, and less than or equal to *MemAvail* plus the current size of the overlay buffer. If the specified size is larger than the current size, additional space is allocated from the beginning of the heap (thus decreasing the size of the heap). Likewise, if the specified size is less than the current size, excess space is returned to the heap. *OvrSetBuf* requires that the heap be empty; an error is returned if dynamic variables have already been allocated using *New* or *GetMem*. Possible error return codes are *ovrError* and *ovrNoMemory*. The overlay manager will continue to function if *OvrSetBuf* returns an error, but the size of the overlay buffer will remain unchanged.

OvrGetBuf

```
function OvrGetBuf: Longint;
```

Returns the current size of the overlay buffer. Initially, the overlay buffer is as small as possible, corresponding to the size of the largest overlay. A buffer of this size is automatically allocated when an overlaid program is executed.

The initial buffer size may be larger than 64K, since it includes both code and fix-up information for the largest overlay.

OvrClearBuf

```
procedure OvrClearBuf;
```

Clears the overlay buffer. All currently loaded overlays are disposed from the overlay buffer, forcing subsequent calls to

overlaid routines to reload the overlays from the overlay file (or from EMS). If *OvrClearBuf* is called from an overlay, that overlay will immediately be reloaded upon return from *OvrClearBuf*. The overlay manager never requires you to call *OvrClearBuf*; in fact, doing so will decrease performance of your application, since it forces overlays to be reloaded. *OvrClearBuf* is solely intended for special use, such as temporarily reclaiming the memory occupied by the overlay buffer.

OvrSetRetry

```
procedure OvrSetRetry(Size: Longint);
```

The *OvrSetRetry* procedure sets the size of the "probation area" in the overlay buffer. If an overlay falls within the *Size* bytes before the overlay buffer tail, it is automatically put on probation. Any free space in the overlay buffer is considered part of the probation area. For reasons of compatibility with earlier versions of the overlay manager, the default probation area size is zero, which effectively disables the probation/reprieval mechanism. Here's an example of how to use *OvrSetRetry*:

```
OvrInit('MYPROG.OVR');
OvrSetBuf(BufferSize);
OvrSetRetry(BufferSize div 3);
```

There is no empirical formula for determining the optimal size of the probationary area—however, experiments have shown that values ranging from one-third to one-half of the overlay buffer size provide the best results.

OvrGetRetry

```
function OvrGetRetry: Longint;
```

The *OvrGetRetry* function returns the current size of the probation area, that is, the value last set with *OvrSetRetry*.

Designing overlaid programs

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

Overlay code generation

Turbo Pascal only allows a unit to be overlaid 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. However, if both units are overlaid, this would not work, since at the call to *WriteStr*, *UnitB*'s code segment may be overwritten by *UnitA*'s, thus rendering the string pointer 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 makes sure to copy all code-segment-based constants into stack temporaries before passing pointers to them.

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

The far call requirement

As mentioned previously, 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 are thus 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, though, 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 of course 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 (TPC.EXE) or the **F**orce Far Calls check box in the **O**ptions | **C**ompiler dialog box. Compared to mixed near and far calls, the added cost of far calls exclusively is usually quite limited: 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. The initialization code must be placed before the first call to an overlaid routine, and would typically be done at the beginning of the program's statement part.

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

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

No error checks are made, so if there is not enough memory for the overlay buffer or if the overlay file was not found, run-time error 208 ("Overlay manager not installed") will occur 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 and, if so, loads the overlay file into EMS.

As mentioned previously, the initial overlay buffer size is as small as possible, or rather, just big enough to contain the largest overlay. This may prove 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 are

overlaid. If the total size of those units is larger than the largest overlay, a substantial amount of swapping will occur if the units make frequent calls to each other.

Obviously, 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 will provide a suitable value.

Using *OvrInitEMS* to place the overlay file in EMS does not 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. However, since as such in-memory transfers are significantly faster than disk reads, the need to increase the size of the overlay buffer becomes less apparent.

Remember, *OvrSetBuf* will expand the overlay buffer by shrinking the heap. Therefore, the heap must be empty or *OvrSetBuf* will have 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 is not 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, since errors in *OvrInit* are fatal. (If they are ignored, a run-time error will occur 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 is not halted. Instead, it will just continue 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 is not enough memory, the overlay manager will just continue to use the minimum buffer that was allocated when the program started.

Initialization sections

Like static units, overlaid units may 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, and the only way to do that is to create an additional non-overlaid unit, which 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, it should be avoided for a number of reasons.

First, the initialization code, even though it is only executed once, is a part of the overlay, and will occupy overlay buffer space whenever the overlay is loaded. Second, if a number of overlaid units have initialization code, each of them will have 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 cannot 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 non-overlay 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.

Likewise, passing overlaid procedures and functions as procedural parameters, and assigning overlaid procedures and functions to procedural type variables is fully supported.

Debugging overlays

Most debuggers have very limited overlay debugging capabilities, if any at all. Not 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 be far and set up a stack frame as the following demonstrates:

```
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 local variables
        pop     bp              ;Restore BP
        ret     ParamSize       ;Return

ExternProc      ENDP
```

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

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 is not itself overlaid, *ExternProc* must still use the far model and still has to set up a stack frame.

In the case where 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 does not have to set up a stack frame.

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

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. Thus in the IDE, make sure that the **S**tandalone option is checked in **O**ptions | De**b**ugger. With the command-line version of the compiler, make sure not to 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, you can use the *ParamStr* standard function to obtain the name of the .EXE file, for example,

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

C H A P T E R

14

Using the 8087

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 8086 processor is designed to easily handle integer values, but it takes considerably more time and effort to handle reals. To improve floating-point performance, the 8086 family of processors has a corresponding family of math coprocessors, the 8087s.

The 8087 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 a lot, you'll probably want a coprocessor.

Turbo Pascal provides optimal floating-point performance whether or not you have an 8087.

- For programs running on any PC, with or without an 8087, 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 2.9×10^{-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 8087 processor.
- If you need the added precision and flexibility of the 8087, you can instruct Turbo Pascal to produce code that uses the 8087 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 **$N** compiler directive or the **8**087/80287 check box in the **O**ptions | **C**ompiler dialog box. The default state is {**$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 8087, giving you increased precision and access to the four additional real types.

Important! 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:

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

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

Even if you don't have an 8087 in your machine, you can instruct Turbo Pascal to include a run-time library that emulates the numeric coprocessor. In that case, if an 8087 is present, it is used. If it's not present, it is emulated by the run-time library, at the cost of running somewhat slower.

The **$E** compiler directive and the **E**mulation check box in the **O**ptions | **C**ompiler dialog box are used to enable and disable 8087 emulation. The default state is {**$E+**}, and in this state, the full 8087 emulator is automatically included in programs that use the 8087. In the {**$E-**} state, a substantially smaller floating-point library is used, and the final .EXE file can only be run on machines with an 8087.

The **$E** compiler directive has no effect if used in a unit; it only applies to the compilation of a program. Furthermore, if the program is compiled in the {**$N-**} state, and if all the units used by the program were compiled with {**$N-**}, then an 8087 run-time library is not required, and the **$E** compiler directive is ignored.

The remainder of this chapter discusses special issues concerning Turbo Pascal programs that use the 8087 coprocessor.

The 8087 data types

For programs that use the 8087, 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 8087. It occupies 10 bytes of memory, providing a range of 3.4×10^{-4932} to 1.1×10^{4932} with 19 to 20 significant digits. Any arithmetic involving real-type values is performed with the range and precision of the Extended type.
- The Comp type stores integral values in 8 bytes, providing a range of $-2^{63}+1$ to $2^{63}-1$, which is approximately -9.2×10^{18} to 9.2×10^{18}. Comp may be compared to a double-precision Longint, but it is considered a real type because all arithmetic done with Comp uses the 8087 coprocessor. Comp is well suited for representing monetary values as integral values of cents or mils (thousandths) in business applications.

Whether or not you have an 8087, the 6-byte Real type is always available, so you need not modify your source code when switching to the 8087, and you can still read data files generated by programs that use software floating point.

Note, however, that 8087 floating-point calculations on variables of type Real are slightly slower than on other types. This is because the 8087 cannot 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 are concerned with optimum speed and never need to run on a system without an 8087, you may 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 8087. 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 instance, will be 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;
```

With no special effort by the programmer, Turbo Pascal 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
    T := T + X[I] * Y[I];
  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 are not 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 will output False:

```
X := 1 / 3;
Y := 1 / 3;
Writeln(X = Y);
```

The reason is that *X* is accurate only to 7 to 8 digits, where *Y* is accurate to 15 to 16 digits, and when both are converted to Extended, they will differ after 7 to 8 digits. Similarly, the statements

```
X := 1 / 3;
Writeln(X = 1 / 3);
```

will output False, since the result of 1/3 in the *Writeln* statement is calculated with 20 significant digits.

The 8087 evaluation stack

The 8087 coprocessor has an internal evaluation stack that can be up to eight levels deep. Accessing a value on the 8087 stack is much faster than accessing a variable in memory; so to achieve the best possible performance, Turbo Pascal uses the 8087's stack for storing temporary results.

In theory, very complicated real-type expressions can cause an 8087 stack overflow. However, this is not likely to occur, since it would require the expression to generate more than eight temporary results.

A more tangible danger lies in recursive function calls. If such constructs are not coded correctly, they can very well cause an 8087 stack overflow.

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 8087 stack overflow for values of *N* larger than 8. The reason is that the calculation of the last assignment requires a temporary on the 8087 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

```
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 of course also overflow, but this would typically require significantly more recursive calls.)

Writing reals with the 8087

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. Likewise, the *Str* standard procedure returns a four-digit exponent when floating-point format is selected.

Units using the 8087

Units that use the 8087 can only be used by other units or programs that are compiled in the {**$N+**} state.

The fact that a unit uses the 8087 is determined by whether it contains 8087 instructions—not by the state of the **$N** compiler directive at the time of its compilation. This makes the compiler more forgiving in cases where you accidentally compile a unit (that doesn't use the 8087) in the {**$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 8087

The Turbo Pascal 8087 run-time library built into your program (compiled with {**$N+**}) includes startup code that automatically detects the presence of an 8087 chip. If an 8087 is available, then the program will use it. If one is not present, the program will use the emulation run-time library. If the program was compiled in the {**$E-**} state, and an 8087 could not be detected at startup, the program displays "Numeric coprocessor required," and terminates.

There are some instances in which you might want to override this default autodetection behavior. For example, your own system may have an 8087, but you want to verify that your program will work as intended on systems without a coprocessor. Or your program may need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that an 8087 is present when it's not, or vice versa).

Turbo Pascal provides an option for overriding the startup code's default autodetection logic; this option is 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 do not want to use the 8087, 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.

Beware! If you set *87* = *Y* when, in fact, there is no 8087 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

```
SET 87 =
```

at the DOS prompt and then press *Enter* immediately.

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 further checks to determine what kind of coprocessor it is (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:

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

Your program may examine the *Test8087* variable to determine the characteristics of the system it is running on. In particular, *Test8087* may be examined to determine whether 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 8087 emulation enabled. For example, if you are using 8087 instructions in assembly language **external** procedures, make sure to enable emulation when you assemble the .ASM files into .OBJ files. Otherwise, the 8087 instructions cannot be emulated on machines without an 8087. Use Turbo Assembler's **/E** command-line switch to enable emulation.

C H A P T E R

15

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 do not 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 videomemory.

The input and output files

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. This corresponds to the following statements being executed 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

```
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 is not 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 screen.

The default window is the entire screen.

Turbo Pascal also supports screen modes for EGA (43 line) and VGA (50 line); see the *TextMode* description in Chapter 15.

Special characters

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

Character	Name	Description
#7	Bell	Emits a beep from the internal speaker.
#8	Backspace	Moves the cursor left one character. If the cursor is already at the left edge of the current window, nothing happens.
#10	Linefeed	Moves the cursor one line down. If the cursor is already at the bottom of the current window, the window scrolls up one line.
#13	Carriage return	Returns the cursor to the left edge of the current window.

All other characters will appear onscreen when written.

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:

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 is False by default.

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

Constants, types, and variables

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

Constants

Crt mode constants

The following constants are used as parameters to the *TextMode* procedure:

Constant	Value	Description
BW40	0	40x25 B/W on color adapter
BW80	2	80x25 B/W on color adapter
Mono	7	80x25 B/W on monochrome adapter
C040	1	40x25 color on color adapter
C080	3	80x25 color on color adapter
Font8x8	256	For EGA/VGA 43 and 50 line
C40	C040	For 3.0 compatibility
C80	C080	For 3.0 compatibility

The C40 and C80 constants are for compatibility with Turbo Pascal version 3.0.

BW40, *C040*, *BW80*, and *C080* represent the four color text modes supported by the IBM PC Color/Graphics Adapter (CGA). The *Mono* constant represents the single black-and-white text mode supported by the IBM PC Monochrome Adapter. *Font8x8* represents EGA/VGA 43- and 50-line modes. *LastMode* returns to the last active text mode after using graphics.

Text color constants

The following constants are used in connection with the *TextColor* and *TextBackground* procedures:

Constant	Value
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
LightGray	7
DarkGray	8
LightBlue	9
LightGreen	10
LightCyan	11
LightRed	12
LightMagenta	13
Yellow	14
White	15
Blink	128

Colors are represented by the numbers between 0 and 15; to easily identify each color, you can use these constants instead of numbers. In the color text modes, the foreground of each character is selectable from 16 colors, and the background from 8 colors. The foreground of each character can also be made to blink.

Variables

Here are the variables in *Crt*:

Variable	Type
CheckBreak	Boolean
CheckEof	Boolean
CheckSnow	Boolean
DirectVideo	Boolean
LastMode	Word
TextAttr	Byte
WindMin	Word
WindMax	Word

CheckBreak

Enables and disables checks for *Ctrl-Break*.

```
var CheckBreak: Boolean;
```

When *CheckBreak* is True, pressing *Ctrl-Break* aborts the program when it next writes to the display. When *CheckBreak* is False, pressing *Ctrl-Break* has no effect. *CheckBreak* is True by default. (At run time, *Crt* stores the old *Ctrl-Break* interrupt vector, $1B, in a global pointer variable called *SaveInt1B*.)

CheckEOF

Enables and disables the end-of-file character:

```
var CheckEOF: Boolean;
```

When *CheckEOF* is True, an end-of-file character is generated if you press *Ctrl-Z* while reading from a file assigned to the screen. When *CheckEOF* is False, pressing *Ctrl-Z* has no effect. *CheckEOF* is False by default.

CheckSnow

Enables and disables "snow-checking" when storing characters directly in video memory.

```
var CheckSnow: Boolean;
```

On most CGAs, interference will result if characters are stored in video memory outside the horizontal retrace intervals. This does not occur with Monochrome Adapters or EGAs.

When a color text mode is selected, *CheckSnow* is set to True, and direct video-memory writes will occur only during the horizontal retrace intervals. If you are running on a newer CGA, you may want to set *CheckSnow* to False at the beginning of your program and after each call to *TextMode*. This will disable snow-checking, resulting in significantly higher output speeds.

CheckSnow has no effect when *DirectVideo* is False.

DirectVideo

Enables and disables direct memory access for *Write* and *Writeln* statements that output to the screen.

```
var DirectVideo: Boolean;
```

When *DirectVideo* is True, *Write*s and *Writeln*s to files associated with the CRT will store characters directly in video memory instead of calling the BIOS to display them. When *DirectVideo* is False, all characters are written through BIOS calls, which is a significantly slower process.

DirectVideo always defaults to True. If, for some reason, you want characters displayed through BIOS calls, set *DirectVideo* to False at the beginning of your program and after each call to *TextMode*.

LastMode

Each time *TextMode* is called, the current video mode is stored in *LastMode*. In addition, *LastMode* is initialized at program startup to the then-active video mode.

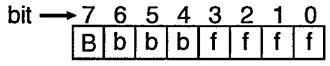
```
var LastMode: Word;
```

TextAttr

Stores the currently selected text attributes.

