# Contents

**Introduction**  
What’s in this manual .......................... 1

**Part 1 The Turbo Pascal standard**

**Chapter 1 Tokens and constants**  
Special symbols and reserved words ........ 5  
Identifiers .................................. 7  
Labels ....................................... 8  
Numbers ...................................... 9  
Character strings ............................ 10  
Constant declarations ...................... 12  
Comments ................................... 13  
Program lines ................................ 13

**Chapter 2 Blocks, locality, and scope**  
Syntax ....................................... 15  
Rules of scope ................................ 17  
Scope of interface and standard identifiers ........................................ 18

**Chapter 3 Types**  
Simple types ................................ 22  
Ordinal types ................................ 22  
Integer types ................................ 23  
Boolean types ................................ 24  
Char type ................................... 24  
Enumerated types ............................ 25  
Subrange types ................................ 25  
Real types .................................... 26  
Software floating point ...................... 27  
8087 floating point .......................... 27  
String types ................................ 27  
Structured types ............................. 28  
Array types ................................ 29  
Record types ................................ 30

Object types ................................ 32  
Components and scope ....................... 35  
Methods .................................... 35  
Virtual methods .............................. 36  
Instantiating objects ....................... 36  
Set types ................................... 38  
File types ................................... 39  
Pointer types ................................ 39  
Procedural types ............................. 40  
Identical and compatible types .............. 41  
Type identity ............................... 41  
Type compatibility ......................... 42  
Assignment compatibility ................... 43  
The type declaration part ................... 44

**Chapter 4 Variables**  
Variable declarations ....................... 47  
The data segment ............................ 48  
The stack segment ........................... 48  
Absolute variables .......................... 49  
Variable references ......................... 50  
Qualifiers ................................... 50  
Arrays, strings, and indexes ............... 51  
Records and field designators ............. 52  
Object component designators ............. 52  
Pointers and dynamic variables ............ 53  
Variable typecasts .......................... 53

**Chapter 5 Typed constants**  
Simple-type constants ....................... 58  
String-type constants ....................... 59  
Structured-type constants ................... 59  
Array-type constants ....................... 59  
Record-type constants ...................... 60  
Object-type constants ...................... 61  
Set-type constants .......................... 62  
Pointer-type constants ..................... 62
Chapter 6 Expressions

Expression syntax .................................. 66

Operators ........................................... 69
  Arithmetic operators ............................. 69
  Logical operators ............................... 70
  Boolean operators ............................... 70
  String operator .................................. 72
  Set operators .................................... 72
  Relational operators ............................ 72
    Comparing simple types ...................... 73
    Comparing strings ............................. 73
    Comparing packed strings .................... 74
    Comparing pointers ........................... 74
    Comparing sets ................................ 74
    Testing set membership ...................... 74
  The @ operator .................................. 74
    @ with a variable .............................. 75
    @ with a value parameter ..................... 75
    @ with a variable parameter ................. 75
    @ with a procedure or function ............. 76
    @ with a method ................................ 76

Function calls .................................... 76

Set constructors .................................. 77

Value typecasts ................................... 78

Procedural types in expressions .................. 79

Chapter 7 Statements

Simple statements ................................ 81
  Assignment statements .......................... 82
    Object type assignments ...................... 82
  Procedure statements ........................... 83
    Method, constructor, and destructor calls ................................................. 83
  Goto statements ................................ 84

Structured statements ............................ 85
  Compound statements ............................ 85
  Conditional statements ........................ 86
    If statements ................................. 86
    Case statements .............................. 87
  Repetitive statements ........................... 88
    Repeat statements ............................ 88
    While statements .............................. 89

For statements ................................... 90

With statements .................................. 92

Chapter 8 Procedures and functions ............. 95

Procedure declarations ........................... 95
  Near and far declarations ...................... 97
  Interrupt declarations ........................ 97
  Forward declarations ........................... 98
  External declarations ........................ 99
  Assembler declarations ......................... 99
  Inline declarations ............................ 99
  Function declarations .......................... 100
  Method declarations ............................ 102
  Constructors and destructors ................... 103
  Parameters ...................................... 105
    Value parameters .............................. 106
    Variable parameters ........................... 106
    Objects ....................................... 107
    Untyped variable parameters ................. 107
  Procedural types ............................... 108
  Procedural variables ........................... 108
  Procedural-type parameters .................... 111

Chapter 9 Programs and units ...................... 113

Program syntax .................................... 113
  The program heading ............................ 113
  The uses clause ................................ 114

Unit syntax ....................................... 114
  The unit heading ................................ 115
  The interface part .............................. 115
  The implementation part ....................... 116
  The initialization part ....................... 117
  Indirect unit references ....................... 117
  Circular unit references ....................... 118
  Sharing other declarations .................... 120

Part 2 The standard libraries

Chapter 10 The System unit ....................... 125

Standard procedures and functions ............... 125
  Flow control procedures ....................... 125
  Dynamic allocation procedures ................. 125
  Dynamic allocation functions ................... 126
  Transfer functions ............................. 126
  Arithmetic functions ........................... 127
Constructor error recovery 236

Chapter 18 Control issues 241
Calling conventions 241
Variable parameters 242
Value parameters 242
Function results 243
NEAR and FAR calls 243
Nested procedures and functions 244
Entry and exit code 245
Register-saving conventions 246
Exit procedures 246
Interrupt handling 248
Writing interrupt procedures 248

Chapter 19 Input and output issues 251
Text file device drivers 251
The Open function 252
The InOut function 253
The Flush function 253
The Close function 253
Direct port access 253

Chapter 20 Automatic optimizations 255
Constant folding 255
Constant merging 256
Short-circuit evaluation 256
Order of evaluation 256
Range checking 257
Shift instead of multiply 257
Automatic word alignment 257
Dead code removal 258
Smart linking 258

Chapter 21 Compiler directives 261
Switch directives 262
Align data 262
Boolean evaluation 263
Debug information 264
Emulation 265
Force far calls 265
Generate 80286 code 266
Input/output checking 266
Local symbol information 267
Numeric processing 268
Overlay code generation 268
Range checking 269
Stack-overflow checking 269
Var-string checking 270
Extended syntax 270
Parameter directives 271
Include file 271
Link object file 271
Memory allocation sizes 272
Overlay unit name 272
Conditional compilation 273
Conditional symbols 274
The DEFINE directive 275
The UNDEF directive 276
The IFDEF directive 276
The IFNDEF directive 276
The IFOPT directive 276
The ELSE directive 277
The ENDIF directive 277

Part 4 Using Turbo Pascal with assembly language

Chapter 22 The inline assembler 281
The asm statement 282
Register use 283
Assembler statement syntax 283
Labels 283
Prefix opcodes 285
Instruction opcodes 285
RET instruction sizing 286
Automatic jump sizing 286
Assembler directives 287
Operands 289
Expressions 290
Differences between Pascal and
Assembler expressions 290
Expression elements 291
Constants 291
Numeric constants 291
String constants 292
Registers 293
Symbols 294
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Turbo Pascal reserved words</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Turbo Pascal standard directives</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Predefined integer types</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Real data types</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>Precedence of operators</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>Binary arithmetic operations</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>Unary arithmetic operations</td>
<td>69</td>
</tr>
<tr>
<td>6.4</td>
<td>Logical operations</td>
<td>70</td>
</tr>
<tr>
<td>6.5</td>
<td>Boolean operations</td>
<td>71</td>
</tr>
<tr>
<td>6.6</td>
<td>String operation</td>
<td>72</td>
</tr>
<tr>
<td>6.7</td>
<td>Set operations</td>
<td>72</td>
</tr>
<tr>
<td>6.8</td>
<td>Relational operations</td>
<td>73</td>
</tr>
<tr>
<td>6.9</td>
<td>Pointer operation</td>
<td>75</td>
</tr>
<tr>
<td>12.1</td>
<td>Graph unit driver and mode</td>
<td>159</td>
</tr>
<tr>
<td>12.2</td>
<td>GraphResult error values</td>
<td>160</td>
</tr>
<tr>
<td>12.3</td>
<td>Graph unit procedures</td>
<td>165</td>
</tr>
<tr>
<td>12.4</td>
<td>Graph unit functions</td>
<td>167</td>
</tr>
<tr>
<td>13.1</td>
<td>OvrResult values</td>
<td>177</td>
</tr>
<tr>
<td>22.1</td>
<td>Values, classes, and types of symbols</td>
<td>295</td>
</tr>
<tr>
<td>22.2</td>
<td>Predefined type symbols</td>
<td>300</td>
</tr>
<tr>
<td>22.3</td>
<td>Inline assembler expression operators</td>
<td>300</td>
</tr>
<tr>
<td>B.1</td>
<td>ASCII table</td>
<td>348</td>
</tr>
<tr>
<td>B.2</td>
<td>Extended key codes</td>
<td>350</td>
</tr>
<tr>
<td>B.3</td>
<td>Keyboard scan codes</td>
<td>352</td>
</tr>
</tbody>
</table>
12.1: Screen with xy-coordinates ........ 151
13.1: Loading and disposing overlays .... 172
16.1: Turbo Pascal memory map .......... 210
16.2: Disposal method using mark and
       release .................................. 212
16.3: Heap layout with Release(P) 
       executed ............................... 213
16.4: Creating a "hole" in the heap ....... 214
16.5: Enlarging the free block ............ 214
16.6: Releasing the free block .......... 215
17.1: Layouts of instances of Location, Point, 
       and Circle .......................... 226
17.2: Point and Circle's VMT layouts ... 228
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.
The Turbo Pascal standard
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.

```
letter
A ... Z a ... z
```

```
digit
0 ... 9
```

```
hex digit
A ... F a ... f
```

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 (I) 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>
<thead>
<tr>
<th>and</th>
<th>end</th>
<th>nil</th>
<th>shr</th>
</tr>
</thead>
<tbody>
<tr>
<td>asm</td>
<td>file</td>
<td>not</td>
<td>string</td>
</tr>
<tr>
<td>array</td>
<td>for</td>
<td>object</td>
<td>then</td>
</tr>
<tr>
<td>begin</td>
<td>function</td>
<td>of</td>
<td>to</td>
</tr>
<tr>
<td>case</td>
<td>goto</td>
<td>or</td>
<td>type</td>
</tr>
<tr>
<td>const</td>
<td>if</td>
<td>packed</td>
<td>unit</td>
</tr>
<tr>
<td>constructor</td>
<td>implementation</td>
<td>procedure</td>
<td>until</td>
</tr>
<tr>
<td>destructor</td>
<td>in</td>
<td>program</td>
<td>uses</td>
</tr>
<tr>
<td>div</td>
<td>inline</td>
<td>record</td>
<td>var</td>
</tr>
<tr>
<td>do</td>
<td>interface</td>
<td>repeat</td>
<td>while</td>
</tr>
<tr>
<td>downto</td>
<td>label</td>
<td>set</td>
<td>with</td>
</tr>
<tr>
<td>else</td>
<td>mod</td>
<td>shl</td>
<td>xor</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>absolute</th>
<th>external</th>
<th>forward</th>
<th>near</th>
</tr>
</thead>
<tbody>
<tr>
<td>assembler</td>
<td>far</td>
<td>interrupt</td>
<td>private</td>
</tr>
<tr>
<td>virtual</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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**.

---

Units are described in Chapter 3 of the User's Guide and Chapter 9 of this manual.
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 $7 \times 10^{-2}$; 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 $FFFFFFFE$. 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.

A constant declaration consists of a constant identifier, an equal sign, and a constant expression:

constant declaration → identifier = constant ;

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

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)

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

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

For expression syntax, see Chapter 6, "Expressions."
The following standard functions are allowed in constant expressions:

- `Abs`  
- `Length`  
- `Ord`  
- `Round`  
- `Swap`  
- `Chr`  
- `Lo`  
- `Pred`  
- `SizeOf`  
- `Trunc`  
- `Hi`  
- `Odd`  
- `Ptr`  
- `Succ`  
- `Round`  
- `SizeD!`  
- `Swap`  
- `Trunc`  

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

```pascal
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = 'Error: ' + Message + ' ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;

Comments

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

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

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

Program lines

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

```
block → declaration part → statement part
```
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.

```
variable declaration part → var → variable declaration
```

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

```
procedure/function declaration part

  procedure declaration

  function declaration

  constructor declaration

  destructor declaration
```

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

```
statement part → compound statement
```

**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 redefine the standard identifiers, but a specific reference can still be made through a qualified identifier, for example, System.Integer or System.Writeln.
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.

```
<table>
<thead>
<tr>
<th>simple type</th>
<th>ordinal type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real type</td>
</tr>
</tbody>
</table>
```

```
| real type | real type identifier |
```

*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 \textit{Ord} can be applied to any ordinal-type value to return the ordinality of the value.
- The standard function \textit{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, \textit{Pred} produces an error.
- The standard function \textit{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, \textit{Succ} produces an error.

The syntax of an ordinal type follows:

\begin{center}
\begin{tikzpicture}
  \node (ordinal) {ordinal type};
  \node (subrange) at (ordinal -| -0.5) {subrange type};
  \node (enumerated) at (subrange |- 0.5) {enumerated type};
  \node (identifier) at (enumerated |- 0.5) {ordinal type identifier};
  \draw [->] (ordinal) -- (subrange);
  \draw [->] (subrange) -- (enumerated);
  \draw [->] (enumerated) -- (identifier);
\end{tikzpicture}
\end{center}

Turbo Pascal has seven predefined ordinal types: \texttt{Integer}, \texttt{Shortint}, \texttt{Longint}, \texttt{Byte}, \texttt{Word}, \texttt{Boolean}, and \texttt{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: \texttt{Shortint}, \texttt{Integer}, \texttt{Longint}, \texttt{Byte}, and \texttt{Word}. Each type denotes a specific subset of the whole numbers, according to the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortint</td>
<td>-128..127</td>
<td>Signed 8-bit</td>
</tr>
<tr>
<td>Integer</td>
<td>-32768..32767</td>
<td>Signed 16-bit</td>
</tr>
<tr>
<td>Longint</td>
<td>-2147483648..2147483647</td>
<td>Signed 32-bit</td>
</tr>
<tr>
<td>Byte</td>
<td>0..255</td>
<td>Unsigned 8-bit</td>
</tr>
<tr>
<td>Word</td>
<td>0..65535</td>
<td>Unsigned 16-bit</td>
</tr>
</tbody>
</table>
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
- \(\text{Ord}(\text{False}) = 0\)
- \(\text{Ord}(\text{True}) = 1\)
- \(\text{Succ}(\text{False}) = \text{True}\)
- \(\text{Pred}(\text{True}) = \text{False}\)

Char type

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

A string constant of length 1 can denote a constant character value. Any character value can be generated with the standard function \(\text{Chr}\).
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.

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:

```plaintext
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:

\[
0..99 \\
-128..127 \\
\text{Club..Heart}
\]

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:

```pascal
const
  X = 50;
  Y = 10;

type
  Color = (Red, Green, Blue);
  Scale = (X - Y) \times 2 .. (X + Y) \times 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:

```pascal

```

### 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>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Significant digits</th>
<th>Size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>(2.9 \times 10^{-39} \ldots 1.7 \times 10^{38})</td>
<td>11-12</td>
<td>6</td>
</tr>
<tr>
<td>Single</td>
<td>(1.5 \times 10^{-45} \ldots 3.4 \times 10^{38})</td>
<td>7-8</td>
<td>4</td>
</tr>
<tr>
<td>Double</td>
<td>(5.0 \times 10^{-324} \ldots 1.7 \times 10^{308})</td>
<td>15-16</td>
<td>8</td>
</tr>
<tr>
<td>Extended</td>
<td>(3.4 \times 10^{-4932} \ldots 1.1 \times 10^{4932})</td>
<td>19-20</td>
<td>10</td>
</tr>
<tr>
<td>Comp</td>
<td>(-2^{63}+1 \ldots 2^{63}-1)</td>
<td>19-20</td>
<td>8</td>
</tr>
</tbody>
</table>

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.

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`. 

---

Chapter 3, Types 27
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.

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

Structured types

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

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.
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:

```plaintext
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,

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

is interpreted the same way by the compiler as

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

You can also express

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

as

```plaintext
packed array[1..10,1..8] of Boolean
```
You access an array's components by supplying the array's identifier with one or more indexes in brackets.

An array type of the form

\[
\text{packed array}[M..N] \text{ of Char}
\]

where \( M \) is less than \( N \) is called a packed string type (the word \text{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:

\[
\text{type}
\]

\[
\text{DateRec} = \text{record}
\]
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: (Side1, 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.

```pascal
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;
Components and scope

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.

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.

```pascal
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:

```pascal
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,

```pascal
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:

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

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:

```pascal
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

\[
T1 = \text{Integer}; \\
T2 = T1; \\
T3 = \text{Integer}; \\
T4 = T2;
\]

result in \( T1, T2, T3, T4 \), and \text{Integer} as identical types. The type declarations

\[
T5 = \text{set of Integer}; \\
T6 = \text{set of Integer};
\]

don’t make \( T5 \) and \( T6 \) identical, since \text{set of Integer} is not a type identifier. Two variables declared in the same declaration, for example,

\[
V1, V2: \text{set of Integer};
\]

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

\[
V1: \text{set of Integer}; \\
V2: \text{set of Integer}; \\
V3: \text{Integer}; \\
V4: \text{Integer};
\]

mean \( V3 \) and \( V4 \) are of identical type, but not \( V1 \) and \( V2 \).

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.
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 $T_1$. 

- 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.
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:

```pascal
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;
```

Turbo Pascal Programmer's Guide
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.
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 re-declared 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:
The data segment

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

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

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:

```
  absolute clause  
  \begin{array}{*{20}c}
    \text{absolute} & \text{unsigned integer} & : & \text{unsigned integer} & \text{variable identifier}
  \end{array}
```

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 \texttt{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 \texttt{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

```
variable reference ← variable identifier
variable typecast
function call
```

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.

```
qualifier ← index
field designator
```

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, \( \text{Length}(S) \) is the same as \( \text{Ord}(S[0]) \). 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.

\[
\text{field designator} \rightarrow \text{field identifier}
\]

Some examples of a field designator include the following:

\begin{verbatim}
Today.Year
Results[1].Count
Results[1].When.Month
\end{verbatim}

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.
Pointers and dynamic variables

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.

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:

\[
\begin{align*}
P^1 \\
P^1^.Sibling^ \\
\text{Results[1].Data}^ \\
\end{align*}
\]

Variable typecasts

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

\[
\text{variable typecast}
\]

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:

```pascal
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

```pascal
type
  Func = function(X: Integer): Integer;

var
  F: Func;
  P: Pointer;
  N: Integer;
```

you can construct the following assignments:
\begin{verbatim}
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 \}
\end{verbatim}

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.
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:

```pascal
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,

```pascal
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.

```pascal
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:

```
class
    Heading: string[7] = 'Section';
    TrueStr: string[5] = 'Yes';
    FalseStr: string[5] = 'No';
end
```

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.

```
array constant ( \{ \  \} )
```

An example of an array-type constant follows:

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

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:

```pascal
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:

```pascal
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'..'9' = ['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:

```pascal
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);
```
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:

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

procedure DefaultError(ErrorCode: Integer); far;
begin
  WriteLn('fError f f f ErrorCode f f .f);
end;

const
  ErrorHandler: ErrorProc = DefaultError;
```
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>
<thead>
<tr>
<th>Operators</th>
<th>Precedence</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>@, not</td>
<td>first (high)</td>
<td>unary operators</td>
</tr>
<tr>
<td>*, /, div, mod, and, shl, shr</td>
<td>second</td>
<td>multiplying operators</td>
</tr>
<tr>
<td>+, -, or, xor</td>
<td>third</td>
<td>adding operators</td>
</tr>
<tr>
<td>=, &lt;&gt;, &lt;, &gt;, &lt;=, &gt;=, in</td>
<td>fourth (low)</td>
<td>relational operators</td>
</tr>
</tbody>
</table>

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:

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

A set constructor denotes a value of a set type.

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:

```
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:

\[ X + Y \]
\[ -X \]
\[ Huel + Hue2 \]
\[ I * J + 1 \]

An expression applies the relational operators to simple expressions:

Here are some examples of expressions:

\[ X = 1.5 \]
\[ Done <> Error \]
\[ (I < J) = (J < K) \]
\[ C in Huel \]
Operators

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>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real type</td>
<td>real type</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real type</td>
<td>real type</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real type</td>
<td>real type</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>integer type</td>
<td>real type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real type</td>
<td>real type</td>
</tr>
<tr>
<td>div</td>
<td>integer division</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td>mod</td>
<td>remainder</td>
<td>integer type</td>
<td>integer type</td>
</tr>
</tbody>
</table>

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.

Table 6.3

Unary arithmetic operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>sign identity</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real type</td>
<td>real type</td>
</tr>
<tr>
<td>-</td>
<td>sign negation</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real type</td>
<td>real type</td>
</tr>
</tbody>
</table>

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

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

If both operands of a +, −, *, div, or mod operator are of an integer type, the result type is of the common type of the two operands.

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,

$$I \mod J = I - (I \div J) \times J$$

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>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>not</code></td>
<td>bitwise negation</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td><code>and</code></td>
<td>bitwise and</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td><code>or</code></td>
<td>bitwise or</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td><code>xor</code></td>
<td>bitwise xor</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td><code>shl</code></td>
<td>shift left</td>
<td>integer type</td>
<td>integer type</td>
</tr>
<tr>
<td><code>shr</code></td>
<td>shift right</td>
<td>integer type</td>
<td>integer type</td>
</tr>
</tbody>
</table>

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 $I \text{ shl } J$ and $I \text{ shr } J$ 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.
The not operator is a unary operator.

### Table 6.5
Boolean operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>negation</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>and</td>
<td>logical and</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>or</td>
<td>logical or</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>xor</td>
<td>logical xor</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

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:

```pascal
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 **Options | Compiler** 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>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>concatenation</td>
<td>string type, Char type, or packed string type</td>
<td>string type</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand types</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>union</td>
<td>compatible set types</td>
</tr>
<tr>
<td>-</td>
<td>difference</td>
<td>compatible set types</td>
</tr>
<tr>
<td>*</td>
<td>intersection</td>
<td>compatible set types</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Operation</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal</td>
<td>compatible simple, pointer, set, string, or packed string types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
<td>compatible simple, pointer, set, string, or packed string types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>compatible simple, string, or packed string types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>compatible simple, string, or packed string types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less or equal</td>
<td>compatible simple, string, or packed string types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal</td>
<td>compatible simple, string, or packed string types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td>subset of</td>
<td>compatible set types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;=</td>
<td>superset of</td>
<td>compatible set types</td>
<td>Boolean</td>
</tr>
<tr>
<td>in</td>
<td>member of</td>
<td>left operand: any ordinal type $T$; right operand: set whose base is compatible with $T$.</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

**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 $N$.  

*Chapter 6, Expressions*  
73
Comparing packed strings

The relational operators =, $\neq$, $<$, $>$, $\geq$, and $\leq$ 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 $\neq$ 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, \( A \neq B \).
- \( A \leq B \) is True only if every member of \( A \) is also a member of \( B \).
- \( A \geq B \) 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Pointer formation</td>
<td>Variable reference or procedure or function identifier</td>
<td>Pointer (same as nil)</td>
</tr>
</tbody>
</table>

@ 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

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

then the statement

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

causes `TwoCharPtr` to point to `Int`. `TwoCharPtr^` becomes a reinterpretation 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

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

Some examples of function calls follow:

- \texttt{Sum(A, 63)}
- \texttt{Maximum(147, J)}
- \texttt{Sin(X + Y)}
- \texttt{Eof(F)}
- \texttt{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.

In the extended syntax ($\texttt{Sx+}$) 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:

\[
[\text{red, C, green}]
\]
\[
[1, 5, 10..X \mod 12, 23]
\]
\[
['A'..'Z', 'a'..'z', \text{Chr(Digit + 48)}]
\]

**Value typecasts**

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

\[
\text{value typecast} \rightarrow \text{type identifier} \rightarrow (\text{expression}) \rightarrow \}
\]

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

The syntax of a value typecast is almost identical to that of a variable typecast. 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)
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:

```pascal
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 = \text{ReadInt} \ then \\
\text{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 \text{ReadInt}, and then compare the returned values. To compare the procedural value in \( F \) to the procedural value of \text{ReadInt}, the following construct must be used:

\[
\text{if } @F = @\text{ReadInt} \text{ then} \\
\text{WriteLn('Equal');}
\]

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 @\text{ReadInt} returns the address of \text{ReadInt}; the two pointer values can then be compared to determine if \( F \) currently refers to \text{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 \( P \).


Statements describe algorithmic actions that can be executed. Labels can prefix statements, and these labels can be referenced by \texttt{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 \textit{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.

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:

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

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 \(X, Y, \text{Len}, \text{Name}\).

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.

```
procedure statement
    procedure identifier
        actual parameter list
```

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:

```pascal
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:

```
goto statement -> goto -> label
```

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).

```
structured statement
    \rightarrow compound statement
    \rightarrow conditional statement
    \rightarrow repetitive statement
    \rightarrow with statement
```

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.

```
compound statement
    \rightarrow begin
    \rightarrow statement
    \rightarrow end
    \rightarrow ;
```

Here's an example of a compound statement:

```
begin
    Z := X;
    X := Y;
    Y := Z;
end;
```
A conditional statement selects for execution a single one (or none) of its component statements.

The syntax for an if statement reads like this:

```
if expression then statement
else statement
```

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:

```plaintext
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.

```
repetitive statement
```

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
  statement
until expression
```

The expression must produce a result of type Boolean. The statements between the symbols `repeat` and `until` are executed
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).

```
while statement  while expression do 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.

```
for statement  ────┐
               └───┐
               └───┘
       ┌─┐    ┌─┐
       │ for │  ┌─┐
       └───┘  └───┘
               ┌─┐
               │ to │
               └───┘
               ┌─┐
               │ do │
               └───┘
       ┌─┐    ┌─┐
       │ control variable │ initial value
       └───┘  └───┘
               ┌─┐
               │ final value │
               └───┘
```

- Control variable → variable identifier
- Initial value → expression
- Final value → expression

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** 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

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

is equivalent to

```plaintext
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

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

is equivalent to

```plaintext
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:

with statement

with \[record or object\] do \[variable reference\] 

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:

```pascal
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

```pascal
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

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

is equivalent to

```pascal
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.
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.

procedure declaration

\[
\text{procedure declaration} \quad \rightarrow \quad ; \quad \text{procedure body} \quad \rightarrow \quad ;
\]
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:

```pascal
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:

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

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

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:

```pascal
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.

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;`

Assembler declarations

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

```
asm block
```

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

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

A function is activated by the evaluation of a **function** call. The **function** call gives the function's identifier and 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:

```pascal
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:

```pascal
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);
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:

```pascal
constructor Field.Copy(var F: Field);
begin
  Self := F;
end;
```
Destructors can be virtual, and often are. Destructors seldom take any parameters.

```pascal
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 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:

```pascal
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 $T_1$, the actual parameter might be of type $T_2$ if $T_2$ is in the domain of $T_1$. 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:

```pascal
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

```pascal
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
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:

```pascal
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,

```pascal
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:

$P := \text{Swap};$

$F := \text{Tan};$

Following these assignments, the call $P(I, J)$ is equivalent to $\text{Swap}(I, J)$, and $F(X)$ is equivalent to $\text{Tan}(X)$.

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

Standard procedures and functions are the procedures and functions declared by the $\text{System}$ unit, such as $\text{WriteLn}$, $\text{ReadLn}$, $\text{Chr}$, and $\text{Ord}$. 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

```pascal
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.

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:

```pascal
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
        begin
          Write(Operation(X, Y):5);
          Writeln;
        end;
      Writeln;
    end;
end;
```

**Chapter 8, Procedures and functions**
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:

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

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

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

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 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 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.

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:

```pascal
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:

```pascal
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:

```pascal
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:

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

interface
procedu_re showError(ErrMsg: String);
implementation
uses
    Display;

procedu_re showError(ErrMsg: String);
begi
    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 `interface` 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:

```pascal
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

The standard libraries
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

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.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Exit</code></td>
<td>Exits immediately from the current block.</td>
</tr>
<tr>
<td><code>Halt</code></td>
<td>Stops program execution and returns to the operating system.</td>
</tr>
<tr>
<td><code>RunError</code></td>
<td>Stops program execution and generates a run-time error.</td>
</tr>
</tbody>
</table>
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.”

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispose</td>
<td>Disposes a dynamic variable.</td>
</tr>
<tr>
<td>FreeMem</td>
<td>Disposes a dynamic variable of a given size.</td>
</tr>
<tr>
<td>GetMem</td>
<td>Creates a new dynamic variable of a given size and sets a pointer variable to point to it.</td>
</tr>
<tr>
<td>Mark</td>
<td>Records the state of the heap in a pointer variable.</td>
</tr>
<tr>
<td>New</td>
<td>Creates a new dynamic variable and sets a pointer variable to point to it.</td>
</tr>
<tr>
<td>Release</td>
<td>Returns the heap to a given state.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chr</td>
<td>Returns a character of a specified ordinal number.</td>
</tr>
<tr>
<td>Ord</td>
<td>Returns the ordinal number of an ordinal-type value.</td>
</tr>
<tr>
<td>Round</td>
<td>Rounds a type Real value to a type Longint value.</td>
</tr>
<tr>
<td>Trunc</td>
<td>Truncates a type Real value to a type Longint value.</td>
</tr>
</tbody>
</table>
### 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 \textit{Extended} instead of \textit{Real}.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs</td>
<td>Returns the absolute value of the argument.</td>
</tr>
<tr>
<td>ArcTan</td>
<td>Returns the arctangent of the argument.</td>
</tr>
<tr>
<td>Cos</td>
<td>Returns the cosine of the argument.</td>
</tr>
<tr>
<td>Exp</td>
<td>Returns the exponential of the argument.</td>
</tr>
<tr>
<td>Frac</td>
<td>Returns the fractional part of the argument.</td>
</tr>
<tr>
<td>Int</td>
<td>Returns the integer part of the argument.</td>
</tr>
<tr>
<td>Ln</td>
<td>Returns the natural logarithm of the argument.</td>
</tr>
<tr>
<td>Pi</td>
<td>Returns the value of ( \pi ) (3.1415926535897932385).</td>
</tr>
<tr>
<td>Sin</td>
<td>Returns the sine of the argument.</td>
</tr>
<tr>
<td>Sqr</td>
<td>Returns the square of the argument.</td>
</tr>
<tr>
<td>.Sqrt</td>
<td>Returns the square root of the argument.</td>
</tr>
</tbody>
</table>

### Ordinal procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec</td>
<td>Decrements a variable.</td>
</tr>
<tr>
<td>Inc</td>
<td>Increments a variable.</td>
</tr>
</tbody>
</table>

### Ordinal functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd</td>
<td>Tests if the argument is an odd number.</td>
</tr>
<tr>
<td>Pred</td>
<td>Returns the predecessor of the argument.</td>
</tr>
<tr>
<td>Succ</td>
<td>Returns the successor of the argument.</td>
</tr>
</tbody>
</table>

### String procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>Deletes a substring from a string.</td>
</tr>
<tr>
<td>Insert</td>
<td>Inserts a substring into a string.</td>
</tr>
<tr>
<td>Str</td>
<td>Converts a numeric value to its string representation.</td>
</tr>
<tr>
<td>Val</td>
<td>Converts the string value to its numeric representation.</td>
</tr>
</tbody>
</table>
### String functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concat</td>
<td>Concatenates a sequence of strings.</td>
</tr>
<tr>
<td>Copy</td>
<td>Returns a substring of a string.</td>
</tr>
<tr>
<td>Length</td>
<td>Returns the dynamic length of a string.</td>
</tr>
<tr>
<td>Pos</td>
<td>Searches for a substring in a string.</td>
</tr>
</tbody>
</table>

### Pointer and address functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>Returns the address of a specified object.</td>
</tr>
<tr>
<td>CSeg</td>
<td>Returns the current value of the CS register.</td>
</tr>
<tr>
<td>DSeg</td>
<td>Returns the current value of the DS register.</td>
</tr>
<tr>
<td>Ofs</td>
<td>Returns the offset of a specified object.</td>
</tr>
<tr>
<td>Ptr</td>
<td>Converts a segment base and an offset address to a pointer-type value.</td>
</tr>
<tr>
<td>Seg</td>
<td>Returns the segment of a specified object.</td>
</tr>
<tr>
<td>SPtr</td>
<td>Returns the current value of the SP register.</td>
</tr>
<tr>
<td>SSeg</td>
<td>Returns the current value of the SS register.</td>
</tr>
</tbody>
</table>

### Miscellaneous procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FillChar</td>
<td>Fills a specified number of contiguous bytes with a specified value.</td>
</tr>
<tr>
<td>Move</td>
<td>Copies a specified number of contiguous bytes from a source range to a destination range.</td>
</tr>
<tr>
<td>Randomize</td>
<td>Initializes the built-in random generator with a random value.</td>
</tr>
</tbody>
</table>
### Miscellaneous functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi</td>
<td>Returns the high-order byte of the argument.</td>
</tr>
<tr>
<td>Lo</td>
<td>Returns the low-order byte of the argument.</td>
</tr>
<tr>
<td>ParamCount</td>
<td>Returns the number of parameters passed to the program on the command line.</td>
</tr>
<tr>
<td>ParamStr</td>
<td>Returns a specified command-line parameter.</td>
</tr>
<tr>
<td>Random</td>
<td>Returns a random number.</td>
</tr>
<tr>
<td>SizeOf</td>
<td>Returns the number of bytes occupied by the argument.</td>
</tr>
<tr>
<td>Swap</td>
<td>Swaps the high- and low-order bytes of the argument.</td>
</tr>
<tr>
<td>UpCase</td>
<td>Converts a character to uppercase.</td>
</tr>
</tbody>
</table>

### 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.

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*.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Eof</em></td>
<td>Returns the end-of-file status of a file.</td>
</tr>
<tr>
<td><em>FilePos</em></td>
<td>Returns the current file position of a file. Not used for text files.</td>
</tr>
<tr>
<td><em>FileSize</em></td>
<td>Returns the current size of a file. Not used for text files.</td>
</tr>
<tr>
<td><em>IOResult</em></td>
<td>Returns an integer value that is the status of the last I/O function performed.</td>
</tr>
</tbody>
</table>
I/O procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign</td>
<td>Assigns the name of an external file to a file variable.</td>
</tr>
<tr>
<td>ChDir</td>
<td>Changes the current directory.</td>
</tr>
<tr>
<td>Close</td>
<td>Closes an open file.</td>
</tr>
<tr>
<td>Erase</td>
<td>Erases an external file.</td>
</tr>
<tr>
<td>GetDir</td>
<td>Returns the current directory of a specified drive.</td>
</tr>
<tr>
<td>MkDir</td>
<td>Creates a subdirectory.</td>
</tr>
<tr>
<td>Rename</td>
<td>Renames an external file.</td>
</tr>
<tr>
<td>Reset</td>
<td>Opens an existing file.</td>
</tr>
<tr>
<td>Rewrite</td>
<td>Creates and opens a new file.</td>
</tr>
<tr>
<td>RmDir</td>
<td>Removes an empty subdirectory.</td>
</tr>
<tr>
<td>Seek</td>
<td>Moves the current position of a file to a specified component. Not used with text files.</td>
</tr>
<tr>
<td>Truncate</td>
<td>Truncates the file size at the current file position. Not used with text files.</td>
</tr>
</tbody>
</table>

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). \textit{Input} and \textit{Output} are automatically opened before a program begins execution, as if the following statements were executed:

\begin{verbatim}
Assign(Input, '');
Reset(Input);
Assign(Output, '');
Rewrite(Output);
\end{verbatim}

Likewise, \textit{Input} and \textit{Output} are automatically closed after a program finishes executing.

If a program uses the \textit{Crt} standard unit, \textit{Input} and \textit{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, \textit{Input} or \textit{Output} are assumed by default, depending on whether the procedure or function is input- or output-oriented. For instance, \texttt{Read(X)} corresponds to \texttt{Read(Input, X)} and \texttt{Write(X)} corresponds to \texttt{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 \texttt{Assign}, and opened using \texttt{Reset, Rewrite, or Append}. An error message is generated if you pass a file that was opened with \texttt{Reset} to an output-oriented procedure or function. Likewise, it's an error to pass a file that was opened with \texttt{Rewrite} or \texttt{Append} to an input-oriented procedure or function.

### Procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Append}</td>
<td>Opens an existing file for appending.</td>
</tr>
<tr>
<td>\texttt{Flush}</td>
<td>Flushes the buffer of an output file.</td>
</tr>
<tr>
<td>\texttt{Read}</td>
<td>Reads one or more values from a text file into one or more variables.</td>
</tr>
<tr>
<td>\texttt{Readln}</td>
<td>Does what a \texttt{Read} does and then skips to the beginning of the next line in the file.</td>
</tr>
<tr>
<td>\texttt{SetTextBuf}</td>
<td>Assigns an I/O buffer to a text file.</td>
</tr>
<tr>
<td>\texttt{Write}</td>
<td>Writes one or more values to a text file.</td>
</tr>
<tr>
<td>\texttt{Writeln}</td>
<td>Does the same as a \texttt{Write}, and then writes an end-of-line marker to the file.</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eoln</td>
<td>Returns the end-of-line status of a file.</td>
</tr>
<tr>
<td>SeekEof</td>
<td>Returns the end-of-file status of a file.</td>
</tr>
<tr>
<td>SeekEoln</td>
<td>Returns the end-of-line status of a file.</td>
</tr>
</tbody>
</table>

### 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,

```pascal
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

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockRead</td>
<td>Reads one or more records into a variable.</td>
</tr>
<tr>
<td>BlockWrite</td>
<td>Writes one or more records from a variable.</td>
</tr>
</tbody>
</table>

### 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 \texttt{Rewrite} are always opened in Read/Write mode, corresponding to $\text{FileMode} = 2$.