```
var TextAttr: Byte;
```

The text attributes are normally set through calls to *TextColor* and *TextBackground*. However, you can also set them by directly storing a value in *TextAttr*. The color information is encoded in *TextAttr* as follows:



where *ffff* is the 4-bit foreground color, *bbb* is the 3-bit background color, and *B* is the blink-enable bit. If you use the color constants for creating *TextAttr* values, note that the background color can only be selected from the first 8 colors, and that it must be multiplied by 16 to get it into the correct bit positions. The following assignment selects blinking yellow characters on a blue background:

```
TextAttr := Yellow + Blue * 16 + Blink;
```

WindMin and WindMax

Store the screen coordinates of the current window.

```
var WindMin, WindMax: Word;
```

These variables are set by calls to the *Window* procedure. *WindMin* defines the upper left corner, and *WindMax* defines the lower right corner. The x-coordinate is stored in the low byte, and the y-coordinate is stored in the high byte. For example, *Lo(WindMin)* produces the x-coordinate of the left edge, and *Hi(WindMax)* produces the y-coordinate of the bottom edge. The upper left corner of the screen corresponds to (*x*,*y*) = (0,0). Note, however, that for coordinates passed to *Window* and *GotoXY*, the upper left corner is at (1,1).

Procedures and functions

Function	Description
KeyPressed	Returns True if a key has been pressed on the keyboard, and False otherwise.
ReadKey	Reads a character from the keyboard.
WhereX	Returns the x-coordinate of the current cursor position, relative to the current window. *X* is the horizontal position.
WhereY	Returns the y-coordinate of the current cursor position, relative to the current window. *Y* is the vertical position.

Procedure	Description
AssignCrt	Associates a text file with the CRT.
ClrEol	Clears all characters from the cursor position to the end of the line without moving the cursor.

ClrScr	Clears the screen and places the cursor in 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.
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.

P A R T

3

Inside Turbo Pascal

C H A P T E R

16

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.

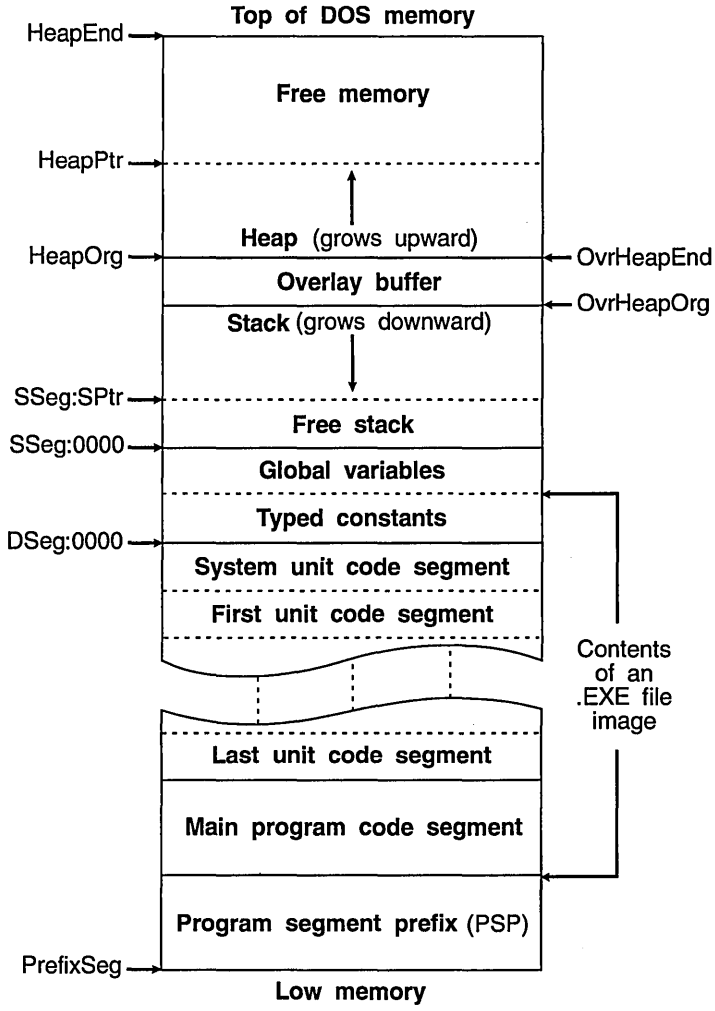
The Turbo Pascal memory map

Figure 16.1 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 Word 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 run-time library (the *System* unit). The size of a single code segment cannot exceed 64K, but the total size of the code is limited only by the available memory.

Figure 16.1
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 cannot 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 cannot exceed 64K; the default size is 16K, but this can be changed with a **$M** compiler directive.

The overlay buffer is used by the *Overlay* standard unit 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 is not available, the program will not execute. The default heap minimum is 0 bytes, and the default heap maximum is 640K; this means that by default the heap will occupy all remaining memory.

As you might expect, the heap manager (which is part of Turbo Pascal's run-time library) manages the heap. It is 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, thus 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), since every variable must be completely contained in a single segment.

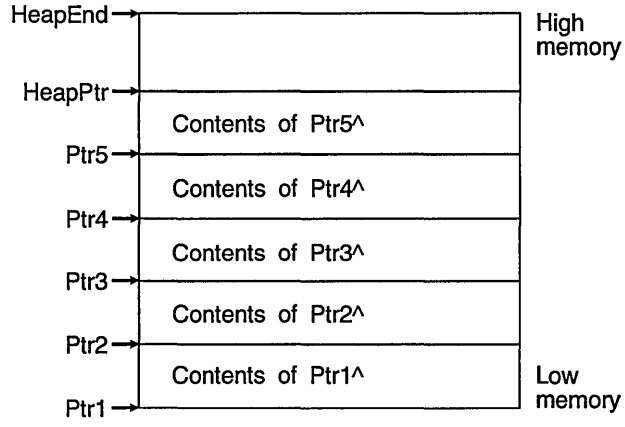
Disposal methods

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 the following figure:

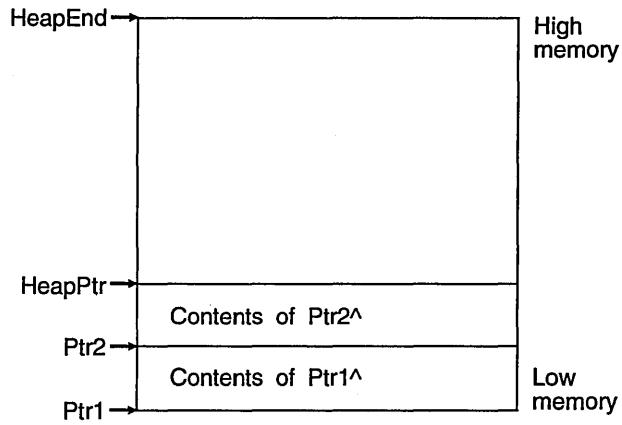
Figure 16.2
Disposal method using mark and release



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 16.3, effectively disposing of all pointers allocated since the call to *Mark*.

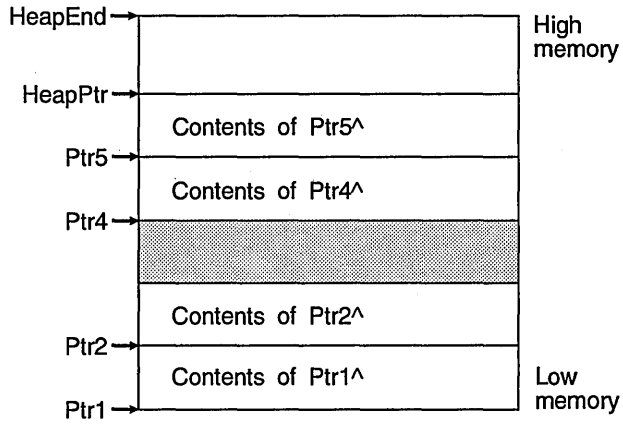
Figure 16.3
Heap layout with Release(P) executed



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.

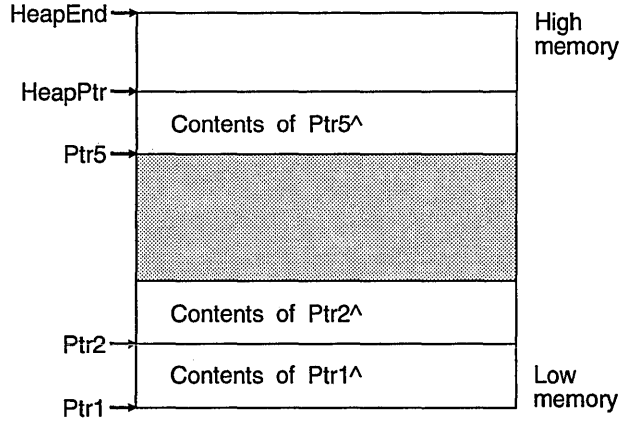
When a dynamic variable that is not the topmost variable on the heap is disposed of through *Dispose* or *FreeMem*, the heap becomes fragmented. Assuming that the same statement sequence has been executed, then after executing *Dispose(Ptr3)*, a "hole" is created in the middle of the heap (see Figure 16.4).

Figure 16.4
Creating a "hole" in the heap



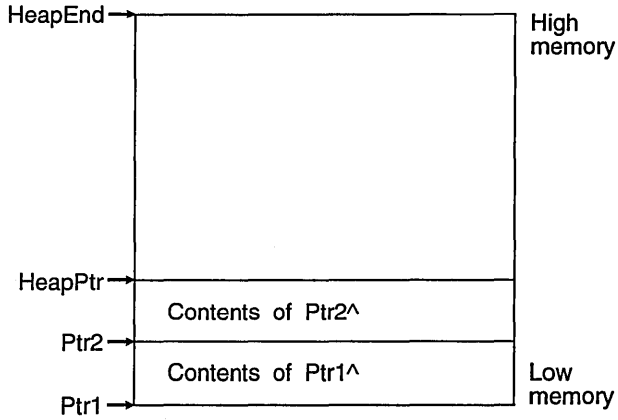
If *New(Ptr3)* had been executed now, it would again occupy the same memory area. On the other hand, executing *Dispose(Ptr4)* enlarges the free block, since *Ptr3* and *Ptr4* were neighboring blocks (see Figure 16.5).

Figure 16.5
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, since the last valid pointer is now *Ptr2* (see Figure 16.6).

Figure 16.6
Releasing the free block



The heap is now in the same state as it would be after executing *Release(P)*, as shown in Figure 16.3. However, the free blocks created and destroyed in the process were tracked for possible reuse.

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 (greater than or equal to the size of the requested block size) exists, it is used.

The *Release* procedure always clears the free list, thus 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 8 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* is *not* a normal 32-bit value; rather, it is 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. Thus, 8 bytes are allocated for blocks of size 1..8, 16 bytes are allocated for blocks of size 9..16, and so on. This may seem an excessive waste of memory at first, and indeed it would be if every block was just 1 byte in size. However, blocks are typically larger, and so the relative size of the unused space is less. Furthermore, and quite importantly, 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, do not heavily fragment the heap. For example, say a 50-byte block is allocated and disposed of, thus 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 gets called whenever the heap manager cannot 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 gets called whenever a call to *New* or *GetMem* cannot complete the request. The *Size* parameter contains the size of the block that could not 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, thus causing a run-time error whenever a call to *New* or *GetMem* cannot be completed. However, for many applications, 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 cannot complete the request, instead of aborting the program.

A call to the heap error function with a *Size* parameter of 0 indicates that to satisfy an allocation request 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. A call with

a *Size* of 0 does not indicate an error condition, since there was still adequate room for expansion between *HeapPtr* and *HeapEnd*—rather, the call serves as a notification that the unused area above *HeapPtr* has shrunk, and the heap manager ignores the return value from a call of this type.

Internal data formats

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 that can assume the value of 0 (False) or 1 (True).

Enumerated types

An enumerated type is stored as an unsigned byte if the enumeration has 256 or fewer values; otherwise, it is stored as an unsigned word.

Floating-point types

The floating-point types (Real, Single, Double, Extended, and Comp) store the binary representations of a sign (+ or –), an *exponent*, and a *significand*. A represented number has the value

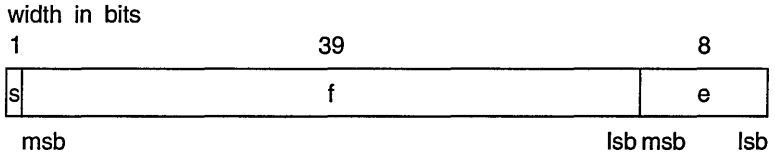
```
+/- significand × 2^exponent
```

where the significand has a single bit to the left of the binary decimal point (that is, 0 <= significand < 2).

In the figures that follow, *msb* means most significant bit, and *lsb* means least significant bit. The leftmost items are stored at the highest addresses. For example, for a real-type value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real type

A 6-byte (48-bit) *Real* number is divided into three fields:



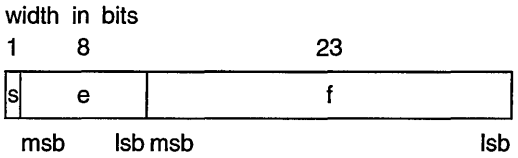
The value *v* of the number is determined by

```
if 0 < e <= 255, then v = (-1)^s * 2^(e-129) * (1.f).
if e = 0, then v = 0.
```

The Real type cannot store denormals, NaNs, and infinities. Denormals become zero when stored in a Real, and NaNs and infinities produce an overflow error if an attempt is made to store them in a Real.

The Single type

A 4-byte (32-bit) Single number is divided into three fields:

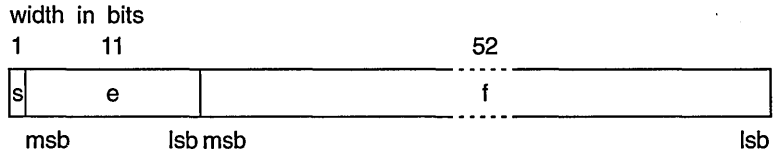


The value *v* of the number is determined by

```
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:

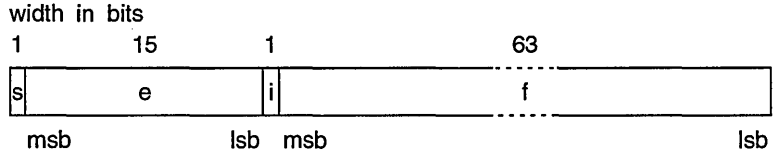


The value *v* of the number is determined by

```
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:

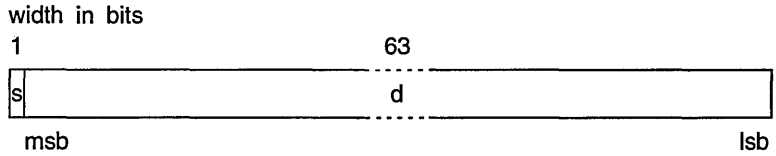


The value *v* of the number is determined by

```
if 0 <= e < 32767,       then v = (-1)^s * 2^(e-16383) * (i.f).
if e = 32767 and f = 0,  then v = (-1)^s * Inf.
if e = 32767 and f <> 0, then v is a NaN.
```

The Comp type

An 8-byte (64-bit) Comp number is divided into two fields:



The value *v* of the number is determined by

```
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 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 multi-dimensional array is stored with the rightmost dimension increasing first.

Record types

The fields of a record are stored as a contiguous sequence of variables. The first field is stored at the lowest memory address. If the record contains variant parts, then each variant starts at the same memory address.

File types

File types are represented as records. Typed files and untyped files occupy 128 bytes, which are laid out as follows:

```
type
  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 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 has not 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 19 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. Some examples include

```
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*.

C H A P T E R

17

Objects

Internal data format of objects

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 is not 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.