The default $\text{FileMode}$ is 2, which allows both reading and writing. Assigning another value to $\text{FileMode}$ causes all subsequent \texttt{Resets} to use that mode.

The range of valid $\text{FileMode}$ 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.)

\section*{Devices in Turbo Pascal}

Turbo Pascal and the DOS operating system regard external hardware, such as the keyboard, the display, and the printer, as \textit{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.

\subsection*{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

\begin{verbatim}
var
  Lst: Text;
begin
  Assign(Lst, 'LPT1');
  Rewrite(Lst);
  Writeln(Lst, 'Hello World...');
  Close(Lst);
end.
\end{verbatim}
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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Text</td>
<td>Input standard file</td>
</tr>
<tr>
<td>Output</td>
<td>Text</td>
<td>Output standard file</td>
</tr>
<tr>
<td>SaveInt00</td>
<td>Pointer</td>
<td>Saved interrupt $00</td>
</tr>
<tr>
<td>SaveInt02</td>
<td>Pointer</td>
<td>Saved interrupt $02</td>
</tr>
<tr>
<td>SaveInt1B</td>
<td>Pointer</td>
<td>Saved interrupt $1B</td>
</tr>
<tr>
<td>SaveInt21</td>
<td>Pointer</td>
<td>Saved interrupt $21</td>
</tr>
<tr>
<td>SaveInt23</td>
<td>Pointer</td>
<td>Saved interrupt $23</td>
</tr>
<tr>
<td>SaveInt24</td>
<td>Pointer</td>
<td>Saved interrupt $24</td>
</tr>
<tr>
<td>SaveInt34</td>
<td>Pointer</td>
<td>Saved interrupt $34</td>
</tr>
<tr>
<td>SaveInt35</td>
<td>Pointer</td>
<td>Saved interrupt $35</td>
</tr>
<tr>
<td>SaveInt36</td>
<td>Pointer</td>
<td>Saved interrupt $36</td>
</tr>
<tr>
<td>SaveInt37</td>
<td>Pointer</td>
<td>Saved interrupt $37</td>
</tr>
<tr>
<td>SaveInt38</td>
<td>Pointer</td>
<td>Saved interrupt $38</td>
</tr>
<tr>
<td>SaveInt39</td>
<td>Pointer</td>
<td>Saved interrupt $39</td>
</tr>
</tbody>
</table>
### Initialized variables

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

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 \{SN+,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 in applications that deal with data encryption, statistics, and simulations.

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

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

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 SavelntXX 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 \{SN+\} state, the program
halts with a run-time error, and, in the (S1-) state, IOResult returns a nonzero value.

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

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

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.
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:
<table>
<thead>
<tr>
<th>Constants</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCarry</td>
<td>$0001</td>
</tr>
<tr>
<td>FParity</td>
<td>$0004</td>
</tr>
<tr>
<td>FAuxiliary</td>
<td>$0010</td>
</tr>
<tr>
<td>FZero</td>
<td>$0040</td>
</tr>
<tr>
<td>FSign</td>
<td>$0080</td>
</tr>
<tr>
<td>FOverflow</td>
<td>$0800</td>
</tr>
</tbody>
</table>

For instance, if \( R \) is a register record, the tests

\[
\text{R.Flags and FCarry} \neq 0 \\
\text{R.Flags and FZero} = 0
\]

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmClosed</td>
<td>$D7B0</td>
</tr>
<tr>
<td>fmInput</td>
<td>$D7B1</td>
</tr>
<tr>
<td>fmOutput</td>
<td>$D7B2</td>
</tr>
<tr>
<td>fmInOut</td>
<td>$D7B3</td>
</tr>
</tbody>
</table>

**File attribute constants**

These constants test, set, and clear file attribute bits in connection with the \( \text{GetFAttr, SetFAttr, FindFirst, and FindNext} \) procedures:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadOnly</td>
<td>$01</td>
</tr>
<tr>
<td>Hidden</td>
<td>$02</td>
</tr>
<tr>
<td>SysFile</td>
<td>$04</td>
</tr>
<tr>
<td>VolumeID</td>
<td>$08</td>
</tr>
<tr>
<td>Directory</td>
<td>$10</td>
</tr>
<tr>
<td>Archive</td>
<td>$20</td>
</tr>
<tr>
<td>AnyFile</td>
<td>$3F</td>
</tr>
</tbody>
</table>

The constants are additive, that is, the statement

\[
\text{FindFirst}('*.*', \text{ReadOnly} + \text{Directory}, S);
\]

will locate all normal files as well as read-only files and subdirectories in the current directory. The \( \text{AnyFile} \) 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.

```pascal
{ 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.

```pascal
{ Registers type }

type
  Registers = record
    case Integer of
    0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
    1: (AL,AR,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.

```pascal
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.

```pascal
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*:

```pascal
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.

```pascal
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:

<table>
<thead>
<tr>
<th>DOS error code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>File not found</td>
</tr>
<tr>
<td>3</td>
<td>Path not found</td>
</tr>
<tr>
<td>5</td>
<td>Access denied</td>
</tr>
<tr>
<td>6</td>
<td>Invalid handle</td>
</tr>
<tr>
<td>8</td>
<td>Not enough memory</td>
</tr>
<tr>
<td>10</td>
<td>Invalid environment</td>
</tr>
<tr>
<td>11</td>
<td>Invalid format</td>
</tr>
<tr>
<td>18</td>
<td>No more files</td>
</tr>
</tbody>
</table>

Procedures and functions

Date and time procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetDate</code></td>
<td>Returns the current date set in the operating system.</td>
</tr>
<tr>
<td><code>GetFTime</code></td>
<td>Returns the date and time a file was last written.</td>
</tr>
<tr>
<td><code>GetTime</code></td>
<td>Returns the current time set in the operating system.</td>
</tr>
<tr>
<td><code>PackTime</code></td>
<td>Converts a <code>DateTime</code> record into a 4-byte, packed date-and-time character</td>
</tr>
<tr>
<td></td>
<td><code>Longint</code> used by <code>SetFTime</code>. The fields of the <code>DateTime</code> record are not</td>
</tr>
<tr>
<td></td>
<td>range-checked.</td>
</tr>
<tr>
<td><code>SetDate</code></td>
<td>Sets the current date in the operating system.</td>
</tr>
<tr>
<td><code>SetFTime</code></td>
<td>Sets the date and time a file was last written.</td>
</tr>
<tr>
<td><code>SetTime</code></td>
<td>Sets the current time in the operating system.</td>
</tr>
<tr>
<td><code>UnpackTime</code></td>
<td>Converts a 4-byte, packed date-and-time character <code>Longint</code> returned by</td>
</tr>
<tr>
<td></td>
<td><code>GetFTime</code>, <code>FindFirst</code>, or <code>FindNext</code> into an unpacked <code>DateTime</code> record.</td>
</tr>
</tbody>
</table>
**Interrupt support procedures**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetIntVec</code></td>
<td>Returns the address stored in a specified interrupt vector.</td>
</tr>
<tr>
<td><code>Intr</code></td>
<td>Executes a specified software interrupt.</td>
</tr>
<tr>
<td><code>MsDos</code></td>
<td>Executes a DOS function call.</td>
</tr>
<tr>
<td><code>SetIntVec</code></td>
<td>Sets a specified interrupt vector to a specified address.</td>
</tr>
</tbody>
</table>

**Disk status functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DiskFree</code></td>
<td>Returns the number of free bytes of a specified disk drive.</td>
</tr>
<tr>
<td><code>DiskSize</code></td>
<td>Returns the total size in bytes of a specified disk drive.</td>
</tr>
</tbody>
</table>

**File-handling procedures**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FindFirst</code></td>
<td>Searches the specified (or current) directory for the first entry matching the specified file name and set of attributes.</td>
</tr>
<tr>
<td><code>FindNext</code></td>
<td>Returns the next entry that matches the name and attributes specified in a previous call to <code>FindFirst</code>.</td>
</tr>
<tr>
<td><code>FSplit</code></td>
<td>Splits a file name into its three component parts (directory, file name, and extension).</td>
</tr>
<tr>
<td><code>GetFAttr</code></td>
<td>Returns the attributes of a file.</td>
</tr>
<tr>
<td><code>SetFAttr</code></td>
<td>Sets the attributes of a file.</td>
</tr>
</tbody>
</table>

**File-handling functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FExpand</code></td>
<td>Takes a file name and returns a fully qualified file name (drive, directory, and extension).</td>
</tr>
<tr>
<td><code>FSearch</code></td>
<td>Searches for a file in a list of directories.</td>
</tr>
</tbody>
</table>

**Process-handling procedures**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Exec</code></td>
<td>Executes a specified program with a specified command line.</td>
</tr>
<tr>
<td><code>Keep</code></td>
<td><code>Keep</code> (or Terminate Stay Resident) terminates the program and makes it stay in memory.</td>
</tr>
<tr>
<td><code>SwapVectors</code></td>
<td>Swaps all saved interrupt vectors with the current vectors.</td>
</tr>
</tbody>
</table>
### Process-handling functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DosExitCode</td>
<td>Returns the exit code of a subprocess.</td>
</tr>
</tbody>
</table>

### Environment-handling functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnvCount</td>
<td>Returns the number of strings contained in the DOS environment.</td>
</tr>
<tr>
<td>EnvStr</td>
<td>Returns a specified environment string.</td>
</tr>
<tr>
<td>GetEnv</td>
<td>Returns the value of a specified environment variable.</td>
</tr>
</tbody>
</table>

### Miscellaneous procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetCBreak</td>
<td>Returns the state of Ctrl-Break checking in DOS.</td>
</tr>
<tr>
<td>GetVerify</td>
<td>Returns the state of the verify flag in DOS.</td>
</tr>
<tr>
<td>SetCBreak</td>
<td>Sets the state of Ctrl-Break checking in DOS.</td>
</tr>
<tr>
<td>SetVerify</td>
<td>Sets the state of the verify flag in DOS.</td>
</tr>
</tbody>
</table>

### Miscellaneous functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DosVersion</td>
<td>Returns the DOS version number.</td>
</tr>
</tbody>
</table>
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.

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.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Equipment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT.BGI</td>
<td>AT&amp;T 6300 (400 line)</td>
</tr>
<tr>
<td>CGA.BGI</td>
<td>IBM CGA, MCGA</td>
</tr>
<tr>
<td>EGA/EGA.BGI</td>
<td>IBM EGA, VGA</td>
</tr>
<tr>
<td>HERC.BGI</td>
<td>Hercules monochrome</td>
</tr>
<tr>
<td>IBM8514.BGI</td>
<td>IBM 8514</td>
</tr>
<tr>
<td>PC3270.BGI</td>
<td>IBM 3270 PC</td>
</tr>
</tbody>
</table>

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 320x200 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

(0,0)-----------------(319,0)

(0,199)---------------(319,199)

(159,99)
```

---

*Chapter 12, The Graph unit*
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).

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 `{SL}` 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
- Bar3D
- ClearViewPort
- CloseGraph
- DetectGraph
- DrawPoly
- FillPoly
- FloodFill
- GetGraphMode
- ImageSize
- InitGraph
- InstallUserDriver
- InstallUserFont
- PieSlice
- RegisterBGIdriver
- RegisterBGLfont
- SetAllPalette
- SetFillPattern
- SetFillStyle
- SetGraphBufSize
- SetGraphMode
- SetLineStyle
- SetPalette
- SetTextJustify
- SetTextStyle

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.
Getting started

Here's a simple graphics program:

```pascal
program GraphTest;
uses Graph;
var
  GraphDriver: Integer;
  GraphMode: Integer;
  ErrorCode: Integer;
begin
  GraphDriver := Detect;  (* Set flag: do detection *)
  InitGraph(GraphDriver, GraphMode, 'C:\DRIVERS');
  ErrorCode := GraphResult;
  if ErrorCode <> grOk then  (* Error? *)
    begin
      Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
      Writeln('Program aborted...');
      Halt(1);
    end;
  Rectangle(0, 0, GetMaxX, GetMaxY);  (* Draw full screen box *)
  SetTextStyle(DefaultFont, HorizDir, 3);
  OutTextXY(GetMaxX div 2, GetMaxY div 2, 'Borland Graphics Interface (BGI)');
  Readln;
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 autodetection 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:

```pascal
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:

```pascal
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:

```pascal
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:

```pascal
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
  PreGraphExitProc := ExitProc;
  ExitProc := @MyExitProc;
  GraphGetMemPtr := @MyGetMem;
  GraphFreeMemPtr := @MyFreeMem;

  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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect</td>
<td>0</td>
<td>Requests autodetection</td>
</tr>
<tr>
<td>CGA</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MCGA</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>EGA</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>EGA64</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>EGAMono</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>IBM8514</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>HercMono</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>ATT400</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>VGA</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>PC3270</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>CurrentDriver</td>
<td>-128</td>
<td>Passed to <code>GetModeRange</code></td>
</tr>
<tr>
<td>ATT400C0</td>
<td>0</td>
<td>320x200 palette 0: LightGreen, LightRed, Yellow; 1 page</td>
</tr>
<tr>
<td>ATT400C1</td>
<td>1</td>
<td>320x200 palette 1: LightCyan, LightMagenta, White; 1 page</td>
</tr>
<tr>
<td>ATT400C2</td>
<td>2</td>
<td>320x200 palette 2: Green, Red, Brown; 1 page</td>
</tr>
<tr>
<td>ATT400C3</td>
<td>3</td>
<td>320x200 palette 3: Cyan, Magenta, LightGray; 1 page</td>
</tr>
<tr>
<td>ATT400Med</td>
<td>4</td>
<td>640x200 1 page</td>
</tr>
<tr>
<td>ATT400Hi</td>
<td>5</td>
<td>640x400 1 page</td>
</tr>
<tr>
<td>CGAC0</td>
<td>0</td>
<td>320x200 palette 0: LightGreen, LightRed, Yellow; 1 page</td>
</tr>
<tr>
<td>CGC1</td>
<td>1</td>
<td>320x200 palette 1: LightCyan, LightMagenta, White; 1 page</td>
</tr>
<tr>
<td>CGC2</td>
<td>2</td>
<td>320x200 palette 2: Green, Red, Brown; 1 page</td>
</tr>
<tr>
<td>CGC3</td>
<td>3</td>
<td>320x200 palette 3: Cyan, Magenta, LightGray; 1 page</td>
</tr>
<tr>
<td>CGAHi</td>
<td>4</td>
<td>640x200 1 page</td>
</tr>
<tr>
<td>EGALo</td>
<td>0</td>
<td>640x200 16 color 4 page</td>
</tr>
<tr>
<td>EGAHi</td>
<td>1</td>
<td>640x350 16 color 2 page</td>
</tr>
<tr>
<td>EGA64Lo</td>
<td>0</td>
<td>640x200 16 color 1 page</td>
</tr>
<tr>
<td>EGA64Hi</td>
<td>1</td>
<td>640x350 4 color 1 page</td>
</tr>
<tr>
<td>EGAMonoHi</td>
<td>3</td>
<td>640x350 64K on card, 1 page; 256K on card, 2 page</td>
</tr>
<tr>
<td>HercMonoHi</td>
<td>0</td>
<td>720x348 2 page</td>
</tr>
<tr>
<td>IBM8514Lo</td>
<td>0</td>
<td>640x480 256 colors</td>
</tr>
<tr>
<td>IBM8514Hi</td>
<td>1</td>
<td>1024x768 256 colors</td>
</tr>
<tr>
<td>MCGAC0</td>
<td>0</td>
<td>320x200 palette 0: LightGreen, LightRed, Yellow; 1 page</td>
</tr>
<tr>
<td>MCGAC1</td>
<td>1</td>
<td>320x200 palette 1: LightCyan, LightMagenta, White; 1 page</td>
</tr>
<tr>
<td>MCGAC2</td>
<td>2</td>
<td>320x200 palette 2: Green, Red, Brown; 1 page</td>
</tr>
<tr>
<td>MCGAC3</td>
<td>3</td>
<td>320x200 palette 3: Cyan, Magenta, LightGray; 1 page</td>
</tr>
<tr>
<td>MCGAMed</td>
<td>4</td>
<td>640x200 1 page</td>
</tr>
</tbody>
</table>
Table 12.1: Graph unit driver and mode constants (continued)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCGAHi</td>
<td>5</td>
<td>640x480 1 page</td>
</tr>
<tr>
<td>PC3270Hi</td>
<td>0</td>
<td>720x350 1 page</td>
</tr>
<tr>
<td>VGAlo</td>
<td>0</td>
<td>640x200 16 color 4 page</td>
</tr>
<tr>
<td>VGAMed</td>
<td>1</td>
<td>640x350 16 color 2 page</td>
</tr>
<tr>
<td>VGAHi</td>
<td>2</td>
<td>640x480 16 color 1 page</td>
</tr>
</tbody>
</table>

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

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

SetPalette and SetAllPalette

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>2</td>
</tr>
<tr>
<td>Cyan</td>
<td>3</td>
</tr>
<tr>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>Magenta</td>
<td>5</td>
</tr>
<tr>
<td>Brown</td>
<td>6</td>
</tr>
<tr>
<td>LightGray</td>
<td>7</td>
</tr>
<tr>
<td>DarkGray</td>
<td>8</td>
</tr>
<tr>
<td>LightBlue</td>
<td>9</td>
</tr>
<tr>
<td>LightGreen</td>
<td>10</td>
</tr>
<tr>
<td>LightCyan</td>
<td>11</td>
</tr>
<tr>
<td>LightRed</td>
<td>12</td>
</tr>
<tr>
<td>LightMagenta</td>
<td>13</td>
</tr>
<tr>
<td>Yellow</td>
<td>14</td>
</tr>
<tr>
<td>White</td>
<td>15</td>
</tr>
</tbody>
</table>
These color constants can be used with `SetRGBPalette` to select the standard EGA colors on an IBM 8514 graphics adapter:

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

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolidLn</td>
<td>0</td>
</tr>
<tr>
<td>DottedLn</td>
<td>1</td>
</tr>
<tr>
<td>CenterLn</td>
<td>2</td>
</tr>
<tr>
<td>DashedLn</td>
<td>3</td>
</tr>
<tr>
<td>UserBitLn</td>
<td>4 (user-defined line style)</td>
</tr>
<tr>
<td>NormWidth</td>
<td>1</td>
</tr>
<tr>
<td>ThickWidth</td>
<td>3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefaultFont</td>
<td>0 (8x8 bit mapped font)</td>
</tr>
<tr>
<td>TriplexFont</td>
<td>1 (&quot;stroked&quot; fonts)</td>
</tr>
<tr>
<td>SmallFont</td>
<td>2</td>
</tr>
<tr>
<td>SansSerifFont</td>
<td>3</td>
</tr>
<tr>
<td>GothicFont</td>
<td>4</td>
</tr>
<tr>
<td>HorizDir</td>
<td>0 (left to right)</td>
</tr>
<tr>
<td>VertDir</td>
<td>1 (bottom to top)</td>
</tr>
<tr>
<td>UserCharSize</td>
<td>0 (user-defined Char size)</td>
</tr>
</tbody>
</table>
Justification constants
These constants control horizontal and vertical justification for 
SetTextJustify:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeftText</td>
<td>0</td>
</tr>
<tr>
<td>CenterText</td>
<td>1</td>
</tr>
<tr>
<td>RightText</td>
<td>2</td>
</tr>
<tr>
<td>BottomText</td>
<td>0</td>
</tr>
<tr>
<td>CenterText</td>
<td>1 (already defined)</td>
</tr>
<tr>
<td>TopText</td>
<td>2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClipOn</td>
<td>True</td>
</tr>
<tr>
<td>ClipOff</td>
<td>False</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopOn</td>
<td>True</td>
</tr>
<tr>
<td>TopOff</td>
<td>False</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmptyFill</td>
<td>0 (Fills area in background color)</td>
</tr>
<tr>
<td>SolidFill</td>
<td>1 (Fills area in solid fill color)</td>
</tr>
<tr>
<td>LineFill</td>
<td>2 (- - - fill)</td>
</tr>
<tr>
<td>LtSlashFill</td>
<td>3 (// / / fill)</td>
</tr>
<tr>
<td>SlashFill</td>
<td>4 (/// / / fill with thick lines)</td>
</tr>
<tr>
<td>BkSlashFill</td>
<td>5 (\ \ \ fill with thick lines)</td>
</tr>
<tr>
<td>LtBkSlashFill</td>
<td>6 (\ \ \ fill)</td>
</tr>
<tr>
<td>HatchFill</td>
<td>7 (Light hatch fill)</td>
</tr>
<tr>
<td>XHatchFill</td>
<td>8 (Heavy cross hatch fill)</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyPut</td>
<td>0 (mov)</td>
</tr>
<tr>
<td>XORPut</td>
<td>1 (xor)</td>
</tr>
</tbody>
</table>

These BitBlt operators are used by PutImage only:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrPut</td>
<td>2 (or)</td>
</tr>
<tr>
<td>AndPut</td>
<td>3 (and)</td>
</tr>
<tr>
<td>NotPut</td>
<td>4 (not)</td>
</tr>
</tbody>
</table>

Palette constant

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxColors</td>
<td>15</td>
</tr>
</tbody>
</table>

Types

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

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

This record is used by GetLineSettings:

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

This record is used by GetTextSettings:

```pascal
type
  TextSettingsType = record
    Font: Word;
  end;
```
Direction: Word;
CharSize: Word;
Horiz: Word;
Vert: Word;
end;

This record is used by by GetFillSettings:

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

This record is used by GetFillPattern and SetFillPattern:

```pascal
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):

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

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

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

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

```pascal
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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GraphGetMemPtr</code></td>
<td>Pointer (steal heap allocation)</td>
</tr>
<tr>
<td><code>GraphFreeMemPtr</code></td>
<td>Pointer (steal heap deallocation)</td>
</tr>
</tbody>
</table>

Table 12.3: Graph unit procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc</td>
<td>Draws a circular arc from start angle to end angle, using ((x,y)) as the center point.</td>
</tr>
<tr>
<td>Bar</td>
<td>Draws a bar using the current fill style and color.</td>
</tr>
<tr>
<td>Bar3D</td>
<td>Draws a 3-D bar using the current fill style and color.</td>
</tr>
<tr>
<td>Circle</td>
<td>Draws a circle using ((x,y)) as the center point.</td>
</tr>
<tr>
<td>ClearDevice</td>
<td>Clears the currently selected output device and homes the current pointer.</td>
</tr>
<tr>
<td>ClearViewport</td>
<td>Clears the current viewport.</td>
</tr>
<tr>
<td>CloseGraph</td>
<td>Shuts down the graphics system.</td>
</tr>
<tr>
<td>DetectGraph</td>
<td>Checks the hardware and determines which graphics driver and mode to use.</td>
</tr>
<tr>
<td>DrawPoly</td>
<td>Draws the outline of a polygon using the current line style and color.</td>
</tr>
<tr>
<td>Ellipse</td>
<td>Draws an elliptical arc from start angle to end angle, using ((x,y)) as the center point.</td>
</tr>
<tr>
<td>FillEllipse</td>
<td>Draws a filled ellipse using ((x,y)) as a center point and (XRadius) and (YRadius) as the horizontal and vertical axes.</td>
</tr>
<tr>
<td>FillPoly</td>
<td>Fills a polygon, using the scan converter.</td>
</tr>
<tr>
<td>FloodFill</td>
<td>Fills a bounded region using the current fill pattern and fill color.</td>
</tr>
<tr>
<td>GetArcCoords</td>
<td>Allows the user to inquire about the coordinates of the last Arc command.</td>
</tr>
<tr>
<td>GetAspectRatio</td>
<td>Returns the effective resolution of the graphics screen from which the aspect ratio ((Xasp:Yasp)) can be computed.</td>
</tr>
<tr>
<td>GetFillPattern</td>
<td>Returns the last fill pattern set by a call to SetFillPattern.</td>
</tr>
<tr>
<td>GetFillSettings</td>
<td>Allows the user to inquire about the current fill pattern and color as set by SetFillStyle or SetFillPattern.</td>
</tr>
<tr>
<td>GetImage</td>
<td>Saves a bit image of the specified region into a buffer.</td>
</tr>
<tr>
<td>GetLineSettings</td>
<td>Returns the current line style, line pattern, and line thickness as set by SetLineStyle.</td>
</tr>
<tr>
<td>GetModeRange</td>
<td>Returns the lowest and highest valid graphics mode for a given driver.</td>
</tr>
<tr>
<td>GetPalette</td>
<td>Returns the current palette and its size.</td>
</tr>
</tbody>
</table>

Chapter 12, The Graph unit
Table 12.3: Graph unit procedures (continued)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetTextSettings</td>
<td>Returns the current text font, direction, size, and justification as set by SetTextStyle and SetTextJustify.</td>
</tr>
<tr>
<td>GetViewSettings</td>
<td>Allows the user to inquire about the current viewport and clipping parameters.</td>
</tr>
<tr>
<td>GraphDefaults</td>
<td>Homes the current pointer (CP) and resets the graphics system.</td>
</tr>
<tr>
<td>InitGraph</td>
<td>Initializes the graphics system and puts the hardware into graphics mode.</td>
</tr>
<tr>
<td>Line</td>
<td>Draws a line from the ((x_1, y_1)) to ((x_2, y_2)).</td>
</tr>
<tr>
<td>LineRel</td>
<td>Draws a line to a point that is a relative distance from the current pointer (CP).</td>
</tr>
<tr>
<td>LineTo</td>
<td>Draws a line from the current pointer to ((x,y)).</td>
</tr>
<tr>
<td>MoveRel</td>
<td>Moves the current pointer (CP) a relative distance from its current position.</td>
</tr>
<tr>
<td>MoveTo</td>
<td>Moves the current graphics pointer (CP) to ((x,y)).</td>
</tr>
<tr>
<td>OutText</td>
<td>Sends a string to the output device at the current pointer.</td>
</tr>
<tr>
<td>OutTextXY</td>
<td>Sends a string to the output device.</td>
</tr>
<tr>
<td>PieSlice</td>
<td>Draws and fills a pie slice, using ((x,y)) as the center point and drawing from start angle to end angle.</td>
</tr>
<tr>
<td>PutImage</td>
<td>Puts a bit image onto the screen.</td>
</tr>
<tr>
<td>PutPixel</td>
<td>Plots a pixel at ((x,y)).</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Draws a rectangle using the current line style and color.</td>
</tr>
<tr>
<td>RestoreCrtMode</td>
<td>Restores the original screen mode before graphics is initialized.</td>
</tr>
<tr>
<td>Sector</td>
<td>Draws and fills an elliptical sector.</td>
</tr>
<tr>
<td>SetActivePage</td>
<td>Set the active page for graphics output.</td>
</tr>
<tr>
<td>SetAllPalette</td>
<td>Changes all palette colors as specified.</td>
</tr>
<tr>
<td>SetAspectRatio</td>
<td>Changes the default aspect ratio.</td>
</tr>
<tr>
<td>SetBkColor</td>
<td>Sets the current background color using the palette.</td>
</tr>
<tr>
<td>SetColor</td>
<td>Sets the current drawing color using the palette.</td>
</tr>
<tr>
<td>SetFillPattern</td>
<td>Selects a user-defined fill pattern.</td>
</tr>
<tr>
<td>SetFillStyle</td>
<td>Sets the fill pattern and color.</td>
</tr>
<tr>
<td>SetGraphBufSize</td>
<td>Lets you change the size of the buffer used for scan and flood fills.</td>
</tr>
<tr>
<td>SetGraphMode</td>
<td>Sets the system to graphics mode and clears the screen.</td>
</tr>
<tr>
<td>SetLineStyle</td>
<td>Sets the current line width and style.</td>
</tr>
<tr>
<td>SetPalette</td>
<td>Changes one palette color as specified by ColorNum and Color.</td>
</tr>
<tr>
<td>SetRGBPalette</td>
<td>Lets you modify palette entries for the IBM 8514 and the VGA drivers.</td>
</tr>
<tr>
<td>SetTextJustify</td>
<td>Sets text justification values used by OutText and OutTextXY.</td>
</tr>
<tr>
<td>SetTextStyle</td>
<td>Sets the current text font, style, and character magnification factor.</td>
</tr>
<tr>
<td>SetUserCharSize</td>
<td>Lets you change the character width and height for stroked fonts.</td>
</tr>
</tbody>
</table>
Table 12.3: Graph unit procedures (continued)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetViewport</td>
<td>Sets the current output viewport or window for graphics output.</td>
</tr>
<tr>
<td>SetVisualPage</td>
<td>Sets the visual graphics page number.</td>
</tr>
<tr>
<td>SetWriteMode</td>
<td>Sets the writing mode (copy or xor) for lines drawn by DrawPoly, Line, LineRel, LineTo, and Rectangle.</td>
</tr>
</tbody>
</table>

Table 12.4: Graph unit functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBkColor</td>
<td>Returns the current background color.</td>
</tr>
<tr>
<td>GetColor</td>
<td>Returns the current drawing color.</td>
</tr>
<tr>
<td>GetDefaultPalette</td>
<td>Returns the default hardware palette in a record of PaletteType.</td>
</tr>
<tr>
<td>GetDriverName</td>
<td>Returns a string containing the name of the current driver.</td>
</tr>
<tr>
<td>GetGraphMode</td>
<td>Returns the current graphics mode.</td>
</tr>
<tr>
<td>GetMaxColor</td>
<td>Returns the highest color that can be passed to SetColor.</td>
</tr>
<tr>
<td>GetMaxMode</td>
<td>Returns the maximum mode number for the currently loaded driver.</td>
</tr>
<tr>
<td>GetMaxX</td>
<td>Returns the rightmost column (x resolution) of the current graphics driver and mode.</td>
</tr>
<tr>
<td>GetMaxY</td>
<td>Returns the bottommost row (y resolution) of the current graphics driver and mode.</td>
</tr>
<tr>
<td>GetModeName</td>
<td>Returns a string containing the name of the specified graphics mode.</td>
</tr>
<tr>
<td>GetPaletteSize</td>
<td>Returns the size of the palette color lookup table.</td>
</tr>
<tr>
<td>GetPixel</td>
<td>Gets the pixel value at (x,y).</td>
</tr>
<tr>
<td>GetX</td>
<td>Returns the x-coordinate of the current position (CP).</td>
</tr>
<tr>
<td>GetY</td>
<td>Returns the y-coordinate of the current position (CP).</td>
</tr>
<tr>
<td>GraphErrorMsg</td>
<td>Returns an error message string for the specified ErrorCode.</td>
</tr>
<tr>
<td>GraphResult</td>
<td>Returns an error code for the last graphics operation.</td>
</tr>
<tr>
<td>ImageSize</td>
<td>Returns the number of bytes required to store a rectangular region of the screen.</td>
</tr>
<tr>
<td>InstallUserDriver</td>
<td>Installs a vendor-added device driver to the BGI device driver table.</td>
</tr>
<tr>
<td>InstallUserFont</td>
<td>Installs a new font file that is not built into the BGI system.</td>
</tr>
<tr>
<td>RegisterBGIdriver</td>
<td>Registers a valid BGI driver with the graphics system.</td>
</tr>
<tr>
<td>RegisterBGIfont</td>
<td>Registers a valid BGI font with the graphics system.</td>
</tr>
<tr>
<td>TextHeight</td>
<td>Returns the height of a string in pixels.</td>
</tr>
<tr>
<td>TextWidth</td>
<td>Returns the width of a string in pixels.</td>
</tr>
</tbody>
</table>

For a detailed description of each procedure or function, refer to Chapter 1, "Run-time library," in the Library Reference.
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.
Failing to observe the FAR call requirement in an overlaid program will cause unpredictable and possibly catastrophic results when the program is executed.

The {SO 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:

```pascal
program Editor;
{$F+}  { Force FAR calls for all procedures & functions } uses Overlay, Crt, Dos, EdInOut, EdFormat, EdPrint, EdFind, EdMain;
{SO EdInOut}
{SO EdFormat}
{SO EdPrint}
{SO EdFind}
{SO EdMain}
```

The compiler reports an error if you attempt to overlay a unit that wasn't compiled in the {SO+} 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.
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/reprieve 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/reprieve 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.
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.

```
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.

```
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.

```
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.

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 Ovrlnit; at this point, OvrReadBuf will contain the address of the default disk read function.

If you also call OvrlnitEMS, 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 OvrlnitEMS.

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.

```pascal
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;
  OvrReadBuf := MyOvrRead;
  UsingEMS := False;
  OvrInitEMS;
  if (OvrResult = OvrOK) then
  begin
    SaveOvrRead := OvrReadBuf;
    OvrReadBuf := MyOvrRead;
    UsingEMS := True;
  end;
  ...
end.
```

Turbo Pascal Programmer's Guide
Result codes

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ovrOk</td>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>ovrError</td>
<td>-1</td>
<td>Overlay manager error</td>
</tr>
<tr>
<td>ovrNotFound</td>
<td>-2</td>
<td>Overlay file not found</td>
</tr>
<tr>
<td>ovrNoMemory</td>
<td>-3</td>
<td>Not enough memory for overlay buffer</td>
</tr>
<tr>
<td>ovrIOError</td>
<td>-4</td>
<td>Overlay file I/O error</td>
</tr>
<tr>
<td>ovrNoEMSDriver</td>
<td>-5</td>
<td>EMS driver not installed</td>
</tr>
<tr>
<td>ovrNoEMSMemory</td>
<td>-6</td>
<td>Not enough EMS memory</td>
</tr>
</tbody>
</table>

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

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

procedure OvrInit(FileName: string);

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**

```plaintext
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:

```plaintext
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**

```plaintext
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.
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:

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

and if UnitB contains the statement

```pascal
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.

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
Chapter 13, The Overlay unit

Initializing the overlay manager

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 Force Far Calls check box in the Options | Compiler 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.

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:

```pascal
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:

```pascal
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
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:

```pascal
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:

```pascal
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');
    ovrNo EMSMemory: 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:

```pascal
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:

```pascal
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.
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 Standalone option is checked in Options | Debugger. With the command-line version of the compiler, make sure not to specify a /N 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));
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 $2.9 \times 10^{-39}$ to $1.7 \times 10^{38}$ 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 $3.4 \times 10^{-4951}$ to $1.1 \times 10^{4932}$ 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 8087/80287 check box in the Options | Compiler dialog box. The default state is `$N-$`, and in this state, the compiler uses the 6-byte floating-point library, allowing you to operate only on variables of type Real. In the `$N+$` state, the compiler generates code for the 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 Emulation check box in the Options | Compiler 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 $1.5 \times 10^{-45}$ to $3.4 \times 10^{38}$ with 7 to 8 significant digits.

- The Double type occupies 8 bytes of memory, providing a range of $5.0 \times 10^{-324}$ to $1.7 \times 10^{308}$ 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 $3.4 \times 10^{-4932}$ to $1.1 \times 10^{4932}$ 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 $-9.2 \times 10^{18}$ to $9.2 \times 10^{18}$. 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:

\[
X := \frac{B + \sqrt{B^2 - A^2}}{A};
\]

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:

\[
\text{sum := } T;
\]

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 \(\text{sum}\). 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:

\[
\begin{align*}
  &X := 1 / 3; \\
  &Y := 1 / 3; \\
  &WriteLn(X = Y); \\
\end{align*}
\]

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

\[
\begin{align*}
  &X := 1 / 3; \\
  &WriteLn(X = 1 / 3); \\
\end{align*}
\]

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:

```pascal
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

```pascal
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 `{SN+}` 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 \textit{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 \textit{$\textbf{S\textbf{N}+}$} state.

The fact that a unit uses the 8087 is determined by whether it contains 8087 instructions—not by the state of the \textit{$\textbf{S\textbf{N}}$} 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 \textit{$\textbf{S\textbf{N}+}$} state.

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

### Detecting the 8087

The Turbo Pascal 8087 run-time library built into your program (compiled with \textit{$\textbf{S\textbf{N}+}$}) 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 \textit{$\textbf{S\textbf{E}-}$} 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 \textit{87} environment variable.

You set the \textit{87} environment variable at the DOS prompt with the \textit{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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No coprocessor detected</td>
</tr>
<tr>
<td>1</td>
<td>8087 detected</td>
</tr>
<tr>
<td>2</td>
<td>80287 detected</td>
</tr>
<tr>
<td>3</td>
<td>80387 detected</td>
</tr>
</tbody>
</table>

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 \($L \text{filename}\) 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 \texttt{external} procedures, make sure to enable emulation when you assemble the \texttt{.ASM} files into \texttt{.OBJ} files. Otherwise, the 8087 instructions cannot be emulated on machines without an 8087. Use Turbo Assembler’s \texttt{/E} command-line switch to enable emulation.
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:

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

All other characters will appear onscreen when written.
Line input

When reading from \textit{Input} or from a text file that has been assigned to \textit{Crt}, text is input one line at a time. The line is stored in the text file's internal buffer, and when variables are read, this buffer is used as the input source. Whenever the buffer has been emptied, a new line is input.

When entering lines, the following editing keys are available:

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

\textit{Ctrl-Z} will only generate an end-of-file marker if the \textit{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 \textit{KeyPressed} and \textit{ReadKey} functions.

 Constants, types, and variables

Each of the constants, types, and variables defined by the \textit{Crt} unit are briefly discussed in this section.
### Constants

#### Crt mode constants

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

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

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>2</td>
</tr>
<tr>
<td>Cyan</td>
<td>3</td>
</tr>
<tr>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>Magenta</td>
<td>5</td>
</tr>
<tr>
<td>Brown</td>
<td>6</td>
</tr>
<tr>
<td>LightGray</td>
<td>7</td>
</tr>
<tr>
<td>DarkGray</td>
<td>8</td>
</tr>
<tr>
<td>LightBlue</td>
<td>9</td>
</tr>
<tr>
<td>LightGreen</td>
<td>10</td>
</tr>
<tr>
<td>LightCyan</td>
<td>11</td>
</tr>
<tr>
<td>LightRed</td>
<td>12</td>
</tr>
<tr>
<td>LightMagenta</td>
<td>13</td>
</tr>
<tr>
<td>Yellow</td>
<td>14</td>
</tr>
<tr>
<td>White</td>
<td>15</td>
</tr>
<tr>
<td>Blink</td>
<td>128</td>
</tr>
</tbody>
</table>
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*:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckBreak</td>
<td>Boolean</td>
</tr>
<tr>
<td>CheckEOF</td>
<td>Boolean</td>
</tr>
<tr>
<td>CheckSnow</td>
<td>Boolean</td>
</tr>
<tr>
<td>DirectVideo</td>
<td>Boolean</td>
</tr>
<tr>
<td>LastMode</td>
<td>Word</td>
</tr>
<tr>
<td>TextAttr</td>
<td>Byte</td>
</tr>
<tr>
<td>WindMin</td>
<td>Word</td>
</tr>
<tr>
<td>WindMax</td>
<td>Word</td>
</tr>
</tbody>
</table>

**CheckBreak**  Enables and disables checks for *Ctrl-Break*.

```pascal
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:

```pascal
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.

```pascal
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.

```pascal
var DirectVideo: Boolean;
```

When DirectVideo is True, Writes and Writeln 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.

```pascal
var LastMode: Word;
```

**TextAttr** Stores the currently selected text attributes.

```pascal
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:

```
bit → 7 6 5 4 3 2 1 0
B b b b f f f f
```
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:

\[ \text{TextAttr} := \text{Yellow} + \text{Blue} \times 16 + \text{Blink}; \]

WindMin and WindMax

Store the screen coordinates of the current window.

\[ \text{var WindMin, WindMax: Word;} \]

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, \( \text{Lo(WindMin)} \) produces the x-coordinate of the left edge, and \( \text{Hi(WindMax)} \) 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyPressed</td>
<td>Returns True if a key has been pressed on the keyboard, and False otherwise.</td>
</tr>
<tr>
<td>ReadKey</td>
<td>Reads a character from the keyboard.</td>
</tr>
<tr>
<td>WhereX</td>
<td>Returns the x-coordinate of the current cursor position, relative to the current window. ( X ) is the horizontal position.</td>
</tr>
<tr>
<td>WhereY</td>
<td>Returns the y-coordinate of the current cursor position, relative to the current window. ( Y ) is the vertical position.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssignCrt</td>
<td>Associates a text file with the CRT.</td>
</tr>
<tr>
<td>ClrEol</td>
<td>Clears all characters from the cursor position to the end of the line without moving the cursor.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ClrScr</td>
<td>Clears the screen and places the cursor in the upper left-hand corner.</td>
</tr>
<tr>
<td>Delay</td>
<td>Delays a specified number of milliseconds.</td>
</tr>
<tr>
<td>DelLine</td>
<td>Deletes the line containing the cursor and moves all lines below that line one line up. The bottom line is cleared.</td>
</tr>
<tr>
<td>GotoXY</td>
<td>Positions the cursor. X is the horizontal position. Y is the vertical position.</td>
</tr>
<tr>
<td>HighVideo</td>
<td>Selects high-intensity characters.</td>
</tr>
<tr>
<td>InsLine</td>
<td>Inserts an empty line at the cursor position.</td>
</tr>
<tr>
<td>LowVideo</td>
<td>Selects low-intensity characters.</td>
</tr>
<tr>
<td>NormVideo</td>
<td>Selects normal characters.</td>
</tr>
<tr>
<td>NoSound</td>
<td>Turns off the internal speaker.</td>
</tr>
<tr>
<td>Sound</td>
<td>Starts the internal speaker.</td>
</tr>
<tr>
<td>TextBackground</td>
<td>Selects the background color.</td>
</tr>
<tr>
<td>TextColor</td>
<td>Selects the foreground character color.</td>
</tr>
<tr>
<td>TextMode</td>
<td>Selects a specific text mode.</td>
</tr>
<tr>
<td>Window</td>
<td>Defines a text window onscreen.</td>
</tr>
</tbody>
</table>
Inside Turbo Pascal
Chapter 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.
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.
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:

![Heap Diagram](image)

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.

(corresponding to $10000$ minus $000F$), since every variable must be completely contained in a single segment.)
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 \textit{New}(Ptr3) had been executed now, it would again occupy the same memory area. On the other hand, executing \textit{Dispose}(Ptr4) enlarges the free block, since \textit{Ptr3} and \textit{Ptr4} were neighboring blocks (see Figure 16.5).

Figure 16.5
Enlarging the free block

Finally, executing \textit{Dispose}(Ptr5) first creates an even bigger free block, and then lowers \textit{HeapPtr}. This, in effect, releases the free block, since the last valid pointer is now \textit{Ptr2} (see Figure 16.6).
The heap is now in the same state as it would be after executing \textit{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 \textit{Dispose} and \textit{FreeMem} operations are kept on a \textit{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 \textit{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 \textit{Mark} and \textit{Release} with calls to \textit{Dispose} and \textit{FreeMem}, you must ensure that no such free blocks exist.

The \textit{FreeList} variable in the \textit{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 \textit{HeapPtr}). If there are no free blocks on the free list, \textit{FreeList} will be equal to \textit{HeapPtr}.

The format of the first 8 bytes of a free block are given by the \textit{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 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:

```pascal
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:

```pascal
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:

```pascal
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

\[ \pm \text{significand} \times 2^{\text{exponent}} \]

where the significand has a single bit to the left of the binary decimal point (that is, \(0 \leq \text{significand} < 2\)).

In the figures that follow, \(\text{msb}\) means most significant bit, and \(\text{lsb}\) 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:

<table>
<thead>
<tr>
<th>width in bits</th>
<th>1</th>
<th>39</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>f</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

\(\text{msb} \quad \text{lsb} \quad \text{msb} \quad \text{lsb}\)

The value \(v\) of the number is determined by

\[ \begin{align*}
\text{if } 0 < e \leq 255, & \quad \text{then } v = (-1)^s \times 2^{(e-128)} \times (1.f) . \\
\text{if } e = 0, & \quad \text{then } v = 0 .
\end{align*} \]

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:

<table>
<thead>
<tr>
<th>width in bits</th>
<th>1</th>
<th>8</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

\(\text{msb} \quad \text{lsb} \quad \text{msb} \quad \text{lsb}\)
The value \( v \) of the number is determined by

\[
\begin{align*}
\text{if } 0 &< e < 255, \quad \text{then } v = (-1)^{s} \times 2^{(e-127)} \times (1.f). \\
\text{if } e = 0 \quad \text{and } f <> 0, \quad \text{then } v = (-1)^{s} \times 2^{(-126)} \times (0.f). \\
\text{if } e = 0 \quad \text{and } f = 0, \quad \text{then } v = (-1)^{s} \times 0. \\
\text{if } e = 255 \quad \text{and } f = 0, \quad \text{then } v = (-1)^{s} \times \text{Inf}. \\
\text{if } e = 255 \quad \text{and } f <> 0, \quad \text{then } v \text{ is a NaN.}
\end{align*}
\]

The Double type
An 8-byte (64-bit) Double number is divided into three fields:

\[
\begin{array}{cccc}
\text{width in bits} & 1 & 11 & 52 \\
\text{s} & e & f \\
\text{msb} & \text{lsb} & \text{msb} & \text{lsb}
\end{array}
\]

The value \( v \) of the number is determined by

\[
\begin{align*}
\text{if } 0 &< e < 2047, \quad \text{then } v = (-1)^{s} \times 2^{(e-1023)} \times (1.f). \\
\text{if } e = 2047 \quad \text{and } f = 0, \quad \text{then } v = (-1)^{s} \times \text{Inf.} \\
\text{if } e = 2047 \quad \text{and } f <> 0, \quad \text{then } v \text{ is a NaN.}
\end{align*}
\]

The Extended type
A 10-byte (80-bit) Extended number is divided into four fields:

\[
\begin{array}{cccc}
\text{width in bits} & 1 & 15 & 1 & 63 \\
\text{s} & e & i & f \\
\text{msb} & \text{lsb} & \text{msb} & \text{lsb}
\end{array}
\]

The value \( v \) of the number is determined by

\[
\begin{align*}
\text{if } 0 &\leq e < 32767, \quad \text{then } v = (-1)^{s} \times 2^{(e-16383)} \times (i.f). \\
\text{if } e = 32767 \quad \text{and } f = 0, \quad \text{then } v = (-1)^{s} \times \text{Inf.} \\
\text{if } e = 32767 \quad \text{and } f <> 0, \quad \text{then } v \text{ is a NaN.}
\end{align*}
\]

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

\[
\begin{array}{cccc}
\text{width in bits} & 1 & 63 \\
\text{s} & d \\
\text{msb} & \text{lsb}
\end{array}
\]

*Turbo Pascal Programmer's Guide*
The value $v$ of the number is determined by

$$\text{if } s = 1 \text{ and } d = 0, \text{ then } v \text{ is a NaN}$$

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

$$\text{ByteSize} = (\text{Max div 8}) - (\text{Min div 8}) + 1$$

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

$$\text{ByteNumber} = (E \text{ div 8}) - (\text{Min div 8})$$

and the bit number within that byte is

$$\text{BitNumber} = E \text{ mod } 8$$

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.
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:

```pascal
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:

```pascal
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:

```plaintext
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

\[
\text{Mem}[(0040:0049)] := 7;
\]
\[
\text{Data} := \text{MemW}[\text{Seg}(V) : \text{Ofs}(V)];
\]
\[
\text{MemLong} := \text{MemL}[64:3*4];
\]

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 \( \text{Data} \). The third statement moves the Longint value stored at $0040:000C$ into the variable \( \text{MemLong} \).
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:

```plaintext
type
    LocationPtr = ^Location;
    Location = object
    X, Y: Integer;
```
procedure Init(PX, PY: Integer);
functionGetX: Integer;
functionGetY: Integer;
end;

PointPtr = 'Point;
Point = object(Location)
  Color: Integer;
  constructorInit(PX, PY, PColor: Integer);
  destructorDone; virtual;
  procedure Show; virtual;
  procedureHide; virtual;
  procedureMoveTo(PX, PY: Integer); virtual;
end;

CirclePtr = 'Circle;
Circle = object(Point)
  Radius: Integer;
  constructorInit(PX, PY, PColor, PRadius: Integer);
  procedureShow; virtual;
  procedureHide; virtual;
  procedureFill; virtual;
end;

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

<table>
<thead>
<tr>
<th>Location</th>
<th>Point</th>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Color</td>
</tr>
<tr>
<td>Color</td>
<td>Color</td>
<td></td>
</tr>
<tr>
<td>VMT</td>
<td>VMT</td>
<td>Radius</td>
</tr>
</tbody>
</table>

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.
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,

```plaintext
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:

```plaintext
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`:

```plaintext
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^MOVE_TO(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 $SF$ 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.

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).

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

\[
\text{New}(P, \text{Construct})
\]

and

\[
\text{Dispose}(P, \text{Destruct})
\]

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

\[
\text{New}(P) ; \\
\text{P}^\backslash \text{Construct} ;
\]

And for \text{Dispose}, the effect of the extended syntax is the same as executing

\[
\text{P}^\backslash \text{Destruct} ; \\
\text{Dispose}(P) ;
\]

Without the extended syntax, occurrences of such “pairs” of a call to \text{New} followed by a constructor call, and a destructor call followed by a call to \text{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 \text{New} and \text{Dispose} syntax:

\[
\text{var} \\
\text{SP: StrFieldPtr;} \\
\text{ZP: ZipFieldPtr;} \\
\text{begin} \\
\text{New}(\text{SP, Init}(1, 1, 25, 'Firstname')) ; \\
\text{New}(\text{ZP, Init}(1, 2, 5, 'Zip code', 0, 99999)) ; \\
\text{SP}^\backslash \text{Edit} ; \\
\text{ZP}^\backslash \text{Edit} ; \\
\text{...} \\
\text{Dispose}(\text{ZP, Done}) ;
\]
An additional extension allows \texttt{New} to be used as a \textit{function}, which allocates and returns a dynamic variable of a specified type. The syntax is

\begin{verbatim}
New(T)
\end{verbatim}

or

\begin{verbatim}
New(T, Construct)
\end{verbatim}

In the first form, \( T \) can be any pointer type. In the second form, \( T \) must point to an object type, and \texttt{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:

\begin{verbatim}
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;
\end{verbatim}

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

\section*{Assembly language methods}

Method implementations written in assembly language can be linked with Turbo Pascal programs using the \$L\$ compiler directive and the \texttt{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.

```pascal
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' methods list only the method header followed by an `external' reserved word.

```pascal
{$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.
; 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 >= Xl 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
Chapter 17, Objects 235
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 runtime error 203. If you install a heap error function that returns 1 rather than the standard function result of 0, a constructor call through New will return nil when it 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.

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
A 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.

```pascal
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.
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 {$F-}$.

---

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 \texttt{external} directive, and they cannot be procedural parameters.

```pascal
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/$5E/$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

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

where \textit{Localsize} is the size of the local variables. The \texttt{SUB} instruction is only present if \textit{Localsize} is not 0. If the procedure's call model is near, the parameters start at \texttt{BP + 4}; if it is far, they start at \texttt{BP + 6}.

The standard exit code is

```assembly
mov sp, bp    ;Dealocate local variables
pop bp        ;Restore BP
ret ParamSize ;Remove parameters and return
```
where \( \text{ParamSize} \) 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 \textit{Halt}, or a termination due to a runtime error.

An exit procedure takes no parameters, and must be compiled in the \{\texttt{SF+}\} 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 \texttt{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 \texttt{ExitProc}. The following program demonstrates a skeleton method of implementing an exit procedure:

```pascal
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.
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):

```pascal
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,

```pascal
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:

```pascal
push  ax
push  bx
push  cx
push  dx
push  si
push  di
push  ds
push  es
push  bp
mov   bp,sp
sub   sp,LocalSize
```
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 ax, SEG DATA
mov ds, ax
```

```
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.
Chapter 19, Input and output issues

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

```pascal
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:

```pascal
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.
Automatic optimizations

Turbo Pascal performs several different types of code optimi-
izations, 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 \textit{Abs, Chr, Hi, Length, Lo, Odd, Ord,}
\textit{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
\textit{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:

```pascal
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

```pascal
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:

\[
\begin{align*}
T & := F(J); \\
I & := T \div G(J);
\end{align*}
\]

As an exception to this rule, when short-circuit evaluation is enabled (the \{SB\} state), Boolean operands grouped with \texttt{and} or \texttt{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 

### Shift instead of multiply

The operation \( X \times C \), where \( C \) is a constant and a power of 2, is coded using a \texttt{SHL} instruction.

Likewise, when the size of an array's components is a power of 2, a \texttt{SHL} instruction (not a \texttt{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.

Data alignment is controlled through the \$A compiler directive. In the default \{SA+\} state, variables and typed constants are aligned as described above. In the \{SA-\} state, no alignment measures are taken.

For further details, refer to Chapter 21, "Compiler directives."
Dead code removal

Statements that are known never to execute do not generate any code. For example, these constructs don't generate any code:

```pascal
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:

```pascal
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.
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}
{SM 65520,8192,655360}
{$DEFINE Debug}
```
You can put compiler directives directly into your source code. You can also change the default directives for both the command-line compiler (TPC.EXE) and the IDE (TURBO.EXE). The Options | Compiler menu contains all the compiler directives; any changes you make to the settings there will affect all subsequent compilations. When using the command-line compiler, you can specify compiler directives on the command line (for example, TPC /$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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$A+} or {$A-}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$A+}</td>
</tr>
<tr>
<td>Type</td>
<td>Global</td>
</tr>
<tr>
<td>Menu equivalent</td>
<td>Options</td>
</tr>
</tbody>
</table>
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**
Options | Compiler | Complete Boolean 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.
Boolean evaluation

In the \{$SB-$} 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$D+$} or {$D-$}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$D+$}</td>
</tr>
<tr>
<td>Type</td>
<td>Global</td>
</tr>
<tr>
<td>Menu equivalent</td>
<td>Options</td>
</tr>
</tbody>
</table>
| 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 Debug 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 Search | Find Error. The Debugging (Options | Debugger) and Map File (Options | Linker) 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 Compile | Destination to Disk and check Standalone in Options | Debugger | Debugging.
Emulation

Syntax \{$E+$} or \{$E-$}\n
Default \{$E+$}\n
Type Global

Menu equivalent Options | Compiler | Emulation

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 \{$F-$}\n
Default \{$F-$}\n
Type Local

Menu equivalent Options | Compiler | Force 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 \texttt{interface} section of a unit; near otherwise.

The near and far call models are described in full in Chapter 18, “Control issues.”
Force far calls

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 Options | Compiler | 286 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 Options | Compiler | 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>{$L+$} or {$L-$}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>{$L+$}</td>
</tr>
<tr>
<td>Type</td>
<td>Global</td>
</tr>
<tr>
<td>Menu equivalent</td>
<td>Options</td>
</tr>
</tbody>
</table>
| 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 Window | Call Stack 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 (Options | Linker) and Debugging (Options | Debugger) 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 Debug Information is unchecked {$D-$}.
Numeric processing

Syntax \{\$N+\} or \{\$N-\}
Default \{\$N-\}
Type Global
Menu equivalents Options | Compiler | 8087/80287
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 Options | Compiler | Overlays 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.

\[\Rightarrow\] 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 Options | Compiler | Range 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 Options | Compiler | Stack 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

Var-string checking

Syntax

{$V+} or {$V-}

Default

{$V+}

Type

Local

Menu equivalent Options | Compiler | Strict Var-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 Options | Compiler | Extended 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.

Turbo Pascal Programmer’s Guide
Include file

**Syntax**  \( \{$I \text{filename}\} \)

**Type**  Local

**Menu equivalent**  Options | Directories | 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 \( \{$I \text{filename}\} \) directive. The default extension for fileame 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 Options | Directories | 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**  \( \{$L \text{filename}\} \)

**Type**  Local

**Menu equivalent**  Options | Directories | Object 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 Options | Directories | Object Directories input box (or in the directories specified in the /O option on the TPC command line). For further details
Memory allocation sizes

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

Options | Memory 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 {$O unitname} directive must have been compiled with Overlays Allowed set to On in the IDE (the equivalent of the {$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

```
{$IFDEF Debug}
  Writeln('X = ', X);
{$ENDIF}
```

causes the source text between {$IFDEF} and {$ENDIF} to be compiled only if the condition specified in {$IFDEF} is True; if the condition is False, the source text between the two directives is ignored.