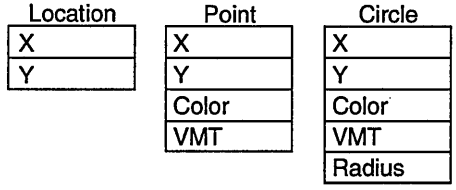
The following examples illustrate the internal data formats of object types:

```
type
  LocationPtr = ^Location;
  Location = object
    X, Y: Integer;
```

```
  procedure Init(PX, PY: Integer);
  function GetX: Integer;
  function GetY: Integer;
end;

PointPtr = ^Point;
Point = object(Location)
  Color: Integer;
  constructor Init(PX, PY, PColor: Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure MoveTo(PX, PY: Integer); virtual;
end;

CirclePtr = ^Circle;
Circle = object(Point)
  Radius: Integer;
  constructor Init(PX, PY, PColor, PRadius: Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure Fill; virtual;
end;
```

Figure 17.1 shows layouts of instances of *Location*, *Point*, and *Circle*; each box corresponds to one word of storage.

Figure 17.1
Layouts of instances of Location, Point, and Circle



Because *Point* is the first type in the hierarchy that introduces virtual methods, the VMT field is allocated right after the *Color* field.

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.

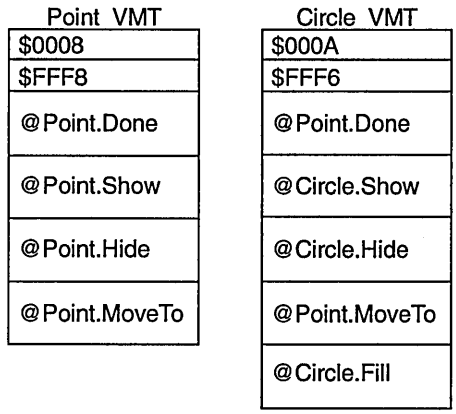
The second word of a VMT contains the negative size of instances of the associated object type; this information is used by the virtual method call validation mechanism to detect uninitialized objects (instances for which no constructor call has been made), and to check the consistency of the VMT. When 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 is not zero, and that the sum of the first and the second word is zero. If either check fails, run-time error 210 is generated.

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.

Finally, starting at offset 4 in the VMT, comes 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 17.2 shows the layouts of the VMTs of the *Point* and *Circle* types (the *Location* type has no VMT, since it contains no virtual methods, constructors, or destructors); each small box corresponds to one word of storage, and each large box corresponds to two words of storage.

Figure 17.2
Point and Circle's VMT layouts



Notice how *Circle* inherits the *Done* and *MoveTo* methods from *Point*, 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 *Point*, and an instance *C* of type *Circle*, a call to *P.Init* will automatically store the offset of *Point*'s VMT in *P*'s VMT field, and a call to *C.Init* will likewise store the offset of *Circle*'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* will already have been set up. Thus, if the need arises, a constructor can make calls to virtual methods.

The SizeOf function

When applied to an instance of an object type that has a VMT, *SizeOf* returns the size stored in the VMT. Thus, for object types that have a VMT, *SizeOf* always returns the *actual* size of the instance, rather than the *declared* size.

The TypeOf function

Turbo Pascal's new standard function *TypeOf* returns a pointer to an object type's VMT. *TypeOf* takes a single parameter, which can be either an object type identifier or an object type instance. In both cases, the result, which is of type *Pointer*, is a pointer to the

object type's VMT. *TypeOf* can be applied only to object types that have a VMT—all other types result in an error.

The *TypeOf* function can be used to test the actual type of an instance. For example,

```
if TypeOf(Self) = TypeOf(Point) then ...
```

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 *PointPtr*, 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+8]    ;Call VMT entry for Show
```

The type compatibility rules of object types allow *PP* to point at a *Point* or a *Circle*, or at any other descendant of *Point*. And if you examine the VMTs shown here, you'll see that for a *Point*, the entry at offset 8 in the VMT points to *Point.Show*; whereas for a *Circle*, it points to *Circle.Show*. Thus, depending upon the *actual* run-time type of *PP*, the **CALL** instruction calls *Point.Show* or *Circle.Show*, or the *Show* method of any other descendant of *Point*.

If *Show* had been a static method, this code would have been generated for the call to *PP^.Show*:

```
les     di,PP               ;Load PP into ES:DI
push    es                  ;Pass as Self parameter
push    di
call    Point.Show          ;Directly call Point.Show
```

Here, no matter what *PP* points to, the code will always call the *Point.Show* method.

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 *PointPtr* as defined earlier, 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+16]   ;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.

Constructors and destructors

Constructors and destructors use the same calling conventions as normal 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 in order to initialize *Self*.

Furthermore, 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. This causes the constructor to allocate a new dynamic object, the address of which is passed back to the caller in DX:AX when the constructor returns. If the constructor could not allocate the object, a **nil** pointer is returned in DX:AX.

See "Constructor error recovery" on page 236.

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).

Extensions to New and Dispose

The *New* and *Dispose* standard procedures have been extended to allow a constructor call or destructor call as a second parameter for allocating or disposing a dynamic object type variable. The syntax is

```
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, occurrences of such "pairs" of a call to *New* followed by a constructor call, and a destructor call followed by a call to *Dispose* would be very common. The extended syntax improves readability, and also generates shorter and more efficient code.

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

```
var
  SP: StrFieldPtr;
  ZP: ZipFieldPtr;
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;
```

An additional extension allows *New* to be used as a *function*, which allocates and returns a dynamic variable of a specified type. The syntax is

```
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: FieldPtr;
begin
  F1 := New(StrFieldPtr, Init(1, 1, 25, 'Firstname'));
  F2 := New(ZipFieldPtr, Init(1, 2, 5, 'Zip code', 0, 99999));
  ...
  WriteLn(F1^.GetStr);                    { Calls StrField.GetStr }
  WriteLn(F2^.GetStr);                    { Calls ZipField.GetStr }
  ...
  Dispose(F2, Done);                          { Calls Field.Done }
  Dispose(F1, Done);                      { Calls StrField.Done }
end;
```

Notice that even though *F1* and *F2* are of type *FieldPtr*, the extended pointer assignment compatibility rules allow *F1* and *F2* to be assigned a pointer to any descendant of *Field*; and since *GetStr* and *Done* are virtual methods, the virtual method dispatch mechanism correctly calls *StrField.GetStr*, *ZipField.GetStr*, *Field.Done*, and *StrField.Done*, respectively.

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 cannot 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.

As an example of assembly language methods, we've implemented a simple *Rect* object.

```
type
  Rect = object
    X1, Y1, X2, Y2: Integer;
    procedure Init(XA, YA, XB, YB: Integer);
    procedure Union(var R: Rect);
    function Contains(X, Y: Integer): Boolean;
  end;
```

A *Rect* represents a rectangle bounded by four coordinates, *X1*, *Y1*, *X2*, and *Y2*. The upper left corner of a rectangle is defined by *X1* and *Y1*, and the lower right corner is defined by *X2* and *Y2*. The *Init* method assigns values to the rectangle's coordinates; the *Union* method calculates the smallest rectangle that contains both the rectangle itself and another rectangle; and the *Contains* method returns True if a given point is within the rectangle, or False if not. Other methods, such as moving, resizing, calculating intersections, and testing for equality, could easily be implemented to make *Rect* a more useful object.

The Pascal implementations of *Rect*'s methods list only the method header followed by an **external** reserved word.

```
{$L RECT}

procedure Rect.Init(XA, YA, XB, YB: Integer); external;
procedure Rect.Union(var R: Rect); external;
function Rect.Contains(X, Y: Integer): Boolean; external;
```

There is, of course, no requirement that all methods be implemented as externals. Each individual method can be implemented in either Pascal or in assembly language, as desired.

The assembly language source file, RECT.ASM, that implements the three external methods is listed here.

```
        TITLE   Rect
        LOCALS  @@
```

```
; Rect structure

Rect    STRUC
X1      DW      ?
Y1      DW      ?
X2      DW      ?
Y2      DW      ?
Rect    ENDS

Code    SEGMENT BYTE PUBLIC

        ASSUME  cs:code

; Procedure Rect.Init(XA, YA, XB, YB: Integer)

        PUBLIC  Rect@Init

Rect@Init       PROC    FAR

@XA             EQU     (WORD PTR [bp+16])
@YA             EQU     (WORD PTR [bp+14])
@XB             EQU     (WORD PTR [bp+12])
@YB             EQU     (WORD PTR [bp+10])
@Self           EQU     (DWORD PTR [bp+6])

        push    bp              ;Save bp
        mov     bp,sp           ;Set up stack frame
        les     di,@Self        ;Load Self into ES:DI
        cld                     ;Move forwards
        mov     ax,@XA          ;X1 := XA
        stosw
        mov     ax,@YA          ;Y1 := YA
        stosw
        mov     ax,@XB          ;X2 := XB
        stosw
        mov     ax,@YB          ;Y2 := YB
        stosw
        pop     bp              ;Restore BP
        ret     12              ;Pop parameters and return

Rect@Init       ENDP

; Procedure Rect.Union(var R: Rect)

        PUBLIC  Rect@Union

Rect@Union      PROC    FAR

@R              EQU     (DWORD PTR [bp+10])
@Self           EQU     (DWORD PTR [bp+6])

        push    bp              ;Save BP
        mov     bp,sp           ;Set up stack frame
        push    ds              ;Save DS
        lds     si,@R           ;Load R into DS:SI
        les     di,@Self        ;Load Self into ES:DI
```

```
        cld                     ;Move forward
        lodsw                   ;If R.X1 >= X1 goto @@1
        scasw
        jge     @@1
        dec     di              ;X1 := R.X1
        dec     di
        stosw
@@1:    lodsw                   ;If R.Y1 >= Y1 goto @@2
        scasw
        jge     @@2
        dec     di              ;Y1 := R.Y1
        dec     di
        stosw
@@2:    lodsw                   ;If R.X2 <= X2 goto @@3
        scasw
        jle     @@3
        dec     di              ;X2 := R.X2
        dec     di
        stosw
@@3:    lodsw                   ;If R.Y2 <= Y2 goto @@4
        scasw
        jle     @@4
        dec     di              ;Y2 := R.Y2
        dec     di
        stosw
@@4:    pop     ds              ;Restore DS
        pop     bp              ;Restore BP
        ret     8               ;Pop parameters and return

Rect@Union      ENDP

; Function Rect.Contains(X, Y: Integer): Boolean

        PUBLIC  Rect@Contains

Rect@Contains   PROC    FAR

@X              EQU     (WORD PTR [bp+12])
@Y              EQU     (WORD PTR [bp+10])
@Self           EQU     (DWORD PTR [bp+6])

        push    bp              ;Save BP
        mov     bp,sp           ;Set up stack frame
        les     di,@Self        ;Load Self into ES:DI
        mov     al,0            ;Return false
        mov     dx,@X           ;If (X < X1) or (X > X2) goto @@1
        cmp     dx,es:[di].X1
        jl      @@1
        cmp     dx,es:[di].X2
        jg      @@1
        mov     dx,@Y           ;If (Y < Y1) or (Y > Y2) goto @@2
        cmp     dx,es:[di].Y1
```

```
        jl      @@1
        cmp     dx,es:[di].Y2
        jg      @@1
        inc     ax              ;Return true
@@1:    pop     bp              ;Restore BP
        ret     8               ;Pop parameters and return

Rect@Contains   ENDP

Code    ENDS

        END
```

Constructor error recovery

As described in Chapter 16, Turbo Pascal allows you to install a heap error function through the *HeapError* variable in the *System* unit. This functionality is still supported in Turbo Pascal, but now it also affects the way object type constructors work.

By default, when there is not enough memory to allocate a dynamic instance of an object type, a constructor call using the extended syntax of the *New* standard procedure generates run-time 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 cannot complete the request (instead of aborting the program).

The code that performs allocation and 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 already have been allocated and initialized. If allocation fails, and if the heap error function returns 1, the constructor skips execution of the statement part and returns a **nil** pointer; thus, the pointer specified in the *New* construct that called the constructor is set to **nil**.

There's a new standard procedure called Fail.

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. However, the constructor itself might attempt to allocate dynamic variables, in order to initialize pointer fields in the instance, and these allocations might in turn fail. If that happens, a well-behaved constructor should reverse any successful allocations, and finally deallocate the object type instance so that the net result becomes a **nil** pointer. To make such "backing out" possible, Turbo Pascal implements a new standard

procedure called *Fail*, which 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.

The following program implements two simple object types that contain pointers. This first version of the program does not implement constructor error recovery.

```
type
  LinePtr = ^Line;
  Line = string[79];

  BasePtr = ^Base;
  Base = object
    L1, L2: LinePtr;
    constructor Init(S1, S2: Line);
    destructor Done; virtual;
    procedure Dump; virtual;
  end;

  DerivedPtr = ^Derived;
  Derived = object(Base)
    L3, L4: LinePtr;
    constructor Init(S1, S2, S3, S4: Line);
    destructor Done; virtual;
    procedure Dump; virtual;
  end;

var
  BP: BasePtr;
  DP: DerivedPtr;

constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  L1^ := S1;
  L2^ := S2;
```

```
end;

destructor Base.Done;
begin
  Dispose(L2);
  Dispose(L1);
end;

procedure Base.Dump;
begin
  WriteLn('B: ', L1^, ', ', L2^, '.');
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  Base.Init(S1, S2);
  New(L3);
  New(L4);
  L3^ := S3;
  L4^ := S4;
end;

destructor Derived.Done;
begin
  Dispose(L4);
  Dispose(L3);
  Base.Done;
end;

procedure Derived.Dump;
begin
  WriteLn('D: ', L1^, ', ', L2^, ', ', L3^, ', ', L4^, '.');
end;

begin
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('North', 'East', 'South', 'West'));
  BP^.Dump;
  DP^.Dump;
  Dispose(DP, Done);
  Dispose(BP, Done);
end.
```

The next example demonstrates how the previous one can be rewritten to implement error recovery. The type and variable declarations are not repeated, because they remain the same.

```
constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  if (L1 = nil) or (L2 = nil) then
```

```
  begin
    Base.Done;
    Fail;
  end;
  L1^ := S1;
  L2^ := S2;
end;

destructor Base.Done;
begin
  if L2 <> nil then Dispose(L2);
  if L1 <> nil then Dispose(L1);
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  if not Base.Init(S1, S2) then Fail;
  New(L3);
  New(L4);
  if (L3 = nil) or (L4 = nil) then
  begin
    Derived.Done;
    Fail;
  end;
  L3^ := S3;
  L4^ := S4;
end;

destructor Derived.Done;
begin
  if L4 <> nil then Dispose(L4);
  if L3 <> nil then Dispose(L3);
  Base.Done;
end;

{$F+}
function HeapFunc(Size: Word): Integer;
begin
  HeapFunc := 1;
end;
{$F-}

begin
  HeapError := @HeapFunc;              { Install heap error handler }
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('North', 'East', 'South', 'West'));
  if (BP = nil) or (DP = nil) then
    WriteLn('Allocation error')
  else
  begin
    BP^.Dump;
```

```
    DP^.Dump;
  end;
  if DP <> nil then Dispose(DP, Done);
  if BP <> nil then Dispose(BP, Done);
end.
```

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.

C H A P T E R

18

Control issues

This chapter describes in detail the various ways that Turbo Pascal implements program control. Included are calling conventions, exit procedures, interrupt handling and error handling.

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 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 does not support byte-sized **PUSH** and **POP** instructions, so byte-sized parameters are always transferred onto the stack as words. The low-order byte of the word contains the value, and the high-order byte is unused (and undefined).

An integer type or parameter is passed as a byte, a word, or a double word, using the same format as an integer-type variable. (For double words, the high-order word is pushed before the low-order word so that the low-order word ends up at the lowest address.)

A Char-type parameter is passed as an unsigned byte.

A Boolean-type parameter is passed as a byte with the value 0 or 1.