The second conditional compilation construct

```
{$IFDEF CPU87}
  {$N+}
type
    Real = Double;
{$ELSE}
  {$N-}
type
    Single = Real;
    Double = Real;
    Extended = Real;
    Comp = Real;
{$ENDIF}
```

causes either the source text between {$IFDEF} and {$ELSE} or the source text between {$ELSE} and {$ENDIF} to be compiled, based on the condition specified by the {$IFDEF}.

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 {$IFDEF}, the corresponding {$ENDIF} must be found within the same source file—which means there must be an equal number of {$IFDEF}'s and {$ENDIF}'s in every source file.
Conditional compilation is based on the evaluation of conditional symbols. Conditional symbols are defined and undefined (forgotten) using the directives

\[
\begin{align*}
\{$DEFINE\ name\} \\
\{$UNDEF \ name\}
\end{align*}
\]

You can also use the /D switch in the command-line compiler (or place it in the Conditional Defines input box from within Options \(\text{Compiler 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

```pascal
const
  Debug = True;
begin
  {$IFDEF Debug}
  Writeln('Debug is on');
  {$ENDIF}
end;
```

will not compile the \(\text{Writeln}\) statement. Likewise, the construct

```pascal
{$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 Conditional Defines input box (Options | Compiler), 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 `$IFxxx` and the next `$ENDIF`.

The ENDIF directive

**Syntax**

```
{$ENDIF}
```

**Remarks**

Ends the conditional compilation initiated by the last `$IFxxx` directive.
Using Turbo Pascal with assembly language
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
  int 16H
  mov CharCode,al
  mov ScanCode,ah
end;

asm
  push ds
  lds si,Source
  les di,Dest
  mov cx,Count
  cld
  rep movsb
  pop 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

```
```

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:

```pascal
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
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
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 **SEGxx**, followed by a **REPxx**). 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 **REPxx** prefix if an interrupt occurs in the middle of a repeated string instruction, so a **LOCK** or **SEGxx** prefix cannot safely be coded before a **REPxx** 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 \{SG+\} state (80286 code generation enabled), and 80287 opcodes are available only in the \{SG+,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.

```pascal
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
where Stop is not within reach of a short jump is converted to a machine code sequence that corresponds to

\[
\begin{align*}
&\text{JC} \quad \text{Stop} \\
&\text{jnc} \quad \text{Skip} \\
&\text{jmp} \quad \text{Stop} \\
&\text{Skip:}
\end{align*}
\]

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 \texttt{NEAR PTR} or \texttt{FAR PTR} construct. For example, the assembler statements

\[
\begin{align*}
&\text{jmp} \quad \text{NEAR PTR Stop} \\
&\text{jmp} \quad \text{FAR PTR Stop}
\end{align*}
\]

will always generate a near jump and a far jump, respectively, even if Stop is a label within reach of a short jump.

\textbf{Assembler directives}

Turbo Pascal's inline assembler supports three assembler directives: \texttt{DB} (define byte), \texttt{DW} (define word), and \texttt{DD} (define double word). They each generate data corresponding to the comma-separated operands that follow the directive.

The \texttt{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 \texttt{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 \texttt{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:

```pascal
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:

```pascal
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

```pascal
var
  ByteVar: Byte;
  WordVar: Word;
```
 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:

<table>
<thead>
<tr>
<th>AH</th>
<th>CL</th>
<th>FAR</th>
<th>SEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>CS</td>
<td>HIGH</td>
<td>SHL</td>
</tr>
<tr>
<td>AND</td>
<td>CX</td>
<td>LOW</td>
<td>SHR</td>
</tr>
<tr>
<td>AX</td>
<td>DH</td>
<td>MOD</td>
<td>SI</td>
</tr>
<tr>
<td>BH</td>
<td>DI</td>
<td>NEAR</td>
<td>SP</td>
</tr>
<tr>
<td>BL</td>
<td>DL</td>
<td>NOT</td>
<td>SS</td>
</tr>
<tr>
<td>BP</td>
<td>DS</td>
<td>OFFSET</td>
<td>ST</td>
</tr>
<tr>
<td>BX</td>
<td>DWORD</td>
<td>OR</td>
<td>TBYTE</td>
</tr>
<tr>
<td>BYTE</td>
<td>DX</td>
<td>PTR</td>
<td>TYPE</td>
</tr>
<tr>
<td>CH</td>
<td>ES</td>
<td>QWORD</td>
<td>WORD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XOR</td>
</tr>
</tbody>
</table>

The reserved words always take precedence over user-defined identifiers. For instance, the code fragment

```pascal
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:

```pascal
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

```pascal
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid inline assembler statement:

```pascal
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,

```pascal
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

\[
\begin{align*}
\text{asm} \\
\text{mov} & \text{ ax, } X \\
\text{add} & \text{ ax, } Y \\
\text{mov} & \text{ Z, ax}
\end{align*}
\end{equation}

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

\[
\begin{align*}
\text{asm} \\
\text{mov} & \text{ ax, } X+4 \\
\end{align*}
\end{equation}

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:

\[
\begin{align*}
\text{asm} \\
\text{MOV} & \text{ AX, } X \\
\text{ADD} & \text{ AX, } 4
\end{align*}
\end{equation}

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

\[
\text{Ord}(Ch1) + \text{Ord}(Ch2) \text{ shl } 8 + \text{Ord}(Ch3) \text{ shl } 16 + \text{Ord}(Ch4) \text{ shl } 24
\]

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\'  00000610H
\['a\'  20020610H
\['a'' 000000E2H
\['a''A'  00000020H
\[not \ 'a\'  FFFFFF9EH

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
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

```pascal
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:

```asm
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>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Class</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Address of label</td>
<td>Memory</td>
<td>SHORT</td>
</tr>
<tr>
<td>Constant</td>
<td>Value of constant</td>
<td>Immediate</td>
<td>0</td>
</tr>
<tr>
<td>Type</td>
<td>0</td>
<td>Memory</td>
<td>Size of type</td>
</tr>
<tr>
<td>Field</td>
<td>Offset of field</td>
<td>Memory</td>
<td>Size of type</td>
</tr>
<tr>
<td>Variable</td>
<td>Address of variable</td>
<td>Memory</td>
<td>Size of type</td>
</tr>
<tr>
<td>Procedure</td>
<td>Address of procedure</td>
<td>Memory</td>
<td>NEAR or FAR</td>
</tr>
<tr>
<td>Function</td>
<td>Address of function</td>
<td>Memory</td>
<td>NEAR or FAR</td>
</tr>
<tr>
<td>Unit</td>
<td>0</td>
<td>Immediate</td>
<td>0</td>
</tr>
<tr>
<td>@Code</td>
<td>Code segment address</td>
<td>Memory</td>
<td>0FF0F0H</td>
</tr>
<tr>
<td>@Data</td>
<td>Data segment address</td>
<td>Memory</td>
<td>0FF0F0H</td>
</tr>
<tr>
<td>@Result</td>
<td>Result var offset</td>
<td>Memory</td>
<td>Size of type</td>
</tr>
</tbody>
</table>

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

```pascal
procedure Test;
var
  Count: Integer;
```

the instruction

```asm
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,

```pascal
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
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
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
asm
  mov   al, [lOOH]
  mov   bx, [lOOH]
end;
```

The inline assembler permits both of these instructions, since the expression [lOOH] 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
asm
  inc   BYTE PTR [lOOH]
  imul  WORD PTR [lOOH]
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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>1</td>
</tr>
<tr>
<td>WORD</td>
<td>2</td>
</tr>
<tr>
<td>DWORD</td>
<td>4</td>
</tr>
<tr>
<td>QWORD</td>
<td>8</td>
</tr>
<tr>
<td>TBYTE</td>
<td>10</td>
</tr>
<tr>
<td>NEAR</td>
<td>0FFFFH</td>
</tr>
<tr>
<td>FAR</td>
<td>0FFFFH</td>
</tr>
</tbody>
</table>

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,

```pascal
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:

```pascal
asm
  push cs
  call NEAR PTR FarProc
end;
```

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>
<thead>
<tr>
<th>Operator(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Identifier override operator</td>
</tr>
<tr>
<td>(), [], ., *</td>
<td>Structure member selector</td>
</tr>
<tr>
<td><strong>HIGH</strong>, <strong>LOW</strong></td>
<td></td>
</tr>
<tr>
<td>+, −</td>
<td>Unary operators</td>
</tr>
<tr>
<td>:</td>
<td>Segment override operator</td>
</tr>
<tr>
<td><strong>OFFSET, SEG, TYPE, PTR,</strong></td>
<td></td>
</tr>
<tr>
<td>* , /, MOD, SHL, SHR</td>
<td>Binary addition/ subtraction</td>
</tr>
<tr>
<td>+, −</td>
<td>operators</td>
</tr>
<tr>
<td><strong>NOT, AND, OR, XOR</strong></td>
<td>Bitwise operators</td>
</tr>
</tbody>
</table>
& **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:

```pascal
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:

  ```assembly
  push bp
  mov bp,sp
  sub sp,Locals
  mov sp,bp
  pop bp
  ret
  ```

  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 \textit{Str} in this case refers to a local variable, since the compiler automatically generates entry code that copies the actual parameter into local storage.

```pascal
function UpperCase(Str: String): String;
begin
  asm
    cld
    lea 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:
  end;
end;
end;
```

The second example is an assembler version of the \textit{UpperCase} function. In this case, \textit{Str} is not copied into local storage, and the function must treat \textit{Str} as a \textbf{var} parameter.

```pascal
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
```

\textit{Chapter 22, The Inline assembler} 305
cmp al,'a'
jb @2
cmp al,'z'
ja @2
sub al,20H
@2:
    stosb
    loop @1
@3:
    pop ds
end:
Chapter 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,

```pascal
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 (TASM) makes it easy to program routines in assembly language and interface them into your Turbo Pascal programs. Turbo Assembler provides simplified segmentation,
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.

```pascal
unit Strings;
interface
function UpperCase(S: String): String;
function StringOf(Ch: Char; Count: Byte): String;
implementation
{$L STRS}
function UpperCase; external;
```

Chapter 23, Linking assembler code
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
UpperCase 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 ;Skip if empty string
    jcxz U3 ;Load character
    lodsb ;Skip if not 'a'..'z'
    cmp al, 'a' ;Convert to uppercase
    jb U2 ;Store in result
    cmp al, 'z'
    ja U2
    sub al, 'a'-'a'
    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 \textit{CheckRange} procedure is shown next. It must be assembled into a file called \texttt{CHECK.OBJ} before the \textit{Numbers} program can be compiled. Note that the procedure uses the near call model because it is declared in a program.

\begin{verbatim}
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
    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
\end{verbatim}
Here's a Turbo Assembler version of the previous assembly language example that takes advantage of TASM's support for Turbo Pascal:

```assembly
.MODEL TPASCAL ;Turbo Pascal code model
LOCALS @0 ;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:

```
inline statement  --> inline ( inline element )
```

Each inline element consists of an optional size specifier, < or >, and a constant or a variable identifier, followed by zero or more offset specifiers (see the syntax that follows). An offset specifier consists of a + or a − followed by a constant.

```
inline element
```

```
constant
```

```
<
```

```
>
```

```
variable identifier
```

```
sign
```

```
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.

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/
    $8B/$BE/Count/
    $8B/$BE/Data/
    $FC/
    $E3/$AB);
  { LES DI,Dest[BP] } { MOV CX,Count[BP] } { MOV AX,Data[BP] } { CLD } { 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:

```
inline directive  --- inline statement
```

When a normal procedure or function is called (including one that contains inline statements), the compiler generates code that pushes the parameters (if any) onto the stack, and then generates a CALL instruction to call the procedure or function. However, when you call an inline procedure or function, the compiler generates code from the inline directive instead of the CALL. Here’s a short example of two inline procedures:

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

When DisableInterrupts is called, it generates 1 byte of code—a CLI instruction.

Procedures and functions declared with inline directives can have parameters; however, the parameters 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/$58/$F7/$EA);
{ POP AX ;Pop X }
{ POP DX ;Pop Y }
{ 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.
Appendixes
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 Compile | Destination is set to Memory, set it to Disk in the integrated environment.
- If the Memory radio button is chosen (O | L | Link 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 begins and ends 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 \texttt{forward}, but its definition was never found.

\textbf{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.

\textbf{61 Invalid typecast.}

\begin{itemize}
  \item The sizes of the variable reference and the destination type differ in a variable typecast.
  \item You are attempting to typecast an expression where only a variable reference is allowed.
\end{itemize}

\textbf{62 Division by zero.}

The preceding operand attempts to divide by zero.

\textbf{63 Invalid file type.}

The file type is not supported by the file-handling procedure; for example, \texttt{Readln} with a typed file or \texttt{Seek} with a text file.

\textbf{64 Cannot Read or Write variables of this type.}

\begin{itemize}
  \item \texttt{Read} and \texttt{Readln} can input variables of \texttt{Char}, integer, real, and string types.
  \item \texttt{Write} and \texttt{Writeln} can output variables of \texttt{Char}, integer, real, string, and \texttt{Boolean} types.
\end{itemize}

\textbf{65 Pointer variable expected.}

The preceding variable must be of a pointer type.

\textbf{66 String variable expected.}

The preceding variable must be of a string type.

\textbf{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:

```pascal
unit U1;
uses U2;
...

unit 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 Compile | Make or Compile | Build 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 Search | Find Error 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 `$IFDEF`s and `$ENDIF`s in a source file.

130 **Error in initial conditional defines.**

The initial conditional symbols specified in **Options | Compiler | Conditional 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 {SF+} 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, \texttt{DEC 1, MOV AX,CL} or \texttt{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, \texttt{MOV AX,BX+SI} instead of \texttt{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 \texttt{Var} is a variable and \texttt{Const} is a constant, then the instructions \texttt{MOV AX, Const+Const} and \texttt{MOV AX, Var+Const} are valid, but \texttt{MOV AX, Var+Var} is not.

158 Invalid register combination.

Valid index register combinations are \texttt{[BX]}, \texttt{[BP]}, \texttt{[SI]}, \texttt{[DI]}, \texttt{[BX+SI]}, \texttt{[BX+DI]}, \texttt{[BP+SI]}, and \texttt{[BP+DI]}. Other index register combinations (such as \texttt{[AX]}, \texttt{[BP+BX]}, and \texttt{[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 \texttt{[BP]} in references to such variables, so even though a construct like \texttt{Local[BX]} (where \texttt{Local} is a local variable) appears valid, it is not since the final operand would become \texttt{Local[BP+BX]}.

159 286/287 instructions are not enabled.

Use a \texttt{($)G+}$ 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:
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 $l$, $mod$, 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
Ovrlnit, or the call to Ovrlnit failed. Note that, if you have initialization
code in any of your overlaid units, you must create an
additional non-overlaid unit which calls Ovrlnit, 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.
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

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>＠</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ETX</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>TAB</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>LF</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>VT</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>FF</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>CR</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>SO</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>SI</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>DLE</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>DC1</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>DC2</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>DC3</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>DC4</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>NAK</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>SYN</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>ETB</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>CAN</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>EM</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>SUB</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>ESC</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>FS</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>GS</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>RS</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>US</td>
</tr>
</tbody>
</table>

The caret in ＠ means to press the Ctrl key and type ＠.
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>80</td>
<td>ç</td>
<td>160</td>
<td>A0</td>
<td>á</td>
<td>192</td>
<td>C0</td>
<td>l</td>
<td>224</td>
<td>E0</td>
<td>α</td>
</tr>
<tr>
<td>129</td>
<td>81</td>
<td>ü</td>
<td>161</td>
<td>A1</td>
<td>i</td>
<td>193</td>
<td>C1</td>
<td>l</td>
<td>225</td>
<td>E1</td>
<td>β</td>
</tr>
<tr>
<td>130</td>
<td>82</td>
<td>ë</td>
<td>162</td>
<td>A2</td>
<td>ö</td>
<td>194</td>
<td>C2</td>
<td>t</td>
<td>226</td>
<td>E2</td>
<td>Γ</td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td>á</td>
<td>163</td>
<td>A3</td>
<td>ü</td>
<td>195</td>
<td>C3</td>
<td>n</td>
<td>227</td>
<td>E3</td>
<td>Π</td>
</tr>
<tr>
<td>132</td>
<td>84</td>
<td>ä</td>
<td>164</td>
<td>A4</td>
<td>ŋ</td>
<td>196</td>
<td>C4</td>
<td>–</td>
<td>228</td>
<td>E4</td>
<td>Σ</td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td>å</td>
<td>165</td>
<td>A5</td>
<td>ñ</td>
<td>197</td>
<td>C5</td>
<td>t</td>
<td>229</td>
<td>E5</td>
<td>σ</td>
</tr>
<tr>
<td>134</td>
<td>86</td>
<td>ä</td>
<td>166</td>
<td>A6</td>
<td>a</td>
<td>198</td>
<td>C6</td>
<td>t</td>
<td>230</td>
<td>E6</td>
<td>μ</td>
</tr>
<tr>
<td>135</td>
<td>87</td>
<td>ç</td>
<td>167</td>
<td>A7</td>
<td>o</td>
<td>199</td>
<td>C7</td>
<td>n</td>
<td>231</td>
<td>E7</td>
<td>ι</td>
</tr>
<tr>
<td>136</td>
<td>88</td>
<td>ê</td>
<td>168</td>
<td>A8</td>
<td>e</td>
<td>200</td>
<td>C8</td>
<td>l</td>
<td>232</td>
<td>E8</td>
<td>Φ</td>
</tr>
<tr>
<td>137</td>
<td>89</td>
<td>ë</td>
<td>169</td>
<td>A9</td>
<td>C</td>
<td>201</td>
<td>C9</td>
<td>n</td>
<td>233</td>
<td>E9</td>
<td>θ</td>
</tr>
<tr>
<td>138</td>
<td>8A</td>
<td>ē</td>
<td>170</td>
<td>AA</td>
<td>1</td>
<td>202</td>
<td>CA</td>
<td>l</td>
<td>234</td>
<td>EA</td>
<td>Ω</td>
</tr>
<tr>
<td>139</td>
<td>8B</td>
<td>ť</td>
<td>171</td>
<td>AB</td>
<td>$</td>
<td>203</td>
<td>CB</td>
<td>t</td>
<td>235</td>
<td>EB</td>
<td>δ</td>
</tr>
<tr>
<td>140</td>
<td>8C</td>
<td>i</td>
<td>172</td>
<td>AC</td>
<td>3</td>
<td>204</td>
<td>CC</td>
<td>t</td>
<td>236</td>
<td>EC</td>
<td>ι</td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td>ñ</td>
<td>173</td>
<td>AD</td>
<td>ñ</td>
<td>205</td>
<td>CD</td>
<td>$</td>
<td>237</td>
<td>ED</td>
<td>Φ</td>
</tr>
<tr>
<td>142</td>
<td>8E</td>
<td>Ė</td>
<td>174</td>
<td>AE</td>
<td>ñ</td>
<td>206</td>
<td>CE</td>
<td>ñ</td>
<td>238</td>
<td>EE</td>
<td>Ε</td>
</tr>
<tr>
<td>143</td>
<td>8F</td>
<td>Á</td>
<td>175</td>
<td>AF</td>
<td>2</td>
<td>207</td>
<td>CF</td>
<td>ñ</td>
<td>239</td>
<td>EF</td>
<td>Ω</td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td>Ė</td>
<td>176</td>
<td>B0</td>
<td>3</td>
<td>208</td>
<td>D0</td>
<td>n</td>
<td>240</td>
<td>F0</td>
<td>Η</td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td>æ</td>
<td>177</td>
<td>B1</td>
<td>3</td>
<td>209</td>
<td>D1</td>
<td>ñ</td>
<td>241</td>
<td>F1</td>
<td>Π</td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td>ì</td>
<td>178</td>
<td>B2</td>
<td>3</td>
<td>210</td>
<td>D2</td>
<td>ñ</td>
<td>242</td>
<td>F2</td>
<td>Π</td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td>õ</td>
<td>179</td>
<td>B3</td>
<td>3</td>
<td>211</td>
<td>D3</td>
<td>ñ</td>
<td>243</td>
<td>F3</td>
<td>Π</td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td>ö</td>
<td>180</td>
<td>B4</td>
<td>3</td>
<td>212</td>
<td>D4</td>
<td>ñ</td>
<td>244</td>
<td>F4</td>
<td>Π</td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td>ò</td>
<td>181</td>
<td>B5</td>
<td>3</td>
<td>213</td>
<td>D5</td>
<td>ñ</td>
<td>245</td>
<td>F5</td>
<td>Π</td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td>ù</td>
<td>182</td>
<td>B6</td>
<td>3</td>
<td>214</td>
<td>D6</td>
<td>ñ</td>
<td>246</td>
<td>F6</td>
<td>Π</td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td>ü</td>
<td>183</td>
<td>B7</td>
<td>3</td>
<td>215</td>
<td>D7</td>
<td>ñ</td>
<td>247</td>
<td>F7</td>
<td>Π</td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td>ý</td>
<td>184</td>
<td>B8</td>
<td>3</td>
<td>216</td>
<td>D8</td>
<td>ñ</td>
<td>248</td>
<td>F8</td>
<td>Π</td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td>ö</td>
<td>185</td>
<td>B9</td>
<td>3</td>
<td>217</td>
<td>D9</td>
<td>ñ</td>
<td>249</td>
<td>F9</td>
<td>Π</td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td>ü</td>
<td>186</td>
<td>BA</td>
<td>3</td>
<td>218</td>
<td>DA</td>
<td>3</td>
<td>250</td>
<td>FA</td>
<td>Π</td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td>ç</td>
<td>187</td>
<td>BB</td>
<td>3</td>
<td>219</td>
<td>DB</td>
<td>3</td>
<td>251</td>
<td>FB</td>
<td>Π</td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td>£</td>
<td>188</td>
<td>BC</td>
<td>3</td>
<td>220</td>
<td>DC</td>
<td>3</td>
<td>252</td>
<td>FC</td>
<td>Π</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>¥</td>
<td>189</td>
<td>BD</td>
<td>3</td>
<td>221</td>
<td>DD</td>
<td>3</td>
<td>253</td>
<td>FD</td>
<td>Π</td>
</tr>
<tr>
<td>158</td>
<td>9E</td>
<td>¶</td>
<td>190</td>
<td>BE</td>
<td>3</td>
<td>222</td>
<td>DE</td>
<td>3</td>
<td>254</td>
<td>FE</td>
<td>Π</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>$</td>
<td>191</td>
<td>BF</td>
<td>3</td>
<td>223</td>
<td>DF</td>
<td>3</td>
<td>255</td>
<td>FF</td>
<td>Π</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Second code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>NUL (null character)</td>
</tr>
<tr>
<td>15</td>
<td>Shift Tab (→&lt;vv)</td>
</tr>
<tr>
<td>16-25</td>
<td>Alt-Q/W/E/R/T/Y/U/I/O/P</td>
</tr>
<tr>
<td>30-38</td>
<td>Alt-A/S/D/F/G/H/I/J/K/L</td>
</tr>
<tr>
<td>44-50</td>
<td>Alt-Z/X/C/V/B/N/M</td>
</tr>
<tr>
<td>59-68</td>
<td>Keys F1-F10 (disabled as softkeys)</td>
</tr>
<tr>
<td>71</td>
<td>Home</td>
</tr>
<tr>
<td>72</td>
<td>↑</td>
</tr>
<tr>
<td>73</td>
<td>PgUp</td>
</tr>
<tr>
<td>75</td>
<td>←</td>
</tr>
<tr>
<td>77</td>
<td>→</td>
</tr>
<tr>
<td>79</td>
<td>End</td>
</tr>
<tr>
<td>80</td>
<td>↓</td>
</tr>
<tr>
<td>81</td>
<td>PgDn</td>
</tr>
<tr>
<td>82</td>
<td>Ins</td>
</tr>
<tr>
<td>83</td>
<td>Def</td>
</tr>
<tr>
<td>84-93</td>
<td>F11-F20 (Shift-F1 to Shift-F10)</td>
</tr>
<tr>
<td>94-103</td>
<td>F21-F30 (Ctrl-F1 through F10)</td>
</tr>
<tr>
<td>104-113</td>
<td>F31-F40 (Alt-F1 through F10)</td>
</tr>
<tr>
<td>114</td>
<td>Ctrl-PrintSc</td>
</tr>
<tr>
<td>115</td>
<td>Ctrl←</td>
</tr>
<tr>
<td>116</td>
<td>Ctrl→</td>
</tr>
<tr>
<td>117</td>
<td>Ctrl-End</td>
</tr>
<tr>
<td>118</td>
<td>Ctrl-PgDn</td>
</tr>
<tr>
<td>119</td>
<td>Ctrl-Home</td>
</tr>
<tr>
<td>120-131</td>
<td>Alt-1/2/3/4/5/6/7/8/9/0/-=</td>
</tr>
<tr>
<td>132</td>
<td>Ctrl-PgUp</td>
</tr>
<tr>
<td>133</td>
<td>F11</td>
</tr>
<tr>
<td>134</td>
<td>F12</td>
</tr>
<tr>
<td>135</td>
<td>Shift-F11</td>
</tr>
<tr>
<td>136</td>
<td>Shift-F12</td>
</tr>
<tr>
<td>137</td>
<td>Ctrl-F11</td>
</tr>
<tr>
<td>138</td>
<td>Ctrl-F12</td>
</tr>
<tr>
<td>139</td>
<td>Alt-F11</td>
</tr>
<tr>
<td>140</td>
<td>Alt-F12</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Key</th>
<th>Scan Code in Hex</th>
<th>Key in Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Esc</td>
<td>01</td>
<td>←/→</td>
</tr>
<tr>
<td>1</td>
<td>02</td>
<td>Q</td>
</tr>
<tr>
<td>@ 2</td>
<td>03</td>
<td>W</td>
</tr>
<tr>
<td># 3</td>
<td>04</td>
<td>E</td>
</tr>
<tr>
<td>$ 4</td>
<td>05</td>
<td>R</td>
</tr>
<tr>
<td>% 5</td>
<td>06</td>
<td>T</td>
</tr>
<tr>
<td>^ 6</td>
<td>07</td>
<td>Y</td>
</tr>
<tr>
<td>&amp; 7</td>
<td>08</td>
<td>U</td>
</tr>
<tr>
<td>* 8</td>
<td>09</td>
<td>I</td>
</tr>
<tr>
<td>( 9</td>
<td>0A</td>
<td>O</td>
</tr>
<tr>
<td>) 0</td>
<td>0B</td>
<td>P</td>
</tr>
<tr>
<td>_</td>
<td>0C</td>
<td>[ [</td>
</tr>
<tr>
<td>+=</td>
<td>0D</td>
<td>] ]</td>
</tr>
<tr>
<td>Backspace</td>
<td>0E</td>
<td>Return</td>
</tr>
<tr>
<td>Ctrl</td>
<td>1D</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>1E</td>
<td>Z</td>
</tr>
<tr>
<td>S</td>
<td>1F</td>
<td>X</td>
</tr>
<tr>
<td>D</td>
<td>20</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>21</td>
<td>V</td>
</tr>
<tr>
<td>G</td>
<td>22</td>
<td>B</td>
</tr>
<tr>
<td>H</td>
<td>23</td>
<td>N</td>
</tr>
<tr>
<td>J</td>
<td>24</td>
<td>M</td>
</tr>
<tr>
<td>K</td>
<td>25</td>
<td>,</td>
</tr>
<tr>
<td>L</td>
<td>26</td>
<td>&gt; ..</td>
</tr>
<tr>
<td>..</td>
<td>27</td>
<td>? /</td>
</tr>
<tr>
<td>^</td>
<td>28</td>
<td>→ Shift</td>
</tr>
<tr>
<td>~</td>
<td>29</td>
<td>PrtSc*</td>
</tr>
<tr>
<td>←-Shift</td>
<td>2A</td>
<td>Alt</td>
</tr>
<tr>
<td>Spacebar</td>
<td>39</td>
<td>7 Home</td>
</tr>
<tr>
<td>Caps Lock</td>
<td>3A</td>
<td>8 ↑</td>
</tr>
<tr>
<td>F1</td>
<td>3B</td>
<td>9 PgUp</td>
</tr>
<tr>
<td>F2</td>
<td>3C</td>
<td>Minus sign</td>
</tr>
<tr>
<td>F3</td>
<td>3D</td>
<td>4 ←</td>
</tr>
<tr>
<td>F4</td>
<td>3E</td>
<td>5</td>
</tr>
<tr>
<td>F5</td>
<td>3F</td>
<td>6 →</td>
</tr>
<tr>
<td>F6</td>
<td>40</td>
<td>+</td>
</tr>
<tr>
<td>F7</td>
<td>41</td>
<td>1 End</td>
</tr>
<tr>
<td>F8</td>
<td>42</td>
<td>2 ↓</td>
</tr>
<tr>
<td>F9</td>
<td>43</td>
<td>3 PgDn</td>
</tr>
<tr>
<td>F10</td>
<td>44</td>
<td>0 Ins</td>
</tr>
<tr>
<td>F11</td>
<td>4D</td>
<td>Del</td>
</tr>
<tr>
<td>F12</td>
<td>DA</td>
<td>Num Lock</td>
</tr>
<tr>
<td>Scroll Lock</td>
<td>46</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

$ See compiler, directives
8087/80287/80387 coprocessor See numeric coprocessor
8087/80287 option 268
80286 code generation compiler switch 266
256-color mode 151
286 Instructions option 266
@ (address-of) operator See address-of (@)
    operator
    ^ (pointer) symbol 39, 40, 53
# (pound) character 11

A
$A compiler directive 257, 262
Abs function 127, 255
absolute clause 49
activation, qualified 83
actual parameters 76, 83
Addr function 128
double 80
    with method designators 76
address functions 128
address-of (@) operator 40, 53, 74, 80
 Addr function 128
address functions 128
address-of (@) operator 40, 53, 74, 80
    double 80
    with method designators 76
aligning data 262
ancestors 32
and operator 71, 154
Append procedure 129, 132
Arc procedure 164, 165
ArcTan function 127
arithmetic
    functions 127
    operators 69
array-type constants 59
arrays 29, 51, 59
    types 29, 221
    variables 51
ASCII codes 347
.ASCII files 197
assembly language 271, 307
    8087 emulation and 197
examples 309
    inline
        directives 316
        statements 314
    interfacing program routines with 308
    routines, overlays and 185
Assign procedure 129, 131, 251
AssignCrt procedure 205, 251
assignment
    compatibility 38, 82
    statements 82
automatic
    call model selection, overriding 244
    word alignment 257
AX register 243, 316

B
$B compiler directive 263
Bar3D procedure 149, 162, 165
Bar procedure 165
.BGI files 149
binding
    late 36
    BIOS 199, 204
    bit images 153
    bit-mapped fonts 152
    bit-oriented routines 149
BitBlt
    operations 154
    operators 163
    bitwise operators 70
BlockRead procedure 133
blocks, program 15
BlockWrite procedure 133
Boolean
    evaluation, compiler switch 263
operators 70
  type 24, 218
values 24
Boolean Evaluation command 263
BP register 186, 246, 248, 315
brackets, in expressions 77
buffers
  overlay 171
  loading and freeing up 172
  optimization algorithm 172
  probationary area 173
BX register 243, 248
Byte data type 23

C
  calling conventions 241
    constructors and destructors 230
    methods 83, 230
case statements 87
CGA 149, 202
  CheckSnow and 203
Char data type 24, 218
characters
  special 200
  strings 10
ChDir procedure 131
CheckBreak variable 203
CheckEOF variable 203
CheckSnow variable 203
.CHR files 149
Chr function 126, 255
Circle procedure 149, 165
circular unit references 118
ClearDevice procedure 165
ClearViewPort procedure 165
Close function 253
Close procedure 131, 251
CloseGraph procedure 149, 165
ClrEol procedure 205
ClrScr procedure 205
CODE 307
COM device 135
command devices 135
command-line parameters 129
comments
  inline assembler 282, 283
  program 13
common type 24
Comp floating-point type 191, 220
comparing values of real types 193
compatibility
  assignment 38, 82
  parameter type 107
  conditional 273
compiler
  directives 13, 261-277
    $A 257, 262
    $B 263
    Boolean evaluation 263
    conditional 261, 273-277
    $D 264
    $DEFINE 274, 275
    $E 190, 265
    $ELSE 277
    $ENDIF 277
    $IFDEF 276
    $IFNDEF 276
    $IFOPT 276
    $L 267, 271, 307
    $L 233
    local symbol 267
    $M 48, 211, 272
    $N 27, 190, 195, 268
    $O 180, 268, 272
    non-overlay units and 185
    parameter 261, 271-272
    $R 227, 269
    $S 48, 269
    switch 261, 262-270
    $UNDEF 274, 276
    $V 270
    $X 270
  error messages 321
Complete Boolean Eval option 263
compound statements 85
CON device 135
Concat function 128
concatenation 72
conditional
  compilation 273
  statements 86
  symbols 274
CONST 307
constant expressions 12
restrictions 12
type definition 26
constants 141
array-type 59
Crt mode 202
declaration part 16
declarations 12
defined by Overlay unit 173
file attribute 142
folding 255
Graph unit 159
inline assembler 291, 297
merging 256
pointer-type 62
procedural-type 63
record-type 60
set-type 62
simple 12
simple-type 58
string-type 59
structured-type 59
text color 202
typed 57
object type 61
constructors
calling conventions 230
declaring 102
defined 103
error recovery 236
implementation 102
inherited 36
virtual methods and 37, 103
VMTP and 225, 228
control characters 11, 347
Copy function 128
Cos function 127
CPU symbols 275
critical errors
messages 343
trapping 138
Crt unit 132, 136, 199
constants 201
functions 205
line input 201
mode constants 202
procedures 205
special characters 200
text color 202
variables 203
CS register 248
CSEG 307
CSeg function 128
current pointer 152
CX register 248

D
$D compiler directive 264
DATA 307
data
alignment 262
corexption 138
ports 253
segment 48
types See types
date and time procedures 145
DateTime type 144
dead code removal 258
Debug Information
command 264
option 264
debugging
information switch 264
overlays 185
range-checking switch 269
run-time error messages 340
stack overflow switch 269
Dec procedure 127
decimal notation 9
declaration
c tors 102
destructors 102
methods 35, 102
object types 33
part
block 15
$DEFINE compiler directive 274, 275
Delay procedure 206
Delete procedure 127
DelLine procedure 206
descendants 32
designators
field 52
method 38, 52
   @ (address operator) with 76
destructors 103
calling conventions 230
declaring 102
defined 103
   implementation 102
DetectGraph procedure 159, 165
devices 134
drivers 251
   handlers 248, 249
DI register 248
direct memory 223
directives 7, See compiler, directives
   external 99
   far 97
   forward 98
   inline 99
   inline assembler 281, 303, 304
   interrupt 97
   near 97
   private 7
directories
   scan procedures for 144
DirectVideo variable 204
DiskFree function 146
disks
   status functions 146
DiskSize function 146
Dispose procedure 126, 212, 213, 215
   extended syntax 227, 231
   constructor passed as parameter 103, 231
div operator 69
domain, object 32
DOS
   device handling 249
   devices 134
   environment 209
   error level 247
   exit code 246
   operating system routines 141
   registers and 143
Dos unit 141
   constants 141
   date and time procedures 145
disk status functions 146
DosError in 145
environment-handling functions 147
file-handling procedures and functions 146
interrupt support procedures 146
miscellaneous functions 147
miscellaneous procedures 147
process-handling procedures and functions 146, 147
types 143