An enumerated-type parameter is passed as an unsigned byte if the enumeration has 256 or fewer values; otherwise, it is passed as an unsigned word.

A Real-type parameter (type Real) is passed as 6 bytes on the stack, thus being an exception to the rule that only 1-, 2-, and 4-byte values are passed directly on the stack.

A floating-point type parameter (Real, Single, Double, Extended, and Comp) is passed as 4, 6, 8, or 10 bytes on the stack, thus being an exception to the rule that only 1-, 2-, and 4-byte values are passed directly on the stack.

Version 4.0 of Turbo Pascal passed 8087-type parameters (Single, Double, Extended, and Comp) on the internal stack of the 8087 numeric coprocessor. For reasons of compatibility with other

languages, and to avoid 8087 stack overflows, this version uses the 8086 stack.

A pointer-type parameter is passed as 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.

A set-type parameter 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.

Function results

Ordinal-type function results (Integer, Char, Boolean, and enumeration types) 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).

8087-type function results (type Single, Double, Extended, and Comp) are returned in the 8087 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 8086 CPU supports two kinds of call and return instructions: near and far. The near instructions transfer control to another location within the same code segment, and the far instructions allow a change of code segment.

A **NEAR CALL** instruction pushes a 16-bit return address (offset only) onto the stack, and a **FAR CALL** instruction pushes a 32-bit return address (both segment and offset). The corresponding **RET** instructions pop only an offset or both an offset and a segment.

Turbo Pascal will automatically select 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 may be required to be far. For example, in an overlaid application, all procedures and functions are generally required to be far; likewise, if a procedure or function is to be assigned to a procedural variable, it has to 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 is declared within another procedure or function. By default, nested procedures and functions always use the near call model, since they are only "visible" within a specific procedure or function in the same code segment. However, in an overlaid application, 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 cannot be declared with the **external** directive, and they cannot be procedural parameters.*

```
procedure PA; near;
var
  IntA: Integer;

procedure B; far;
var
  IntB: Integer;

procedure C; near;
var
  IntC: Integer;
begin
  inline(
    $8B/$46/<IntC/        { MOV  AX,[BP + IntC]    ;AX = IntC }
    $8B/$5E/$04/          { MOV  BX,[BP + 4]       ;BX = B's stack
                                                    frame }
    $36/$8B/$47/<IntB/    { MOV  AX,SS:[BX + IntB] ;AX = IntB }
    $8B/$5E/$04/          { MOV  BX,[BP + 4]       ;BX = B's stack
                                                    frame }
    $36/$8B/$5F/$06/      { MOV  BX,SS:[BX + 6]    ;BX = A's stack
                                                    frame }
    $36/$8B/$47/<IntA);   { MOV  AX,SS:[BX + IntA] ;AX = IntA }
end;

begin end;

begin end;
```

Entry and exit code

Each Pascal procedure and function begins and ends with standard entry and exit code that creates and removes its activation.

The standard entry code is

```
push  bp                ;Save BP
mov   bp, sp            ;Set up stack frame
sub   sp, Localsize     ;Allocate local variables
```

where *LocalSize* is the size of the local variables. The **SUB** instruction is only present if *LocalSize* is not 0. If the procedure's call model is near, the parameters start at BP + 4; if it is far, they start at BP + 6.

The standard exit code is

```
mov   sp, bp            ;Deallocate local variables
pop   bp                ;Restore BP
ret   ParamSize         ;Remove parameters and return
```

where *ParamSize* is the size of the parameters. The **RET** instruction is either a near or a far return, depending on the procedure's call model.

Register-saving conventions

Procedures and functions should preserve the BP, SP, SS, and DS registers. All other registers may 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 always gets called as a part of a program's termination, whether it is 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 in the {**$F+**} state 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; for instance, to close files or to restore interrupt vectors. The procedures on the exit chain get executed in reverse order of installation. This ensures that the exit code of one unit does not get 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. Furthermore, 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 will not be called again.

An exit procedure may 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, and restores the interrupt vectors that were captured by Turbo Pascal. In addition, if *ErrorAddr* is not **nil**, it outputs a run-time error message.

If you wish to present run-time error messages yourself, install an exit procedure that examines *ErrorAddr* and outputs a message if it is not **nil**. In addition, before returning, make sure to set *ErrorAddr* to **nil**, so that the error is not reported again by other exit procedures.

Once the run-time library has called all exit procedures, it returns to DOS, passing as a return code the value stored in *ExitCode*.

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

Interrupt procedures are declared 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 pseudo-parameters 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 is 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 call }
procedure IntHandler(SI, DI, DS, ES, BP: Word);      { Valid call }
```

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 further 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 refrain from using any of Turbo Pascal's input and output or dynamic memory allocation routines, because they are not reentrant. Likewise, no DOS functions can be used because DOS is not reentrant.

C H A P T E R

19

Input and output issues

Text file device drivers

As mentioned in Chapter 10, "The System unit," Turbo Pascal allows you to 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: TextRec): Integer;
```

where *TextRec* is the text file record type defined in the earlier section, "File types," in Chapter 3. 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*. The return value of 0 indicates a successful operation.

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.

Assuming, for example, that the four device interface functions are called *DevOpen*, *DevInOut*, *DevFlush*, and *DevClose*, the *Assign* procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

The device interface functions can use the *UserData* field in the file record to store private information. This field is not 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, *Assign* 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*, and *Close* functions 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 0 in *BufPos*. If the *InOut* function returns 0 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 0 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 0 in *BufPos* and *BufEnd* to flush the remaining (un-read) 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 will not 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 *InOut* to ensure that all characters have been written to the device.

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. Some examples include:

```
Port[$20] := $20;
Port[Base] := Port[Base] xor Mask;
while Port[$B2] and $80 = 0 do  { Wait };
```

Use of the *Port* and *PortW* arrays is restricted to assignment and reference in expressions only, that is, components of *Port* and *PortW* cannot be used as variable parameters. Furthermore, references to the entire *Port* or *PortW* array (reference without index) are not allowed.

C H A P T E R

20

Automatic optimizations

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.

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 will reference 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 instance:

```
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 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.

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*, since this enables the compiler to produce better code. Because of this, it is 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

The operation *X* * *C*, where *C* is a constant and a power of 2, is coded using a **SHL** 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, since word-sized items on even addresses are accessed faster than words on odd addresses.

For further details, refer to Chapter 21, "Compiler directives."

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.

Dead code removal

Statements that are known never to execute do not 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 never get 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; and since *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. However, no references are made to *P1*, 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.

C H A P T E R

21

Compiler directives

Some of the Turbo Pascal compiler's features are controlled through *compiler directives*. A compiler directive is a comment with a special syntax. Turbo Pascal allows compiler directives wherever comments are allowed.

A compiler directive starts with a **$** as the first character after the opening comment delimiter, and is immediately followed by a name (one or more letters) that designates the particular directive. There are three types of directives:

- **Switch directives.** These directives turn particular compiler features on or off by specifying + or – immediately after the directive name.
- **Parameter directives.** These directives specify parameters that affect the compilation, such as file names and memory sizes.
- **Conditional directives.** These directives control conditional compilation of parts of the source text, based on user-definable conditional symbols.

All directives, except switch directives, must have at least one blank between the directive name and the parameters. Here are some examples of compiler directives:

```
{$B+}
{$R- Turn off range checking}
{$I TYPES.INC}
{$O EdFormat}
{$M 65520,8192,655360}
{$DEFINE Debug}
```

```
{$IFDEF Debug}
{$ENDIF}
```

You can put compiler directives directly into your source code. You can also change the default directives for both the command-line compiler (TPC.EXE) and the IDE (TURBO.EXE). The **O**ptions | **C**ompiler menu contains all the compiler directives; any changes you make to the settings there will affect all subsequent compilations. When using the command-line compiler, you can specify compiler directives on the command line (for example, `TPC /$R+ MYPROG`), or you can place directives in a configuration file (TPC.CFG—see Chapter 9 of the *User's Guide* for information). Compiler directives in the source code always override the default values in both the command-line compiler and the IDE.

Switch directives

Switch directives are either *global* or *local*. Global directives affect the entire compilation, whereas local directives affect only the part of the compilation that extends from the directive until the next occurrence of the same directive.

Global directives must appear before the declaration part of the program or the unit being compiled, that is, before the first **uses**, **label**, **const**, **type**, **procedure**, **function**, or **begin** keyword. Local directives, on the other hand, can appear anywhere in the program or unit.

Multiple switch directives can be grouped in a single compiler directive comment by separating them with commas; for example,

```
{$B+,R-,S-}
```

There can be no spaces between the directives in this case.

Align data

Syntax `{$A+}` or `{$A-}`

Default `{$A+}`

Type Global

Menu equivalent **O**ptions | **C**ompiler | **W**ord Align Data

Command-line The command-line compiler equivalent is the **/$A** option.

Remarks The **$A** directive switches between byte and word alignment of variables and typed constants. Word alignment has no effect on the 8088 CPU. However, on all 80x86 CPUs, word alignment means faster execution, since word-sized items on even addresses are accessed in one memory cycle, in comparison to two memory cycles for words on odd addresses.

In the {**$A+**} state, all variables and typed constants larger than one byte are aligned on a machine-word boundary (an even-numbered address). If required, unused bytes are inserted between variables to achieve word alignment. The {**$A+**} directive does not affect byte-sized variables; neither does it affect fields of record structures and elements of arrays. A field in a record will align on word boundary only if the total size of all fields before it is even. Likewise, for every element of an array to align on a word boundary, the size of the elements must be even.

In the {**$A-**} state, no alignment measures are taken. Variables and typed constants are simply placed at the next available address, regardless of their size. If you are recompiling programs using the Turbo Pascal Editor Toolbox, make sure to compile all programs that use the toolbox with {**$A-**}.

Regardless of the state of the **$A** directive, each global **var** and **const** declaration section always starts at a word boundary. Likewise, the compiler always attempts to keep the stack pointer (SP) word aligned, by allocating an extra unused byte in a procedure's stack frame if required.

Boolean evaluation

Syntax `{$B+}` or `{$B-}`

Default `{$B-}`

Type Local

Menu equivalent **O**ptions | **C**ompiler | Complete **B**oolean Eval

Remarks The **$B** directive switches between the two different models of code generation for the **and** and **or** Boolean operators.

In the {**$B+**} state, the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression, built from the **and** and **or** operators, is guaranteed to be evaluated, even when the result of the entire expression is already known.

In the {**$B-**} state, the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident.

For further details, refer to the section "Boolean operators" in Chapter 6, "Expressions."

Debug information

Syntax {$D+} or {$D-}

Default {$D+}

Type Global

Menu equivalent **O**ptions | **C**ompiler | **D**ebug Information

Remarks The **$D** directive enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object code addresses into source text line numbers.

When the **D**ebug Information option is checked for a given program or unit, Turbo Pascal's integrated debugger allows you to single-step and set breakpoints in that module. Furthermore, when a run-time error occurs in a program or unit compiled with {**$D+**}, Turbo Pascal can automatically take you to the statement that caused the error with **S**earch | Find **E**rror.

The Debugging (**O**ptions | De**b**ugger) and Map File (**O**ptions | **L**inker) options produce complete information for a given module only if you've compiled that module in the {**$D+**} state.

For units, the debug information is recorded in the .TPU file along with the unit's object code. Debug information increases the size of .TPU files, and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

The **$D** switch is usually used in conjunction with the **$L** switch, which enables and disables the generation of local symbol information for debugging.

If you want to use the Turbo Debugger to debug your program, set **C**ompile | **D**estination to *Disk* and check Standalone in **O**ptions | De**b**ugger | Debugging.

Emulation

Syntax {$E+} or {$E-}

Default {$E+}

Type Global

Menu equivalent **O**ptions | **C**ompiler | **E**mulation

Remarks The **$E** directive enables or disables linking with a run-time library that will emulate the 8087 numeric coprocessor if it is not present.

When you compile a program in the **{$N+,E+}** state, Turbo Pascal links with the full 8087 emulator. The resulting .EXE file can be used on any machine, regardless of whether an 8087 is present. If one is found, Turbo Pascal will use it; otherwise, the run-time library emulates it.

In the **{$N+,E-}** state, Turbo Pascal links with a substantially smaller floating-point library, which can only be used if an 8087 is present.

The 8087 emulation switch has no effect if used in a unit; it applies only to the compilation of a program. Furthermore, if the program is compiled in the **{$N-}** state, and if all the units used by the program were compiled with **{$N-}**, then an 8087 run-time library is not required, and the 8087 emulation switch is ignored.

Force far calls

Syntax {$F+} or {$F-}

Default {$F-}

Type Local

Menu equivalent **O**ptions | **C**ompiler | **F**orce Far Calls

Remarks The **$F** directive controls which call model to use for subsequently compiled procedures and functions. Procedures and functions compiled in the **{$F+}** state always use the far call model. In the **{$F-}** state, Turbo Pascal automatically selects the appropriate model: far if the procedure or function is declared in the **interface** section of a unit; near otherwise.

The near and far call models are described in full in Chapter 18, "Control issues."

For programs that use overlays, we suggest that you place a {**$F+**} directive at the beginning of the program and each unit, in order to satisfy the far call requirement. For more discussion, see Chapter 13, "The Overlay unit." For programs that use procedural variables, all such procedures must use the far code model. For more discussion, see "Procedural variables" in Chapter 8.

Generate 80286 code

Syntax {$G+} or {$G-}

Default {$G-}

Type Local

Menu equivalent **O**ptions | **C**ompiler | **2**86 instructions

The **$G** directive enables or disables 80286 code generation. In the {**$G-**} state, only generic 8086 instructions are generated, and programs compiled in this state can run on any 80x86 family processor. In the {**$G+**} state, the compiler uses the additional instructions of the 80286 to improve code generation, but programs compiled in this state cannot run on 8088 and 8086 processors. Additional instructions used in the {**$G+**} state include **ENTER, LEAVE, PUSH** immediate, extended **IMUL**, and extended **SHL** and **SHR**.

Input/output checking

Syntax {$I+} or {$I-}

Default {$I+}

Type Local

Menu equivalent **O**ptions | **C**ompiler | **I**/O Checking

Remarks The **$I** directive enables or disables the automatic code generation that checks the result of a call to an I/O procedure. I/O procedures are described in Chapter 19, "Input and output issues." If an I/O procedure returns a nonzero I/O result when this switch is on, the program terminates, displaying a run-time error message. When this switch is off, you must check for I/O errors by using the *IOResult* function.

Local symbol information

Syntax {$L+} or {$L-}

Default {$L+}

Type Global

Menu equivalent **O**ptions | **C**ompiler | **L**ocal Symbols

Remarks The **$L** directive enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module's implementation part, and the symbols within the module's procedures and functions.

When local symbols are on for a given program or unit, Turbo Pascal's integrated debugger allows you to examine and modify the module's local variables. Furthermore, calls to the module's procedures and functions can be examined via the **W**indow | Call Stac**k** window.

Object method implementations written in assembly language can be linked with Turbo Pascal programs using the **$L** compiler directive and the **external** keyword. For more information, see Chapter 23, "Linking assembler code."

The Map File (**O**ptions | **L**inker) and Debugging (**O**ptions | De**b**ugger) options produce local symbol information for a given module only if that module was compiled in the **{$L+}** state.

For units, the local symbol information is recorded in the .TPU file along with the unit's object code. Local symbol information increases the size of .TPU files, and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

The **$L** switch is usually used in conjunction with the **$D** switch, which enables and disables the generation of line-number tables for debugging. Note that the **$L** directive is ignored if **D**ebug Information is unchecked **{$D-}**.

Numeric processing

Syntax {$N+} or {$N-}

Default {$N-}

Type Global

Menu equivalent **O**ptions | **C**ompiler | **8**087/80287

Remarks The **$N** directive switches between the two different models of floating-point code generation supported by Turbo Pascal. In the **{$N-}** state, code is generated to perform all real-type calculations in software by calling run-time library routines. In the **{$N+}** state, code is generated to perform all Real-type calculations using the 8087 numeric coprocessor.

Note that you can also use the **{$E+}** directive to emulate the 8087. This gives you access to the IEEE floating-point types without requiring that you install an 8087 chip.

Overlay code generation

Syntax {$O+} or {$O-}

Default {$O-}

Type Global

Menu equivalents **O**ptions | **C**ompiler | **O**verlays Allowed

The **$O** directive enables or disables overlay code generation. Turbo Pascal allows a unit to be overlaid only if it was compiled with **{$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.

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

A **{$O+}** compiler directive is almost always used in conjunction with a **{$F+}** directive to satisfy the overlay manager's far call requirement.

For further details on overlay code generation, refer to Chapter 13, "The Overlay unit."

Range checking

Syntax {$R+} or {$R-}

Default {$R-}

Type Local

Menu equivalent **O**ptions | **C**ompiler | **R**ange Checking

Remarks The **$R** directive enables or disables the generation of range-checking code. In the **{$R+}** state, all array and string-indexing expressions are verified as being within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, the program terminates and displays a run-time error message. Enabling range checking slows down your program and makes it larger. Use this option when debugging, then turn it off once the program is bug free.

If **$R** is switched on, all calls to virtual methods are checked for the initialization status of the object instance making the call. If the instance making the call has not been initialized by its constructor, a range check run-time error occurs.

Enabling range checking and virtual method call checking slows down your program and makes it somewhat larger, so use the **{$R+}** only for debugging.

Stack-overflow checking

Syntax {$S+} or {$S-}

Default {$S+}

Type Local

Menu equivalent **O**ptions | **C**ompiler | **S**tack Checking

Remarks The **$S** directive enables or disables the generation of stack-overflow checking code. In the **{$S+}** state, the compiler generates code at the beginning of each procedure or function that checks whether there is sufficient stack space for the local variables and other temporary storage. When there is not enough stack space, a call to a procedure or function compiled with **{$S+}** causes the program to terminate and display a run-time error message. In the **{$S-}** state, such a call is most likely to cause a system crash.

Var-string checking

Syntax {$V+} or {$V-}

Default {$V+}

Type Local

Menu equivalent **O**ptions | **C**ompiler | Strict **V**ar-Strings

Remarks The **$V** directive controls type checking on strings passed as variable parameters. In the **{$V+}** state, strict type checking is performed, requiring the formal and actual parameters to be of *identical* string types. In the **{$V-}** (relaxed) state, any string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

Extended syntax

Syntax {$X+} or {$X-}

Default {$X-}

Type Global

Menu equivalent **O**ptions | **C**ompiler | **E**xtended Syntax

The **$X** compiler directive enables or disables Turbo Pascal's extended syntax. In the **{$X+}** mode, function calls can be used as statements; that is, the result of a function call can be discarded. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. However, in certain cases a function can carry out multiple operations based on its parameters, and some of those cases may not produce a sensible result—in such cases, the **{$X+}** extensions allow the function to be treated as a procedure.

The **{$X+}** directive does not apply to built-in functions (that is, functions defined in the *System* unit).

In the default state, **{$X-}**, this extension is disabled and attempting to use it will cause an error.

Parameter directives

Include file

Syntax `{$I filename}`

Type Local

Menu equivalent **O**ptions | **D**irectories | Include Directories

Remarks The **$I** directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text right after the {**$I** *filename*} directive. The default extension for *filename* is .PAS. If *filename* does not specify a directory, then, in addition to searching for the file in the current directory, Turbo Pascal searches in the directories specified in the **O**ptions | **D**irectories | Include Directories input box (or in the directories specified in the **/I** option on the TPC command line).

You can nest Include files up to 15 levels deep.

There is one restriction to the use of Include files: An Include file cannot be specified in the middle of a statement part. In fact, all statements between the **begin** and **end** of a statement part must reside in the same source file.

Link object file

Syntax `{$L filename}`

Type Local

Menu equivalent **O**ptions | **D**irectories | **O**bject Directories

Remarks The Link object file directive instructs the compiler to link the named file with the program or unit being compiled. The **$L** directive is used to link with code written in assembly language for subprograms declared to be **external**. The named file must be an Intel relocatable object file (.OBJ file). The default extension for *filename* is .OBJ. If *filename* does not specify a directory, then, in addition to searching for the file in the current directory, Turbo Pascal searches in the directories specified in the **O**ptions | **D**irectories | **O**bject Directories input box (or in the directories specified in the **/O** option on the TPC command line). For further details

about linking with assembly language, see Chapter 23, "Linking assembler code."

Memory allocation sizes

Syntax `{$M stacksize,heapmin,heapmax}`

Default `{$M 16384,0,655360}`

Type Global

Menu equivalent **O**ptions | **M**emory Sizes

Remarks The Memory allocation sizes directive specifies a program's memory allocation parameters. *stacksize* must be an integer number in the range 1,024 to 65,520, which specifies the size of the stack segment. *heapmin* must be in the range 0 to 655,360, and *heapmax* must be in the range *heapmin* to 655,360. *heapmin* and *heapmax* specify the minimum and maximum sizes of the heap, respectively.

The stack segment and the heap are further discussed in Chapter 4, "Variables," and Chapter 16, "Memory issues."

The **$M** directive has no effect when used in a unit.

Overlay unit name

Syntax `{$O unitname}`

Type Local

Menu equivalent None

Remarks The Overlay unit name directive turns a unit into an overlay.

The {**$O** *unitname*} directive has no effect if used in a unit; when compiling a program, it specifies which of the units used by the program should be placed in an .OVR file instead of in the .EXE file.

{**$O** *unitname*} directives must be placed after the program's **uses** clause. Turbo Pascal reports an error if you attempt to overlay a unit that wasn't compiled in the {**$O+**} state. Any unit named in a {**$O** *unitname*} directive must have been compiled with **O**verlays Allowed set to On in the IDE (the equivalent of the {**$O+**} compiler directive).

For further details on overlays, refer to Chapter 13, "The Overlay unit."

Conditional compilation

Turbo Pascal's conditional compilation directives allow you to produce different code from the same source text, based on conditional symbols.

There are two basic conditional compilation constructs, which closely resemble Pascal's **if** statement. The first construct

```
{$IFxxx} ... {$ENDIF}
```

causes the source text between {**$IF***xxx*} and {**$ENDIF**} to be compiled only if the condition specified in {**$IF***xxx*} is True; if the condition is False, the source text between the two directives is ignored.

The second conditional compilation construct

```
{$IFxxx} ... {$ELSE} ... {$ENDIF}
```

causes either the source text between {**$IF***xxx*} and {**$ELSE**} or the source text between {**$ELSE**} and {**$ENDIF**} to be compiled, based on the condition specified by the {**$IF***xxx*}.

Here are some examples of conditional compilation constructs:

```
{$IFDEF Debug}
  Writeln('X = ', X);
{$ENDIF}

{$IFDEF CPU87}
  {$N+}
  type
    Real = Double;
{$ELSE}
  {$N-}
  type
    Single = Real;
    Double = Real;
    Extended = Real;
    Comp = Real;
{$ENDIF}
```

You can nest conditional compilation constructs up 16 levels deep. For every {**$IF***xxx*}, the corresponding {**$ENDIF**} must be found within the same source file—which means there must be an equal number of {**$IF***xxx*}'s and {**$ENDIF**}'s in every source file.

Conditional symbols

Conditional compilation is based on the evaluation of conditional symbols. Conditional symbols are defined and undefined (forgotten) using the directives

```
{$DEFINE name}
{$UNDEF name}
```

You can also use the **/D** switch in the command-line compiler (or place it in the Conditional Defines input box from within **O**ptions | **C**ompiler of the IDE).

Conditional symbols are best compared to Boolean variables: They are either True (defined) or False (undefined). The **{$DEFINE}** directive sets a given symbol to True, and the **{$UNDEF}** directive sets it to False.

Conditional symbols follow the exact same rules as Pascal identifiers: They must start with a letter, followed by any combination of letters, digits, and underscores. They can be of any length, but only the first 63 characters are significant.

Important! Conditional symbols and Pascal identifiers have no correlation whatsoever. Conditional symbols cannot be referenced in the actual program, and the program's identifiers cannot be referenced in conditional directives. For example, the construct

```
const
  Debug = True;
begin
  {$IFDEF Debug}
   Writeln('Debug is on');
  {$ENDIF}
end;
```

will *not* compile the *Writeln* statement. Likewise, the construct

```
{$DEFINE Debug}
begin
  if Debug then
   Writeln('Debug is on');
end;
```

will result in an unknown identifier error in the **if** statement.

Turbo Pascal defines the following standard conditional symbols:

VER60 Always defined, indicating that this is version 6.0 of Turbo Pascal. Other versions (starting with 4.0) define their corresponding version symbol; for instance, VER40 for version 4.0, and so on.

MSDOS Always defined, indicating that the operating system is MS-DOS or PC-DOS. Versions of Turbo Pascal for other operating systems will instead define a symbolic name for that particular operating system.

CPU86 Always defined, indicating that the CPU belongs to the 80x86 family of processors. Versions of Turbo Pascal for other CPUs will instead define a symbolic name for that particular CPU.

CPU87 Defined if an 80x87 numeric coprocessor is present at compile time. If the construct

```
{$IFDEF CPU87} {$N+} {$ELSE} {$N-} {$ENDIF}
```

appears at the beginning of a compilation, Turbo Pascal automatically selects the appropriate model of floating-point code generation for that particular computer.

Other conditional symbols can be defined before a compilation by using the **C**onditional Defines input box (**O**ptions | **C**ompiler), or the **/D** command-line option if you are using TPC.

The DEFINE directive

Syntax `{$DEFINE name}`

Remarks Defines a conditional symbol of *name*. The symbol is recognized for the remainder of the compilation of the current module in which the symbol is declared, or until it appears in an {**$UNDEF** *name*} directive. The {**$DEFINE** *name*} directive has no effect if *name* is already defined.

The UNDEF directive

Syntax `{$UNDEF name}`

Remarks Undefines a previously defined conditional symbol. The symbol is forgotten for the remainder of the compilation or until it reappears in a {**$DEFINE** *name*} directive. The {**$UNDEF** *name*} directive has no effect if *name* is already undefined.

The IFDEF directive

Syntax `{$IFDEF name}`

Remarks Compiles the source text that follows it if *name* is defined.

The IFNDEF directive

Syntax `{$IFNDEF name}`

Remarks Compiles the source text that follows it if *name* is not defined.

The IFOPT directive

Syntax `{$IFOPT switch}`

Remarks Compiles the source text that follows it if *switch* is currently in the specified state. *switch* consists of the name of a switch option, followed by a + or a – symbol. For example, the construct

```
{$IFOPT N+}
  type Real = Extended;
{$ENDIF}
```

will compile the type declaration if the **$N** option is currently active.

The ELSE directive

Syntax {$ELSE}

Remarks Switches between compiling and ignoring the source text delimited by the last {**$IF***xxx*} and the next {**$ENDIF**}.

The ENDIF directive

Syntax {$ENDIF}

Remarks Ends the conditional compilation initiated by the last {**$IF***xxx*} directive.

PART 4

Using Turbo Pascal with assembly language

C H A P T E R

22

The inline assembler

Turbo Pascal's inline 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 inline assembler implements a large subset of the syntax supported by Turbo Assembler and Microsoft's Macro Assembler. The inline 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 inline 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 inline assembler can be thought of as an assembler language compiler that uses Pascal syntax for all declarations.

The **asm** statement

The inline assembler is accessed through **asm** statements. The syntax of an **asm** statement is

```
asm AsmStatement < Separator AsmStatement > end
```

where *AsmStatement* is an assembler statement, and *Separator* is a semicolon, a new-line, or a Pascal comment. Here are some examples of **asm** statements:

```
if EnableInts then
  asm
    sti
  end
  else
  asm
    cli
  end;

asm
  mov ax,Left; xchg ax,Right; mov Left,ax;
end;

asm
  mov     ah,0                          { Read keyboard function code }
  int     16H                             { Call PC BIOS to read key }
  mov     CharCode,al                             { Save ASCII code }
  mov     ScanCode,ah                              { Save scan code }
end;

asm
  push    ds                                                 { Save DS }
  lds     si,Source                            { Load source pointer }
  les     di,Dest                            { Load destination pointer }
  mov     cx,Count                                  { Load block size }
  cld                                                 { Move forwards }
  rep     movsb                                          { Copy block }
  pop     ds                                             { Restore DS }
end;
```

Notice that multiple assembler statements can be placed on one line if they are separated by semicolons. Also notice that a semicolon is not required between two assembler statements if the statements are on separate lines. Finally, notice that a semicolon does *not* indicate that the rest of the line is a comment—comments must be written in Pascal style using **{** and **}** or **(*** and ***)**.

Register use

The rules of register use in an **asm** statement are in general 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

The syntax of an assembler statement is

```
[ Label ":" ] < Prefix > [ Opcode [ Operand < "," Operand > ] ]
```

where *Label* is a label identifier, *Prefix* is an assembler prefix opcode (operation code), *Opcode* is an assembler instruction opcode or directive, and *Operand* is an assembler expression.

Comments are allowed between assembler statements, but not within them. For example, this is allowed:

```
asm
  mov ax,1 {Initial value}
  mov cx,100 {Count}
end;
```

but this is an error:

```
asm
  mov {Initial value} ax,1;
  mov cx, {Count} 100
end;
```

Labels

Labels are defined in assembler just as in Pascal, by writing a label identifier and a colon before a statement; and just as in Pascal, labels defined in assembler must be declared in a **label** declaration part in the block containing the **asm** statement. There is however one exception to this rule: *local labels*.

Local labels are labels that start with an at-sign (@). Since an at-sign cannot 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 does not have to be declared in a **label** declaration part before it is 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 (:).

The following program fragment demonstrates use of normal and local labels in **asm** statements:

```
label Start, Stop;

  ...

begin
  asm
    Start:

    ...

      jz        Stop
    @1:
      .
      .
      loop    @1
  end;
  asm
    @1:
      .
      .
      jc      @2
      .
      .
      jmp     @1
    @2:
  end;
  goto Start;
  Stop:
end;
```

Notice that a normal label can be defined within an **asm** statement and referenced outside an **asm** statement and vice

versa. Also, notice that the same local label name can be used in different **asm** statements.

Prefix opcodes

The inline assembler supports the following prefix opcodes:

LOCK	Bus lock
REP	Repeat string operation
REPE/REPZ	Repeat string operation while equal/zero
REPNE/REPNZ	Repeat string operation while not equal/zero
SEGCS	CS (code segment) override
SEGDS	DS (data segment) override
SEGES	ES (extra segment) override
SEGSS	SS (stack segment) override

Zero or more of these can prefix an assembler instruction. For example,

```
asm
  rep movsb              { Move CX bytes from DS:SI to ES:DI }
  SEGES lodsw            { Load word from ES:SI }
  SEGCS mov ax,[bx]      { Same as MOV AX,CS:[BX] }
  SEGES                  { Affects next assembler statement }
  mov WORD PTR [DI],0    { Becomes MOV WORD PTR ES:[DI],0 }
end;
```

Notice that a prefix opcode can be specified without an instruction opcode in the same statement—in that case, the prefix opcode affects the instruction opcode in the following assembler statement.

An instruction opcode seldom, if ever, has more than one prefix opcode, and at most no more than three prefix opcodes can make sense (a **LOCK**, followed by a **SEG*xx***, followed by a **REP*xx***). Be careful about using multiple prefix opcodes—ordering is important, and some 80x86 processors do not handle all combinations correctly. For example, an 8086 or 8088 "remembers" only the **REP*xx*** prefix if an interrupt occurs in the middle of a repeated string instruction, so a **LOCK** or **SEG*xx*** prefix cannot safely be coded before a **REP*xx*** string instruction.

Instruction opcodes

The inline 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;
```

The **RETN** and **RETF** instructions on the other hand 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 inline 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 inline 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 inline 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* is not within reach of a short jump is converted to a machine code sequence that corresponds to

```
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 inline 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
```

will 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 inline 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 may 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 may be a constant expression with a value between –32,768 and 65,535, or an address expression. For an address expression, the inline 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 may 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 inline 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 inline assembler statements. To generate uninitialized or initialized data in the data segment, you should use normal 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 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 inline assembler does not support such variable declarations. In Turbo Pascal, the only kind of symbol that can be defined in an inline assembler statement is a label. All variables must be declared using Pascal syntax, and the preceding construct corresponds to

```
var
  ByteVar: Byte;
  WordVar: Word;
```

```
  ...
asm
  mov     al,ByteVar
  mov     bx,WordVar
end;
```

Operands

Inline assembler operands are expressions, which consist of a combination of constants, registers, symbols, and operators. Although inline assembler expressions are built using the same basic principles as Pascal expressions, there are a number of important differences, as will be explained in a following section.

Within operands, the following reserved words have a predefined meaning to the inline assembler:

AH	CL	FAR	SEG
AL	CS	HIGH	SHL
AND	CX	LOW	SHR
AX	DH	MOD	SI
BH	DI	NEAR	SP
BL	DL	NOT	SS
BP	DS	OFFSET	ST
BX	DWORD	OR	TBYTE
BYTE	DX	PTR	TYPE
CH	ES	QWORD	WORD
			XOR

The reserved words always take precedence over user-defined identifiers. For instance, the code fragment

```
var
  ch: Char;
  ...
asm
  mov     ch, 1
end;
```

will load 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 is strongly suggested that you avoid user-defined identifiers with the same names as inline assembler reserved words, since such name confusion can easily lead to very obscure and hard-to-find bugs.

Expressions

The inline assembler evaluates all expressions as 32-bit integer values; it does not support floating-point and string values, except string constants.

Inline 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 inline assembler expressions is that all inline assembler expressions must resolve to a *constant value*, in other words a value that can be computed at compile time. For example, given the declarations

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid inline assembler statement:

```
asm
  mov     Z,X+Y
end;
```

Since 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 inline assembler can no longer compute the value of *X* + *Y* at compile time. The correct inline assembler construct to move the sum of *X* and *Y* into *Z* now becomes

```
asm
  mov     ax,X
  add     ax,Y
  mov     Z,ax
end;
```

Another important difference between Pascal and inline 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 inline 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 inline 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 does not 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 inline 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.

Numeric constants by default use decimal (base 10) notation, but the inline 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 are not 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; thus, 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 since 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. Some examples of string constants and their corresponding numeric values follow:

'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:

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 is 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 inline assembler allows you to access almost all Pascal symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions. In addition, the inline 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 the 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 inline assembler:

```
function Sum(X, Y: Integer): Integer;
begin
  asm
    mov     ax,X
    add     ax,Y
    mov     @Result,AX
  end;
end;
```

The following symbols cannot be used in inline 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 22.1 summarizes the value, class, and type of the different kinds of symbols that can be used in inline assembler expressions. (Expression classes and types are described in a following section.)

Table 22.1
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 the declarations

```
procedure Test;
var
  Count: Integer;
```

the instruction

```
asm
  mov     ax,Count
end;
```

assembles into `MOV AX,[BP-2]`.

The inline 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 is not the case in inline assembler. Since **var** parameters are really pointers, you have to treat them as such in inline assembler. 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 which can be accessed using the period (.) structure member selector operator. For example, given the declarations

```
type
  Point = record
    X, Y: Integer;
  end;
  Rect = record
    A, B: Point;
  end;
var
  P: Point;
  R: Rect;
```

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 instructions below generate the same machine code, which loads the contents of ES:[DI+4] into AX.

```
asm
  mov     ax,(Rect PTR es:[di]).B.X
  mov     ax,Rect(es:[di]).B.X
  mov     ax,es:Rect[di].B.X
  mov     ax,Rect[es:di].B.X
  mov     ax,es:[di].Rect.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 inline 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;
```

Since *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 does not know the final address of a label, variable, procedure, or function; it does not become known until link time, when the linker assigns a specific absolute address to the symbol.

The inline 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 inline assembler expression has an associated type—or more correctly, an associated size, since the inline 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, since it occupies 2 bytes.

The inline assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;

...

asm
  mov     al,QuitFlag
  mov     bx,OutBufPtr
end;
```

the inline 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, the following is not allowed:

```
asm
  mov     dl,OutBufPtr
end;
```

since DL is a byte-sized register and *OutBufPtr* is a word. The type of a memory reference can be changed through a typecast; correct ways of writing the previous instruction are

```
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 inline assembler permits both of these instructions, since 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 cannot be determined from another operand, the inline assembler requires an explicit typecast:

```
asm
  inc     BYTE PTR [100H]
  imul    WORD PTR [100H]
end;
```

Table 22.2 summarizes the predefined type symbols that the inline assembler provides in addition to any currently declared Pascal types.

Table 22.2
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** pseudo-types, 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 inline 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 inline assembler provides a variety of operators, divided into 12 classes of precedence. Table 22.3 lists the inline assembler's expression operators in decreasing order of precedence.

Table 22.3
Inline asssembler expression operators

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

*Inline assembler operator precedence is different from Pascal. For example, in an inline assembler expression, the **AND** operator has lower precedence than the plus (+) and minus (-) operators, whereas in a Pascal expression, it has higher precedence.*

&	**Identifier override.** The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as an inline assembler reserved symbol.
(...)	**Subexpression.** Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression may 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 may be combined with the BX, BP, SI, or DI registers using the plus (+) operator, to indicate CPU register indexing. Another expression may optionally precede the expression within the brackets; the result in this case becomes the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector.** The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	**Unary plus.** Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
–	**Unary minus.** Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.

:	**Segment override.** Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction will be prefixed by an appropriate segment override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (low-order word) of the expression following the operator. The result is an immediate value.
SEG	Returns the segment part (high-order word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	**Typecast operator.** The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	**Multiplication.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
+	**Addition.** The expressions can be immediate values or memory references, but only one of the expressions

can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions are memory references, the result is also a memory reference.

–	**Subtraction.** The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
NOT	**Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	**Bitwise AND.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	**Bitwise OR.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	**Bitwise exclusive OR.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembler procedures and functions

So far, every **asm**...**end** construct you've seen has been a statement within a normal **begin**...**end** statement part. Turbo Pascal's assembler directive allows you to write complete procedures and functions in inline 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 param-

eters, and other value parameters whose size is not 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 which gets allocated by the caller.
- The compiler generates no stack frame for procedures and functions that 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
```

  where *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, Char, Boolean, and enumerated types) 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 normal functions and assembler functions. The

first example uses an **asm** statement in a normal function to convert a string to upper case. Notice that the value parameter *Str* in this case refers to a local variable, since 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* is not copied into local storage, and the function must treat *Str* as a **var** parameter.

```
function UpperCase(S: 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**, and 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**, and 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 cannot 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 is 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 are not allowed. For example, this means that the assembly language **HIGH** and **LOW** operators cannot 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,

memory model, and language support for Turbo Pascal programmers.

Using **TPASCAL** with the **.MODEL** directive sets up Pascal calling conventions, defines the segment names, does the **PUSH** BP and **MOV** BP,SP, and it also sets up the return with **POP** BP and **RET** *N* (where *N* is the number of parameter bytes).

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 TPASCAL
        .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;
```

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 Strings;
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 *Strings* unit can be compiled. Note that the routines use the far call model because they are declared in the **interface** section of the unit.

```
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
TPC strings
```

The next example shows how an assembly language routine can refer to Pascal routines and variables. The *Numbers* program reads up to 100 Integer values, and then calls an assembly language procedure to check the range of each of these values. If a value is out of range, the assembly language procedure calls a Pascal procedure to print it.

```
program Numbers;
{$L CHECK}
var
  Buffer: array[1..100] of Integer;
  Count: Integer;

procedure RangeError(N: Integer);
begin
  Writeln('Range error: ',N);
end;

procedure CheckRange(Min, Max: Integer); external;
begin
  Count := 0;
  while not Eof and (Count < 100) do
  begin
  { Ends when you type Ctrl-Z or after 100 iterations }
    Count := Count + 1;
    Readln(Buffer[Count]);
  end;
  CheckRange(-10, 10);
end.
```

The assembly language file that implements the *CheckRange* procedure is shown next. It must be assembled into a file called CHECK.OBJ before the *Numbers* program can be compiled. Note that the procedure uses the near call model because it is declared in a program.

```
DATA    SEGMENT WORD PUBLIC
        EXTRN   Buffer : WORD, Count : WORD ;Pascal variables
DATA    ENDS
CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS : CODE, DS : Buffer
        EXTRN   RangeError : NEAR       ;Implemented in Pascal
        PUBLIC  CheckRange              ;Implemented here

CheckRange      PROC    NEAR

        mov     bx,sp                   ;Get parameters pointer
        mov     ax, ss:[BX + 4]         ;Load Min
        mov     dx, ss:[BX + 2]         ;Load Max
        xor     bx, bx                  ;Clear Data index
        mov     cx, count               ;Load Count
        jcxz    SD4                     ;Skip if zero
SD1:    cmp     Buffer[BX], AX          ;Too small?
        jl      SD2                     ;Yes, jump
        cmp     Buffer[BX], DX          ;Too large?
        jle     SD3                     ;No, jump
SD2:    push    ax                      ;Save registers
        push    bx
        push    cx
        push    dx
        push    Buffer[BX]              ;Pass offending value to
                                        ; Pascal
        call    RangeError              ;Call Pascal procedure
        pop     dx                      ;Restore registers
        pop     cx
        pop     bx
        pop     ax
SD3:    inc     bx                      ;Point to next element
        inc     bx
        loop    SD1                     ;Loop for each item
SD4:    ret     4                       ;Clean stack and return

CheckRange      ENDP
CODE    ENDS
        END
```

Turbo Assembler example

Here's a Turbo Assembler version of the previous assembly language example that takes advantage of TASM's support for Turbo Pascal:

```
        .MODEL  TPASCAL                 ;Turbo Pascal code model
        LOCALS  @@                      ;Define local labels
                                        ; prefix
        .DATA                           ;Data segment
        EXTRN   Buffer : WORD, Count : WORD
                                        ;Pascal variables

        .CODE                           ;Code segment
        EXTRN   RangeError : NEAR       ;Implemented in Pascal
        PUBLIC  CheckRange              ;Implemented here

CheckRange      PROC NEAR Min : WORD, Max : WORD

        mov     ax, Min                 ;Keep Min in AX
        mov     dx, Max                 ;Keep Max in DX
        xor     bx, BX                  ;Clear Buffer index
        mov     cx, Count               ;Load Count
        jcxz    @@4                     ;Skip if zero
@@1:    cmp     ax, Buffer[BX]          ;Too small?
        jg      @@2                     ;Yes, go to CR2
        cmp     dx, Buffer[BX]          ;Too large?
        jge     @@3                     ;No, go to CR3
@@2:    push    ax                      ;Save registers
        push    bx
        push    cx
        push    dx
        push    Buffer[BX]              ;Pass offending value to
                                        ; Pascal
        call    RangeError              ;Call Pascal procedure
        pop     dx                      ;Restore registers
        pop     cx
        pop     bx
        pop     ax
@@3:    inc     bx                      ;Point to next element
        inc     bx
        loop    @@1                     ;Loop for each item
@@4:    ret                             ;Done

CheckRange      ENDP
        END
```

Notice that with **.MODEL TPASCAL** Turbo Assembler automatically generates entry code before the first instruction, and generates exit code upon seeing the **RET**.

Inline machine code

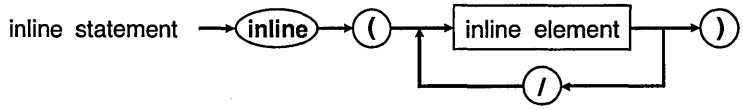
For very short assembly language subroutines, Turbo Pascal's **inline** statements and directives are very convenient. They let you insert machine code instructions directly into the program or unit text instead of through an object file.

Inline statements

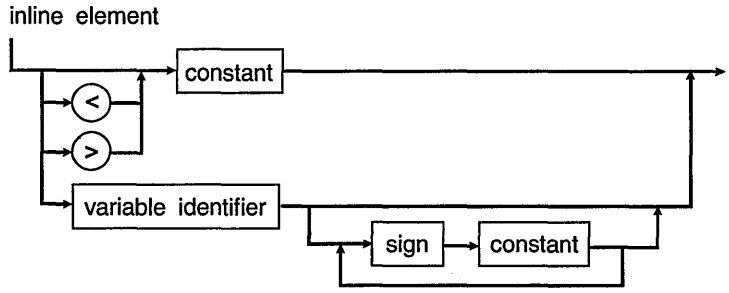
An **inline** statement consists of the reserved word **inline** followed by one or more inline elements, separated by slashes and enclosed in parentheses:

```
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 one 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, one 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 is 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.

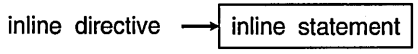
The following example of an **inline** statement generates machine code for storing a specified number of words of data in a specified variable. When called, procedure *FillWord* stores *Count* words of the value *Data* in memory, starting at the first byte occupied by *Dest*.

```
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

Inline directives let you 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:



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 cannot 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 are not required, because the 4 bytes are inserted into the instruction stream when *LongMul* is called.

Inline directives are intended for very short (less than 10 bytes) procedures and functions only.

Because of the macro-like nature of inline procedures and functions, they cannot be used as arguments to the @ operator and the *Addr*, *Ofs*, and *Seg* functions.

PART 5

Appendixes

A P P E N D I X

A

Error messages

Compiler error messages

The following lists the possible error messages you can get from the compiler during program development. Whenever possible, the compiler will display additional diagnostic information in the form of an identifier or a file name. For example,

```
Error 15: File not found (WINDOW.TPU).
```

When an error is detected, Turbo Pascal (in the IDE) automatically loads the source file and places the cursor at the error. The command-line compiler displays the error message and number and the source line, and uses a caret (^) to indicate where the error occurred. Note, however, that some errors are not detected until a little later in the source text. For example, a type mismatch in an assignment statement cannot be detected until the entire expression after the := has been evaluated. In such cases, look for the error to the left of or above the cursor.

1 Out of memory.

This error occurs when the compiler has run out of memory. There are a number of possible solutions to this problem:

- If **C**ompile | **D**estination is set to *Memory*, set it to *Disk* in the integrated environment.
- If the Memory radio button is chosen (**O** | **L** | **L**ink Buffer) in the integrated environment, toggle it to *Disk*. Use the **/L** option to link to disk in the command-line compiler.

- If you are using any memory-resident utilities, such as SideKick and SuperKey, remove them from memory.
- If you are using TURBO.EXE, try using TPC.EXE instead—it takes up less memory.

If none of these suggestions help, your program or unit may simply be too large to compile in the amount of memory available, and you may have to break it into two or more smaller units.

2 Identifier expected.

An identifier was expected at this point. You may be trying to redeclare a reserved word.

3 Unknown identifier.

This identifier has not been declared, or may not be visible within the current scope.

4 Duplicate identifier.

The identifier has already been used within the current block.

5 Syntax error.

An illegal character was found in the source text. You may have forgotten the quotes around a string constant.

6 Error in real constant.

The syntax of real-type constants is defined in Chapter 1, "Tokens and constants."

7 Error in integer constant.

The syntax of integer-type constants is defined in Chapter 1, "Tokens and constants." Note that whole real numbers outside the maximum integer range must be followed by a decimal point and a zero; for example, 12,345,678,912.0.

8 String constant exceeds line.

You have most likely forgotten the ending quote in a string constant.

9 Too many nested files.

The compiler allows no more than 15 nested source files. Most likely you have more than four nested Include files.

10 Unexpected end of file.

You might have gotten this error message because of one of the following:

- Your source file ends before the final **end** of the main statement part. Most likely, your **begin**s and **end**s are unbalanced.
- An Include file ends in the middle of a statement part. Every statement part must be entirely contained in one file.
- You didn't close a comment.

11 Line too long.

The maximum line length is 126 characters.

12 Type identifier expected.

The identifier does not denote a type as it should.

13 Too many open files.

If this error occurs, your CONFIG.SYS file does not include a FILES=*xx* entry or the entry specifies too few files. Increase the number to some suitable value, for instance, 20.

14 Invalid file name.

The file name is invalid or specifies a nonexistent path.

15 File not found.

The file could not be found in the current directory or in any of the search directories that apply to this type of file.

16 Disk full.

Delete some files or use a new disk.

17 Invalid compiler directive.

The compiler directive letter is unknown, one of the compiler directive parameters is invalid, or you are using a global compiler directive when compilation of the body of the program has begun.

18 Too many files.

There are too many files involved in the compilation of the program or unit. Try not to use that many files, for instance, by merging Include files or making the file names shorter.

19 Undefined type in pointer definition.

The type was referenced in a pointer-type declaration previously, but it was never declared.

20 Variable identifier expected.

The identifier does not denote a variable as it should.

21 Error in type.

This symbol cannot start a type definition.

22 Structure too large.

The maximum allowable size of a structured type is 65,520 bytes.

23 Set base type out of range.

The base type of a set must be a subrange with bounds in the range 0..255 or an enumerated type with no more than 256 possible values.

24 File components may not be files or objects.

file of file and **file of object** constructs are not allowed; nor is any structured type that includes an object type or file type.

25 Invalid string length.

The declared maximum length of a string must be in the range 1..255.

26 Type mismatch.

This is due to one of the following:

- incompatible types of the variable and the expression in an assignment statement
- incompatible types of the actual and formal parameter in a call to a procedure or function
- an expression type that is incompatible with index type in array indexing
- incompatible types of operands in an expression

27 Invalid subrange base type.

All ordinal types are valid base types.

28 Lower bound greater than upper bound.

The declaration of a subrange type specifies a lower bound greater than the upper bound.

29 Ordinal type expected.

Real types, string types, structured types, and pointer types are not allowed here.

30 Integer constant expected.

31 Constant expected.

32 Integer or real constant expected.

33 Pointer type identifier expected.

The identifier does not denote a pointer type as it should.

34 Invalid function result type.

Valid function result types are all simple types, string types, and pointer types.

35 Label identifier expected.

The identifier does not denote a label as it should.

36 BEGIN expected.

A **begin** is expected here, or there is an error in the block structure of the unit or program.

37 END expected.

An **end** is expected here, or there is an error in the block structure of the unit or program.

38 Integer expression expected.

The preceding expression must be of an integer type.

39 Ordinal expression expected.

The preceding expression must be of an ordinal type.

40 Boolean expression expected.

The preceding expression must be of type boolean.

41 Operand types do not match operator.

The operator cannot be applied to operands of this type, for example, *'A'* **div** '2'.

42 Error in expression.

This symbol cannot participate in an expression in the way it does. You may have forgotten to write an operator between two operands.

43 Illegal assignment.

- Files and untyped variables cannot be assigned values.
- A function identifier can only be assigned values within the statement part of the function.

44 Field identifier expected.

The identifier does not denote a field in the preceding record variable.

45 Object file too large.

Turbo Pascal cannot link in .OBJ files larger than 64K.

46 Undefined external.

The **external** procedure or function did not have a matching **PUBLIC** definition in an object file. Make sure you have specified all object files in {**$L** *filename*} directives, and check the spelling of the procedure or function identifier in the .ASM file.

47 Invalid object file record.

The .OBJ file contains an invalid object record; make sure the file is in fact an .OBJ file.

48 Code segment too large.

The maximum size of the code of a program or unit is 65,520 bytes. If you are compiling a program, move some procedures or functions into a unit. If you are compiling a unit, break it into two or more units.

49 Data segment too large.

The maximum size of a program's data segment is 65,520 bytes, including data declared by the used units. If you need more global data than this, declare the larger structures as pointers, and allocate them dynamically using the *New* procedure.

50 DO expected.

The reserved word **do** does not appear where it should.

51 Invalid PUBLIC definition.

- Two or more **PUBLIC** directives in assembly language define the same identifier.
- The .OBJ file defines **PUBLIC** symbols that do not reside in the **CODE** segment.

52 Invalid EXTRN definition.

- The identifier was referred to through an **EXTRN** directive in assembly language, but it is not declared in the Pascal program or unit, nor in the interface part of any of the used units.

- The identifier denotes an **absolute** variable.
- The identifier denotes an **inline** procedure or function.

53 Too many EXTRN definitions.

Turbo Pascal cannot handle .OBJ files with more than 256 **EXTRN** definitions.

54 OF expected.

The reserved word **of** does not appear where it should.

55 INTERFACE expected.

The reserved word **interface** does not appear where it should.

56 Invalid relocatable reference.

- The .OBJ file contains data and relocatable references in segments other than **CODE**. For example, you are attempting to declare initialized variables in the **DATA** segment.
- The .OBJ file contains byte-sized references to relocatable symbols. This error occurs if you use the **HIGH** and **LOW** operators with relocatable symbols or if you refer to relocatable symbols in **DB** directives.
- An operand refers to a relocatable symbol that was not defined in the **CODE** segment or in the **DATA** segment.
- An operand refers to an **EXTRN** procedure or function with an offset, for example, **CALL** *SortProc+8*.

57 THEN expected.

The reserved word **then** does not appear where it should.

58 TO or DOWNTO expected.

The reserved word **to** or **downto** does not appear where it should.

59 Undefined forward.

- The procedure or function was declared in the **interface** part of a unit, but its definition never occurred in the **implementation** part.

- The procedure or function was declared with **forward**, but its definition was never found.

60 Too many procedures.

Turbo Pascal does not allow more than 512 procedures or functions per module. If you are compiling a program, move some procedures or functions into a unit. If you are compiling a unit, break it into two or more units.

61 Invalid typecast.

- The sizes of the variable reference and the destination type differ in a variable typecast.
- You are attempting to typecast an expression where only a variable reference is allowed.

62 Division by zero.

The preceding operand attempts to divide by zero.

63 Invalid file type.

The file type is not supported by the file-handling procedure; for example, *Readln* with a typed file or *Seek* with a text file.

64 Cannot Read or Write variables of this type.

- *Read* and *Readln* can input variables of Char, integer, real, and string types.
- *Write* and *Writeln* can output variables of Char, integer, real, string, and Boolean types.

65 Pointer variable expected.

The preceding variable must be of a pointer type.

66 String variable expected.

The preceding variable must be of a string type.

67 String expression expected.

The preceding expression must be of a string type.

68 Circular unit reference.

Two units are not allowed to use each other:

```
unit U1;          unit U2;
uses U2;          uses U1;
...               ...
```

In this example, doing a Make on either unit generates error 68.

69 Unit name mismatch.

The name of the unit found in the .TPU file does not match the name specified in the **uses** clause.

70 Unit version mismatch.

One or more of the units used by this unit have been changed since the unit was compiled. Use **C**ompile | **M**ake or **C**ompile | **B**uild in the IDE and **/M** or **/B** options in the command-line compiler to automatically compile units that need recompilation.

71 Duplicate unit name.

You have already named this unit in the **uses** clause.

72 Unit file format error.

The .TPU file is somehow invalid; make sure it is in fact a .TPU file.

73 IMPLEMENTATION expected.

The reserved word **implementation** does not appear where it should.

74 Constant and case types do not match.

The type of the **case** constant is incompatible with the **case** statement's selector expression.

75 Record variable expected.

The preceding variable must be of a record type.

76 Constant out of range.

You are trying to

- index an array with an out-of-range constant
- assign an out-of-range constant to a variable
- pass an out-of-range constant as a parameter to a procedure or function

77 File variable expected.

The preceding variable must be of a file type.

78 Pointer expression expected.

The preceding expression must be of a pointer type.

79 Integer or real expression expected.

The preceding expression must be of an integer or a real type.

80 Label not within current block.

A **goto** statement cannot reference a label outside the current block.

81 Label already defined.

The label already marks a statement.

82 Undefined label in preceding statement part.

The label was declared and referenced in the preceding statement part, but it was never defined.

83 Invalid @ argument.

Valid arguments are variable references and procedure or function identifiers.

84 UNIT expected.

The reserved word **unit** does not appear where it should.

85 ";" expected.

A semicolon does not appear where it should.

86 ":" expected.

A colon does not appear where it should.

87 "," expected.

A comma does not appear where it should.

88 "(" expected.

An opening parenthesis does not appear where it should.

89 ")" expected.

A closing parenthesis does not appear where it should.

90 "=" expected.

An equal sign does not appear where it should.

91 ":=" expected.

An assignment operator does not appear where it should.

92 "[" or "(." expected.

A left bracket does not appear where it should.

93 "]" or ".)" expected.

A right bracket does not appear where it should.

94 "." expected.

A period does not appear where it should.

95 ".." expected.

A subrange does not appear where it should.

96 Too many variables.

- The total size of the global variables declared within a program or unit cannot exceed 64K.

- The total size of the local variables declared within a procedure or function cannot exceed 64K.

97 Invalid FOR control variable.

The **for** statement control variable must be a simple variable defined in the declaration part of the current subprogram.

98 Integer variable expected.

The preceding variable must be of an integer type.

99 File and procedure types are not allowed here.

A typed constant cannot be of a file or procedural type.

100 String length mismatch.

The length of the string constant does not match the number of components in the character array.

101 Invalid ordering of fields.

The fields of a record-type constant must be written in the order of declaration.

102 String constant expected.

A string constant does not appear where it should.

103 Integer or real variable expected.

The preceding variable must be of an integer or real type.

104 Ordinal variable expected.

The preceding variable must be of an ordinal type.

105 INLINE error.

The < operator is not allowed in conjunction with relocatable references to variables—such references are always word-sized.

106 Character expression expected.

The preceding expression must be of a Char type.

107 Too many relocation items.

The size of the relocation table part of the .EXE file exceeds 64K, which is Turbo Pascal's upper limit. If you encounter this error, your program is simply too big for Turbo Pascal's linker to handle. It is probably also too big for DOS to execute. You will have to split the program into a "main" part that executes two or more "subprogram" parts using the *Exec* procedure in the *Dos* unit.

112 CASE constant out of range.

For integer type **case** statements, the constants must be within the range –32,768..32,767.

113 Error in statement.

This symbol cannot start a statement.

114 Cannot call an interrupt procedure.

You cannot directly call an interrupt procedure.

116 Must be in 8087 mode to compile this.

This construct can only be compiled in the {**$N+**} state. Operations on the 8087 real types (Single, Double, Extended, and Comp) are not allowed in the {**$N-**} state.

117 Target address not found.

The **S**earch | Find **E**rror command in the IDE or the **/F** option in the command-line version could not locate a statement that corresponds to the specified address.

118 Include files are not allowed here.

Every statement part must be entirely contained in one file.

120 NIL expected.

Typed constants of pointer types may only be initialized to the value **nil**.

121 Invalid qualifier.

You are trying to do one of the following:

- index a variable that is not an array
- specify fields in a variable that is not a record
- dereference a variable that is not a pointer

122 Invalid variable reference.

The preceding construct follows the syntax of a variable reference, but it does not denote a memory location. Most likely, you are calling a pointer function, but forgetting to dereference the result.

123 Too many symbols.

The program or unit declares more than 64K of symbols. If you are compiling with {**$D+**}, try turning it off—note, however, that this will prevent you from finding run-time errors in that module. Otherwise, you could try moving some declarations into a separate unit.

124 Statement part too large.

Turbo Pascal limits the size of a statement part to about 24K. If you encounter this error, move sections of the statement part into one or more procedures. In any case, with a statement part of that size, it's worth the effort to clarify the structure of your program.

126 Files must be var parameters.

You are attempting to declare a file-type value parameter. File-type parameters must be **var** parameters.

127 Too many conditional symbols.

There is not enough room to define further conditional symbols. Try to eliminate some symbols, or shorten some of the symbolic names.

128 Misplaced conditional directive.

The compiler encountered an {**$ELSE**} or {**$ENDIF**} directive without a matching {**$IFDEF**}, {**$IFNDEF**}, or {**$IFOPT**} directive.

129 ENDIF directive missing.

The source file ended within a conditional compilation construct. There must be an equal number of {**$IF***xxx*}s and {**$ENDIF**}s in a source file.

130 Error in initial conditional defines.

The initial conditional symbols specified in **O**ptions | **C**ompiler | **C**onditional Defines (in the IDE) or in a **/D** directive (with the command-line compiler) are invalid. Turbo Pascal expects zero or more identifiers separated by blanks, commas, or semicolons.

131 Header does not match previous definition.

The procedure or function header specified in the **interface** part or **forward** declaration does not match this header.

132 Critical disk error.

A critical error occurred during compilation (for example, drive not ready error).

133 Cannot evaluate this expression.

You are attempting to use a non-supported feature in a constant expression or in a debug expression. For example, you're attempting to use the *Sin* function in a **const** declaration, or you are attempting to call a user-defined function in a debug expression. For a description of the allowed syntax of constant expressions, refer to Chapter 1, "Tokens and constants." For a description of the allowed syntax of debug expressions, refer to Chapter 5 in the *User's Guide*, "Debugging Turbo Pascal programs."

134 Expression incorrectly terminated.

Turbo Pascal expects either an operator or the end of the expression at this point, but neither was found.

135 Invalid format specifier.

You are using an invalid format specifier, or the numeric argument of a format specifier is out of range. For a list of valid format specifiers, refer to Chapter 5 in the *User's Guide*, "Debugging Turbo Pascal programs."

136 Invalid indirect reference.

The statement attempts to make an invalid indirect reference. For example, you are using an **absolute** variable whose base variable is not known in the current module, or you are using an **inline** routine that references a variable not known in the current module.

137 Structured variables are not allowed here.

You are attempting to perform a non-supported operation on a structured variable. For example, you are trying to multiply two records.

138 Cannot evaluate without System unit.

Your TURBO.TPL library must contain the *System* unit for the debugger to be able to evaluate expressions.

139 Cannot access this symbol.

A program's entire set of symbols is available as soon as you have compiled the program. However, certain symbols, such as variables, cannot be accessed until you actually run the program.

140 Invalid floating-point operation.

An operation on two real type values produced an overflow or a division by zero.

141 Cannot compile overlays to memory.

A program that uses overlays must be compiled to disk.

142 Procedural or function variable expected.

In this context, the address operator (@) can only be used with a procedural or function variable.

143 Invalid procedure or function reference.

- You are attempting to call a procedure in an expression.
- If you are going to assign a procedure or function to a procedural variable, it must be compiled in the {**$F+**} state and cannot be declared with **inline** or **interrupt**.

144 Cannot overlay this unit

You are attempting to overlay a unit that wasn't compiled in the {**$O+**} state.

147 Object type expected.

The identifier does not denote an object type.

148 Local object types are not allowed.

Object types can be defined only in the outermost scope of a program or unit. Object-type definitions within procedures and functions are not allowed.

149 VIRTUAL expected.

The reserved word **virtual** is missing.

150 Method identifier expected.

The identifier does not denote a method.

151 Virtual constructors are not allowed.

A constructor method must be static.

152 Constructor identifier expected.

The identifier does not denote a constructor.

153 Destructor identifier expected.

The identifier does not denote a destructor.

154 Fail only allowed within constructors.

The *Fail* standard procedure can be used only within constructors.

155 Invalid combination of opcode and operands.

The assembler opcode does not accept this combination of operands. Possible causes are:

- There are too many or too few operands for this assembler opcode; for example, **INC AX,BX** or **MOV AX**.

- The number of operands is correct, but their types or order do not match the opcode; for example, **DEC 1**, **MOV AX,CL** or **MOV 1,AX**.

156 Memory reference expected.

The assembler operand is not a memory reference, which is required here. Most likely you have forgotten to put square brackets around an index register operand, for example, **MOV AX,BX+SI** instead of **MOV AX,[BX+SI]**.

157 Cannot add or subtract relocatable symbols.

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant. Variables, procedures, functions, and labels are relocatable symbols. Assuming that *Var* is a variable and *Const* is a constant, then the instructions `MOV AX,Const+Const` and `MOV AX,Var+Const` are valid, but `MOV AX,Var+Var` is not.

158 Invalid register combination.

Valid index register combinations are **[BX]**, **[BP]**, **[SI]**, **[DI]**, **[BX+SI]**, **[BX+DI]**, **[BP+SI]**, and **[BP+DI]**. Other index register combinations (such as **[AX]**, **[BP+BX]**, and **[SI+DX]**) are not allowed.

Local variables (variables declared in procedures and functions) are always allocated on the stack and accessed via the BP register. The assembler automatically adds **[BP]** in references to such variables, so even though a construct like **Local[BX]** (where **Local** is a local variable) appears valid, it is not since the final operand would become **Local[BP+BX]**.

159 286/287 instructions are not enabled.

Use a {**$G+**} compiler directive to enable 286/287 opcodes, but be aware that the resulting code cannot be run on 8086 and 8088-based machines.

160 Invalid symbol reference.

This symbol cannot be accessed in an assembler operand. Possible causes follow:

- You are attempting to access a standard procedure, a standard function, or the *Mem*, *MemW*, *MemL*, *Port*, or *PortW* special arrays in an assembler operand.
- You are attempting to access a string, floating-point, or set constant in an assembler operand.
- You are attempting to access an **inline** procedure or function in an assembler operand.
- You are attempting to access the *@Result* special symbol outside a function.
- You are attempting to generate a short **JMP** instruction that jumps to something other than a label.

161 Code generation error.

The preceding statement part contains a **LOOPNE, LOOPE, LOOP,** or **JCXZ** instruction that cannot reach its target label.

162 ASM expected.

Run-time errors

Certain errors at run time cause the program to display an error message and terminate:

```
Run-time error nnn at xxxx:yyyy
```

where *nnn* is the run-time error number, and *xxxx:yyyy* is the run-time error address (segment and offset).

The run-time errors are divided into four categories: DOS errors 1 through 99; I/O errors, 100 through 149; critical errors, 150 through 199; and fatal errors, 200 through 255.

DOS errors

1 Invalid function number.

You made a call to a nonexistent DOS function.

2 File not found.

Reported by *Reset*, *Append*, *Rename*, or *Erase* if the name assigned to the file variable does not specify an existing file.

3 Path not found.

- Reported by *Reset*, *Rewrite*, *Append*, *Rename*, or *Erase* if the name assigned to the file variable is invalid or specifies a nonexistent subdirectory.
- Reported by *ChDir*, *MkDir*, or *RmDir* if the path is invalid or specifies a nonexistent subdirectory.

4 Too many open files.

Reported by *Reset*, *Rewrite*, or *Append* if the program has too many open files. DOS never allows more than 15 open files per process. If you get this error with less than 15 open files, it may indicate that the CONFIG.SYS file does not include a FILES=*xx* entry or that the entry specifies too few files. Increase the number to some suitable value, for instance, 20.

5 File access denied.

- Reported by *Reset* or *Append* if *FileMode* allows writing and the name assigned to the file variable specifies a directory or a read-only file.
- Reported by *Rewrite* if the directory is full or if the name assigned to the file variable specifies a directory or an existing read-only file.
- Reported by *Rename* if the name assigned to the file variable specifies a directory or if the new name specifies an existing file.
- Reported by *Erase* if the name assigned to the file variable specifies a directory or a read-only file.
- Reported by *MkDir* if a file with the same name exists in the parent directory, if there is no room in the parent directory, or if the path specifies a device.
- Reported by *RmDir* if the directory isn't empty, if the path doesn't specify a directory, or if the path specifies the root directory.
- Reported by *Read* or *BlockRead* on a typed or untyped file if the file is not open for reading.
- Reported by *Write* or *BlockWrite* on a typed or untyped file if the file is not open for writing.

6 Invalid file handle.

This error is reported if an invalid file handle is passed to a DOS system call. It should never occur; if it does, it is an indication that the file variable is somehow trashed.

12 Invalid file access code.

Reported by *Reset* or *Append* on a typed or untyped file if the value of *FileMode* is invalid.

15 Invalid drive number.

Reported by *GetDir* or *ChDir* if the drive number is invalid.

16 Cannot remove current directory.

Reported by *RmDir* if the path specifies the current directory.

17 Cannot rename across drives.

Reported by *Rename* if both names are not on the same drive.

I/O errors

These errors cause termination if the particular statement was compiled in the {**$I+**} state. In the {**$I-**} state, the program continues to execute, and the error is reported by the *IOResult* function.

100 Disk read error.

Reported by *Read* on a typed file if you attempt to read past the end of the file.

101 Disk write error.

Reported by *Close, Write, Writeln, Flush,* or *Page* if the disk becomes full.

102 File not assigned.

Reported by *Reset, Rewrite, Append, Rename,* and *Erase* if the file variable has not been assigned a name through a call to *Assign*.

103 File not open.

Reported by *Close, Read, Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead,* or *BlockWrite* if the file is not open.

104 File not open for input.

Reported by *Read, Readln, Eof, Eoln, SeekEof,* or *SeekEoln* on a text file if the file is not open for input.

105 File not open for output.

Reported by *Write* and *Writeln* on a text file if the file is not open for output.

106 Invalid numeric format.

Reported by *Read* or *Readln* if a numeric value read from a text file does not conform to the proper numeric format.

Critical errors

150 Disk is write-protected.

151 Unknown unit.

152 Drive not ready.

153 Unknown command.

154 CRC error in data.

155 Bad drive request structure length.

156 Disk seek error.

157 Unknown media type.

158 Sector not found.

159 Printer out of paper.

160 Device write fault.

161 Device read fault.

162 Hardware failure.

Refer to your DOS programmer's reference manual for more information about critical errors.

Fatal errors

These errors always immediately terminate the program.

200 Division by zero.

The program attempted to divide a number by zero during a **/**, **mod**, or **div** operation.

201 Range check error.

This error is reported by statements compiled in the {**$R+**} state when one of the following situations arises:

- The index expression of an array qualifier was out of range.
- You attempted to assign an out-of-range value to a variable.
- You attempted to assign an out-of-range value as a parameter to a procedure or function.

202 Stack overflow error.

This error is reported on entry to a procedure or function compiled in the {**$S+**} state when there is not enough stack space to allocate the subprogram's local variables. Increase the size of the stack by using the **$M** compiler directive.

This error may also be caused by infinite recursion, or by an assembly language procedure that does not maintain the stack project.

203 Heap overflow error.

This error is reported by *New* or *GetMem* when there is not enough free space in the heap to allocate a block of the requested size.

For a complete discussion of the heap manager, see Chapter 16, "Memory issues."

204 Invalid pointer operation.

This error is reported by *Dispose* or *FreeMem* if the pointer is **nil** or points to a location outside the heap, or if the free list cannot be expanded due to a full free list or to *HeapPtr* being too close to the bottom of the free list.

205 Floating point overflow.

A floating-point operation produced a number too large for Turbo Pascal or the numeric coprocessor (if any) to handle.

206 Floating point underflow

A floating-point operation produced an underflow. This error is only reported if you are using the 8087 numeric coprocessor with a control word that unmasks underflow exceptions. By default, an underflow causes a result of zero to be returned.

207 Invalid floating point operation

- The real value passed to *Trunc* or *Round* could not be converted to an integer within the Longint range (-2,147,483,648 to 2,147,483,647).
- The argument passed to the *Sqrt* function was negative.
- The argument passed to the *Ln* function was zero or negative.
- An 8087 stack overflow occurred. For further details on correctly programming the 8087, refer to Chapter 14, "Using the 8087."

208 Overlay manager not installed

Your program is calling an overlaid procedure or function, but the overlay manager is not installed. Most likely, you are not calling *OvrInit*, or the call to *OvrInit* failed. Note that, if you have initialization code in any of your overlaid units, you must create an additional non-overlaid unit which calls *OvrInit*, and use that unit before any of the overlaid units. For a complete description of the overlay manager, refer to Chapter 13, "The Overlay unit."

209 Overlay file read error

A read error occurred when the overlay manager tried to read an overlay from the overlay file.

210 Object not initialized

With range-checking on, you made a call to an object's virtual method, before the object had been initialized via a constructor call.

211 Call to abstract method.

This error is generated by the *Abstract* procedure in the *Objects* unit; it indicates that your program tried to execute an abstract virtual method. When an object type contains one or more abstract methods it is called an *abstract object type*. It is an error to instantiate objects of an abstract type—abstract object types exist only so that you can inherit from them and override the abstract methods.

For example, the *Compare* method of the *TSortedCollection* type in the *Objects* unit is abstract, indicating that to implement a sorted collection you must create an object type that inherits from *TSortedCollection* and overrides the *Compare* method.

212 Stream registration error.

This error is generated by the *RegisterType* procedure in the *Objects* unit indicating that one of the following errors have occurred:

- The stream registration record does not reside in the data segment.
- The *ObjType* field of the stream registration record is zero.
- The type has already been registered.
- Another type with the same *ObjType* value already exists.

213 Collection index out of range.

The index passed to a method of a *TCollection* is out of range.

214 Collection overflow error.

The error is reported by a *TCollection* if an attempt is made to add an element when the collection cannot be expanded.

A P P E N D I X

B

Reference materials

This appendix is devoted to certain reference materials: an ASCII table, keyboard scan codes, and extended codes.

ASCII codes

The American Standard Code for Information Interchange (ASCII) is a code that translates alphabetic and numeric characters and symbols and control instructions into 7-bit binary code. Table B.1 shows both printable characters and control characters.

Table B.1
ASCII table

The caret in ^@ means to press the Ctrl key and type @.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	^@ NUL	32	20		64	40	@	96	60	'
1	1	☺ SOH	33	21	!	65	41	A	97	61	a
2	2	☻ STX	34	22	"	66	42	B	98	62	b
3	3	♥ ETX	35	23	#	67	43	C	99	63	c
4	4	♦ EOT	36	24	$	68	44	D	100	64	d
5	5	♣ ENQ	37	25	%	69	45	E	101	65	e
6	6	♠ ACK	38	26	&	70	46	F	102	66	f
7	7	• BEL	39	27	'	71	47	G	103	67	g
8	8	◘ BS	40	28	(	72	48	H	104	68	h
9	9	○ TAB	41	29	)	73	49	I	105	69	i
10	A	◙ LF	42	2A	*	74	4A	J	106	6A	j
11	B	♂ VT	43	2B	+	75	4B	K	107	6B	k
12	C	♀ FF	44	2C	,	76	4C	L	108	6C	l
13	D	♪ CR	45	2D	-	77	4D	M	109	6D	m
14	E	♫ SO	46	2E	.	78	4E	N	110	6E	n
15	F	☼ SI	47	2F	/	79	4F	O	111	6F	o
16	10	► DLE	48	30	0	80	50	P	112	70	p
17	11	◄ DC1	49	31	1	81	51	Q	113	71	q
18	12	↕ DC2	50	32	2	82	52	R	114	72	r
19	13	‼ DC3	51	33	3	83	53	S	115	73	s
20	14	¶ DC4	52	34	4	84	54	T	116	74	t
21	15	§ NAK	53	35	5	85	55	U	117	75	u
22	16	▬ SYN	54	36	6	86	56	V	118	76	v
23	17	↨ ETB	55	37	7	87	57	W	119	77	w
24	18	↑ CAN	56	38	8	88	58	X	120	78	x
25	19	↓ EM	57	39	9	89	59	Y	121	79	y
26	1A	→ SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	← ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	└ FS	60	3C	<	92	5C	\	124	7C	|
29	1D	↔ GS	61	3D	=	93	5D	]	125	7D	}
30	1E	▲ RS	62	3E	>	94	5E	^	126	7E	~
31	1F	▼ US	63	3F	?	95	5F	_	127	7F	⌂

Table B.1: ASCII table (continued)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	∈
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	⣿	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	•
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	▉	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	n
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	2
158	9E	Pt	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Extended key codes

Extended key codes are returned by those keys or key combinations that cannot be represented by the standard ASCII codes listed in Table B.1. (See *ReadKey* in Chapter 1 of the *Library Reference* for a description about how to determine if an extended key has been pressed.)

Table B.2 shows the second code and what it means.

Table B.2
Extended key codes

Second code	Meaning
3	*NUL* (null character)
15	*Shift Tab* (—<vv)
16-25	Alt-Q/W/E/R/T/Y/U/I/O/P
30-38	*Alt-A/S/D/F/G/H/I/J/K/L*
44-50	*Alt-Z/X/C/V/B/N/M*
59-68	Keys *F1-F10* (disabled as softkeys)
71	*Home*
72	↑
73	*PgUp*
75	←
77	→
79	*End*
80	↓
81	*PgDn*
82	*Ins*
83	*Del*
84-93	*F11-F20* (*Shift-F1* to *Shift-F10*)
94-103	*F21-F30* (*Ctrl-F1* through *F10*)
104-113	*F31-F40* (*Alt-F1* through *F10*)
114	*Ctrl-PrtSc*
115	*Ctrl-←*
116	*Ctrl-→*
117	*Ctrl-End*
118	*Ctrl-PgDn*
119	*Ctrl-Home*
120-131	*Alt-1/2/3/4/5/6/7/8/9/0/-/=*
132	*Ctrl-PgUp*
133	*F11*
134	*F12*
135	*Shift-F11*
136	*Shift-F12*
137	*Ctrl-F11*
138	*Ctrl-F12*
139	*Alt-F11*
140	*Alt-F12*

Keyboard scan codes

Keyboard scan codes are the codes returned from the keys on the IBM PC keyboard, as they are seen by Turbo Pascal. These keys are useful when you're working at the assembly language level. Note that the keyboard scan codes displayed in Table B.3 are in hexadecimal values.

Table B.3
Keyboard scan codes

Key	Scan Code in Hex	Key	Scan Code in Hex
Esc	01	←/→	0F
! 1	02	Q	10
@ 2	03	W	11
# 3	04	E	12
$ 4	05	R	13
% 5	06	T	14
^ 6	07	Y	15
& 7	08	U	16
* 8	09	I	17
(9	0A	O	18
) 0	0B	P	19
_ -	0C	{ [	1A
+ =	0D	}]	1B
Backspace	0E	Return	1C
Ctrl	1D	| \	2B
A	1E	Z	2C
S	1F	X	2D
D	20	C	2E
F	21	V	2F
G	22	B	30
H	23	N	31
J	24	M	32
K	25	< ,	33
L	26	> .	34
: ;	27	? /	35
" '	28	→Shift	36
~ `	29	PrtSc*	37
←Shift	2A	Alt	38
Spacebar	39	7 Home	47
Caps Lock	3A	8 ↑	48
F1	3B	9 PgUp	49
F2	3C	Minus sign	4A
F3	3D	4 ←	4B
F4	3E	5	4C
F5	3F	6 →	4D
F6	40	+	4E
F7	41	1 End	4F
F8	42	2 ↓	50
F9	43	3 PgDn	51
F10	44	0 Ins	52
F11	D9	Del	53
F12	DA	Num Lock	45
Scroll Lock	46		

I N D E X

C

D

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

6.0

PROGRAMMER'S GUIDE

TURBO PASCAL®

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300
Offices in: Australia, Denmark, England, France, Germany, Italy, Japan and Sweden ■ Part# 11MN-PAS05-60 ■ BOR 1851