DosError variable 145
DosExitCode function 147
DosVersion function 147
Double floating-point type 191, 220
DrawPoly procedure 149, 165
drivers
   graphics 149
DS register 246, 248, 308, 315
DSEG 307
DSeg function 128
DX register 243, 248
dynamic
   memory allocation 125
   functions 125
   variables 39, 48, 53, 211
dynamic object instances
   allocation and disposal 103, 230

E
$E compiler directive 190, 265
EGA, CheckSnow and 203
Ellipse procedure 164, 165
$ELSE compiler directive 277
empty set 39
EMS memory, overlay files and 170, 181
emulating numeric coprocessor (8087) 27
   compiler switch 265
Emulation
   command 265
   option 265
end of file
   error messages 323
$ENDIF compiler directive 277
entry code, procedures and functions 245
enumerated type 25, 218
EnvCount function 147
EnvStr function 147
Eof function 130
Eoln function 133
Erase procedure 131
error checking
dynamic object allocation 236
virtual method calls 227
ErrorAddr variable 138, 247
errors
critical 343
fatal, in OvrInit 183
handling 154
messages 321
critical 343
fatal 344
range 269
reporting 246
run-time See run-time, errors
ES register 248
.EXE files 169
building 258
Exec procedure 146
exit
functions 245
procedures 245, 246
implementing 138
Exit procedure 125
ExitCode variable 138, 247
exiting a program 246
ExitProc variable 138, 246
Exp function 127
Expanded Memory Specification See EMS
memory
exponents 219
expressions 65
constant 12
address 57
examples 68
inline assembler 290
classes 297-298
elements of 291-297
Extended
floating-point type 191, 220
range arithmetic 191
extended
key codes 199, 350
memory support See EMS memory
syntax 270
Extended Syntax option 270
extensibility 38
external
declarations 99, 271, 307
procedure errors 327
external (reserved word) 233
ExternProc 186
EXTRN definition errors 308, 327
F
$F compiler directive 181, 244, 265
factor (syntax) 66
Fail procedure 237
far
directives 97
FAR calls 243
model 180
forcing use of 246
requirement 170
far calls
model
forcing use of 265
fatal run-time errors 344
FExpand function 146
Fibonacci numbers 194
field
designators 52
list (of records) 30
record 52
fields, object 32
designators 52
scope 35, 102
file-handling functions 146
file-handling procedures 146
 FileMode variable 133, 138
FilePos function 130
FileRec 143, 222
files
access, read-only 133
access-denied error 341
.ASM 197
attributes
constants 142
.BGI 149
buffer 223
.CHR 149
.EXE 169
building 258
functions for 130
handles 222
I/O 125, 199
modes 222
  constants 142
.OBJ 307
  linking with 271
.OVR 169
procedures for 131
record types 143
text 131
typed 138, 222
types 39, 222
  untyped 133, 138, 222
FileSize function 130
FillChar procedure 128
FillEllipse procedure 165
FillPoly procedure 153, 165
FindFirst procedure 142, 146
SearchRec and 144
FindNext procedure 142, 146
SearchRec and 144
fixed part (of records) 30
flags constants 141
floating-point
  calculations, type Real and 191
  code generation, switching 190
errors 345
numbers 189
numeric coprocessor (8087) 27
parameters 242
routines 125
software 27
  types See types, floating-point
FloodFill procedure 151, 153, 165
Flush function 252, 253
Flush procedure 132
Font8x8 variable 202
fonts
  files 157
for statements, syntax 90
Force Far Calls
  command 181, 265
  option 265
force far calls compiler switch 265
formal parameters 83, 105
forward declarations 98
Frac function 127
free list 215
FreeList variable 137
FreeMem procedure 126, 212, 213, 215
FSearch function 146
FSplit procedure 146
functions 95
  address 128
  arithmetic 127
  body 100
  calls 76, 241
  Crt unit 205
dynamic allocation 126
  entry/exit code
    inline assembler 304
extended syntax 77, 270
file-handling 146
Graph unit 167
headings 100
heap error 236
inline assembler 303-306
methods denoting 77
nested 110
ordinal 127
OvrGetRetry 173, 179
parameters
  inline assembler 303
pointer 128
results 243
  discarding 77, 270
returns
  inline assembler 304
SizeOf 228
stack frame
  inline assembler 304
standard 125
string 128
type 126
transfer 126
TypeOf 229

$G compiler directive 266
GetArcCoords procedure 164, 165
getAspctRatio procedure 165
GetBkColor function 167
GetCBreak procedure 147
GetColor function 167
GetDate procedure 145
GetDefaultPalette function 163, 167
GetDir procedure 131
GetDriverName function 167
GetEnv function 147
GetFAttr procedure 142, 146
GetFillPattern procedure 164, 165
GetFillSettings procedure 162, 164, 165
GetFTime procedure 145
GetGraphMode function 167
GetImage procedure 149, 165
GetIntVec procedure 146
GetLineSettings procedure 161, 163, 165
GetMaxColor function 167
GetMaxMode function 167
GetMaxX function 167
GetMaxY function 167
GetMem procedure 126, 217
GetModeName function 167
GetModeRange procedure 159, 165
GetPalette procedure 163, 165
GetPaletteSize function 167
GetPixel function 154, 167
GetTextSettings procedure 153, 161, 163, 166
GetTime procedure 145
GetVerify procedure 147
GetViewSettings procedure 164, 166
GetX function 167
GetY function 167
goto statements 84
GotoXY procedure 206
GRAPH.TPU 150
Graph unit 149, 182
  bit images in 153
colors 154
constants 159
error handling 154
figures and styles in 153
functions 167
heap management routines 156
paging 154
procedures 165
sample program 155, 156
text in 152
types 163
variables 165
viewports in 153
GraphDefaults procedure 166
GraphDriver variable  
  IBM 8514 and 150
GraphErrorMsg function 167
GraphFreeMem procedure 156
GraphFreeMemPtr variable 165
GraphGetMem procedure 156
GraphGetMemPtr variable 165
graphics
  CloseGraph 149
current pointer in 152
drivers 149
figures and styles 153
InitGraph in 149
sample program 155, 156
GraphResult function 154, 160, 167
H
Halt procedure 125, 246
handles
  file 222
hardware, interrupts 248
heap error function 236
heap management 211
  allocating 211, 212, 215, 217
deallocating 212
dynamic memory allocation 137
  fragmenting 211
  free list 215
  map 210
  pointers 210
  routines 156
  sizes 272
HeapEnd variable 137
HeapError variable 137, 217
HeapOrg variable 137, 211, 212
HeapPtr variable 137, 211
hexadecimal constants 9
Hi function 129, 255
high
  resolution graphics 150
  HighVideo procedure 206
host type 25
directives 316
machine code 314
statements 314
inline assembler
asm statement 282
assembler directive 303
comments 282, 283
constants 291-293
numeric 291
string 292-293
untyped 297
directives 281, 287-289
assembler
external versus 304
eexpressions 290-303
classes 297-298
elements of 291-297
immediate values 297
operators 300-303
Pascal expressions versus 290
registers 297
types 298-300
labels 283-285
memory references 297
opcodes
instruction 285-287
sizing 286-287
prefix 285
operands 289-290
operator precedence 300
procedures and functions 303-306
registers 293
use 282
relocation 298
reserved words 289
separators 282
symbols 294-297
invalid 294
scope access 296
special 294
syntax 283
InOut function 253
InOutRes variable 138
input
files 129, 138
Input standard file 138
Insert procedure 127
InsLine procedure 206
InstallUserDriver function 167
InstallUserFont function 167
instances
dynamic objects 37
object 36
INT 24 handler 138
Int function 127
Integer data type 23, 218
interfaces section (program) 115, 121, 244, 308
interfacing Turbo Pascal with Turbo Assembler 308
internal data formats 218
interrupt
directives 97
handlers 248
units and 185
handling routines 138, 248
service routines (ISRs) 248
support procedures 146
vectors 138, 139
Intr procedure 146
registers and 143
invalid typecasting errors 329
IResult function 130, 138
IP flag 248
ISRs (interrupt service routines) 248

K
Keep procedure 146
key codes 350
keyboard
scan codes 351
status 201
KeyPressed function 201, 205

L
$L$ compiler directive 233, 267, 271, 307
labels 8
declaration part 16
local 284
LastMode variable 204
late binding 36
Length function 128, 255
line
input, Crt 201
Line procedure 166
LineRel procedure 166
LineTo procedure 166
linking
assembly language 307
object files 271
smart 258
Ln function 127
Lo function 129, 255
Local labels 283
local symbol information switch 267
Local Symbols
command 267
option 267
logical operators 70
Longint data type 23
LowVideo procedure 206
LPT devices 135

M
SM compiler directive 48, 211, 272
machine code 314
macros, inline 316
Mark procedure 126, 212
math coprocessor See numeric coprocessor
MaxAvail function 126
Mem array 223
MemAvail function 126
MemL array 223
memory
allocation 182
compiler directive 272
DirectVideo and 204
error messages 321
inline assembler references 297
map 210
size 272
Memory Sizes command 272
MemW array 223
methods
activation, qualified 83
assembly language 233
calling
as functions or procedures 77, 83
conventions 83, 230
declaring 102
defined 32
designators 38, 52
  @ (address operator) with 76
external 233
identifiers, qualified 35
  accessing object fields 52
  in method calls 38, 83
  in method declarations 102
implementation 35, 102
overridden, calling 84
overriding inherited 36
parameters
  Self 83, 84, 102
    defined 230
  type compatibility 107
qualified activation 83
static 36
  calling 38
virtual 36
  calling 38, 83, 229
  error checking 227
methods, declaring 35
MkDir procedure 131
mod operator 70
.MODEL directive
  setting up calling conventions with 309
modular programming 114
monochrome adapters, CheckSnow and 203
Move procedure 128
MoveRel procedure 166
MoveTo procedure 166
MsDos procedure 146
MSDOS symbol 275

N
$N compiler directive 27, 190, 195, 268
near
  directives 97
NEAR calls 243
nesting
  files 271
  procedures and functions 244
network file access, read-only 133
New procedure 40, 126, 211, 217
  extended syntax 227
    constructor passed as parameter 103, 230,
    231
  used as function 232
nil 40, 53
NormVideo procedure 206
NoSound procedure 206
not operator 71, 154
NUL device 136
null strings 10, 28
numbers, counting 9, 218
numeric coprocessor
  compiler switch 268
  emulating 27, 125, 190
    assembly language and 197
  evaluation stack 193
  floating-point 27
  mode 334
  numeric processing option 27
  using 189-197
Numeric Processing command 268
O
$O compiler directive 180, 268, 272
  non-overlay units and 185
.OBJ files 307
  linking with 271
object
  directories, compiler directive 271
  files 307
  linking with 271
Object Directories command 271
objects
  ancestor 32
  constructors
    declaring 102
    defined 103
    error recovery 236
    implementation 102
  inherited 36
  virtual methods and 37, 103
  VMTP and 225, 228
descendant 32
destructors 103
  declaring 102
  defined 103
  implementation 102
domain 32
dynamic
  instances 37
with OvrSetBuf 211
returning 178
setting 178
cautions 185
code generation, compiler switch 268
debugging 185
files
loading into EMS 181
in .EXE files 187
load operations, customizing 175
loading
into EMS 177
into memory 169
manager 125
implementing 137
initializing 177, 181, 184
Overlays Allowed
command 268
option 268
overridden methods, calling 84
overriding inherited methods 36
.OVR files 169
OvrClearBuf procedure 178
OvrCodeList variable 137
OvrDebugPtr variable 137
OvrDosHandle variable 137
OvrEmsHandle variable 137
Ovr FileMode variable 174
OvrGetBuf function 178
OvrGetRetry function 173, 179
OvrHeapEnd variable 137
OvrHeapOrg variable 137
OvrHeapPtr variable 137
OvrHeapSize variable 137
OvrInit procedure 177
OvrInitEMS procedure 177, 182
OvrLoadCount variable 174
OvrLoadList variable 137
OvrReadBuf variable 175
OvrResult variable 174
OvrSetBuf procedure 137, 178, 182
increasing size of overlay buffer with 211
OvrSetRetry procedure 173, 179
OvrTrapCount variable 174

P
Pack procedure 126
packed (reserved word) 28
PackTime procedure 145
DateTime and 144
palette
manipulation routines 151
ParamCount function 129
parameter directives See compiler, directives, parameter
parameters
actual 83
call-command-line 129
floating-point 242
formal 83, 105
passing 83, 241
procedural-type 111
Self 83, 84, 102
defined 230
type compatibility 107
value 106, 242
variable 106
untyped 107
VMT 230
ParamStr function 129
Pi function 127
PieSlice procedure 166
point (§) symbol 39, 40, 53
pointer and address functions 128
pointer-type constants 62
pointers
assignment compatibility 38
comparing 74
to objects 37
types 39, 221
values 53
variables 53, 75
polymorphism
object instance assignment 82
parameter type compatibility 107
pointer assignment 38
Port array 253
PortW array 253
Pos function 128
pound (#) character 11
precedence of operators 65, 69
Pred function 23, 127, 255
PreFixSeg variable 209
PrefixSeg variable 209
printer devices 135
private
   fields
      scope 35
   methods
      scope 35
PRN 135
probationary area, overlay buffer 173
PROC directive, defining parameters with 309
procedural
types 40, 108, 108-112
   declarations 40
   in expressions 79
   in statements 79
   variable declaration 108
   variable typecasts and 54
values, assigning 108
variables 108
   restrictions 109
   using standard procedures and functions with 109
procedural-type constants 63
procedural-type parameters 111
procedure and function declaration part
   (program) 17
procedure call models 97
procedures 95
   body 96
   declarations 95
Dispose
   extended syntax 227, 231
      constructor passed as parameter 103, 230, 231
   dynamic allocation 125
entry/exit code
   inline assembler 304
Exit 125
Fail 237
file-handling 146
Graph unit 165
Halt 125
headings 96
inline assembler 303-306
methods denoting
   calls to 83
   nesting 110, 244

New
   extended syntax 227
      constructor passed as parameter 103, 230, 231
      used as function 232
   ordinal 127
   OvrSetRetry 173, 179
parameters
   inline assembler 303
   pointers, calling overlaid routines 185
stack frame
   inline assembler 304
   standard 125
   statements 83
   string 127
process-handling routines 146, 147
Program Segment Prefix (PSP) 138, 209
programs
   headings 113
   lines 13
   parameters 113
   syntax 113
   termination 246
Ptr function 40, 128, 255
PUBLIC 307
   definition errors 327
PutImage procedure 149, 154, 163, 166
PutPixel procedure 154, 166
Q
qualified
   activation 83, 84
   identifiers 7, 18
   method identifiers 35
      accessing object fields 52
      in method calls 38, 83
      in method declarations 102
R
   $R compiler directive 269
      virtual method checking 227
Random function 129, 138
random number generator 138
Randomize procedure 128
RandSeed function 138
Range Checking
   command 269
   option 269
range checking
   compile time 257
   compiler switch 269
read-only file access 133
Read procedure
   text files 130, 132
ReadKey function 201, 205
Readln procedure 132
real
   numbers 26, 189, 219
   types 26
record-type constants 60
records 30, 52, 60, 222
   fields 52
Rectangle procedure 166
redeclaration 17, 47
redirection 199
reentrant code 248, 249
referencing errors 335
register-saving conventions 246
RegisterBGldriver function 150, 157, 167, 185
RegisterBCIfont function 157, 167, 185
registers
   AX 243, 316
   BX 243, 248
   BP 246, 248, 315
   overlays and 185
   BS 243, 248
   CS 248
   CX 248
   DI 248
   DS 246, 248, 308, 315
   DX 243, 248
   ES 248
   inline assembler 297
   inline assembler use 282
   SI 248
   SP 138, 246
   SS 246
   using 243, 246, 248, 315
Registers type 143
relational operators 72
relaxed string parameter checking 270
Release procedure 126, 212
relocatable reference errors 328
relocation
   inline assembler 298
Rename procedure 131
repeat statements 88
repetitive statements 88
reserved words 6, 7
   external 233
   inline assembler 289
   virtual 36
   with 52
Reset procedure 129, 131, 138
RestoreCrtMode procedure 150, 166
result codes 177
results
   functions
      discarding 77, 270
Rewrite procedure 129, 131
RmDir procedure 131
Round function 126, 255
round-off errors, minimizing 192
routines, operating system 146, 147
rules, scope 17
run-time
   errors 246, 340
   fatal 344
   support routines 125
S
$S compiler directive 48, 269
SaveInitXX variables 138
SaveInt24 139
scale factor 10
scan codes, keyboard 351
scope
   object 35
   of declaration 17
screen
   mode control 199
   output operations 199
SearchRec type 144
Sector procedure 166
Seek procedure 130, 131
SeekEof function 133
SeekEoln function 133
Self parameter 83, 84, 102
   defined 230
Seq function 128
<table>
<thead>
<tr>
<th>set-type constants 62</th>
<th>SizeOf function 129, 228</th>
</tr>
</thead>
<tbody>
<tr>
<td>set types 38, 221</td>
<td>smart linking 258</td>
</tr>
<tr>
<td>SetActivePage procedure 166</td>
<td>snow-checking 203</td>
</tr>
<tr>
<td>SetAllPalette procedure 160, 163, 166</td>
<td>software</td>
</tr>
<tr>
<td>SetAspectRatio procedure 166</td>
<td>floating-point 27</td>
</tr>
<tr>
<td>SetBkColor procedure 166</td>
<td>interrupts 248</td>
</tr>
<tr>
<td>SetCBreak procedure 147</td>
<td>numeric processing See numeric coprocessor,</td>
</tr>
<tr>
<td>SetColor procedure 166</td>
<td>emulating</td>
</tr>
<tr>
<td>SetDate procedure 145</td>
<td>sound operations</td>
</tr>
<tr>
<td>SetFAttr procedure 142, 146</td>
<td>NoSound 206</td>
</tr>
<tr>
<td>SetFillPattern procedure 153, 162, 164, 166</td>
<td>Sound 206</td>
</tr>
<tr>
<td>SetFillStyle procedure 153, 162, 166</td>
<td>Sound procedure 206</td>
</tr>
<tr>
<td>SetFTime procedure 145</td>
<td>source debugging compiler switch 264</td>
</tr>
<tr>
<td>SetGraphBufSize procedure 157, 166</td>
<td>SP register 138, 246, 315</td>
</tr>
<tr>
<td>SetGraphMode procedure 150, 166</td>
<td>space characters 5</td>
</tr>
<tr>
<td>SetIntVec procedure 146</td>
<td>SHR operator 70</td>
</tr>
<tr>
<td>SetLineStyle procedure 153, 161, 166</td>
<td>stack</td>
</tr>
<tr>
<td>SetPalette procedure 160, 166</td>
<td>8087 193</td>
</tr>
<tr>
<td>SetRGBPalette procedure 151, 161, 166</td>
<td>checking switch directive 269</td>
</tr>
<tr>
<td>IBM 8514 and 151</td>
<td>overflow 48</td>
</tr>
<tr>
<td>sets</td>
<td>switch directive 269</td>
</tr>
<tr>
<td>comparing 74</td>
<td>segment 48</td>
</tr>
<tr>
<td>constructors 66, 77</td>
<td>size 272</td>
</tr>
<tr>
<td>membership 74</td>
<td>Stack Checking</td>
</tr>
<tr>
<td>operators 72</td>
<td>command 269</td>
</tr>
<tr>
<td>SetTextBuf procedure 132</td>
<td>option 269</td>
</tr>
<tr>
<td>SetTextStyle procedure 153, 161, 166</td>
<td>stack frame</td>
</tr>
<tr>
<td>SetTime procedure 145</td>
<td>inline assembler use of 304</td>
</tr>
<tr>
<td>SetUserCharSize procedure 153, 166</td>
<td>standard</td>
</tr>
<tr>
<td>SetVerify procedure 147</td>
<td>functions, constant expressions and 13</td>
</tr>
<tr>
<td>SetViewPort procedure 149, 162, 167</td>
<td>units See units, standard</td>
</tr>
<tr>
<td>SetVisualPage procedure 167</td>
<td>statement part (program) 17</td>
</tr>
<tr>
<td>SetWriteMode procedure 163, 167</td>
<td>statements 81</td>
</tr>
<tr>
<td>shl operator 70</td>
<td>assignment 82</td>
</tr>
<tr>
<td>short-circuit Boolean evaluation 256, 263</td>
<td>case 87</td>
</tr>
<tr>
<td>Shortint data type 23</td>
<td>compound 85</td>
</tr>
<tr>
<td>shr operator 70</td>
<td>conditional 86</td>
</tr>
<tr>
<td>SI register 248</td>
<td>for 90</td>
</tr>
<tr>
<td>signed number (syntax) 10</td>
<td>goto 84</td>
</tr>
<tr>
<td>significand 219</td>
<td>if 86</td>
</tr>
<tr>
<td>simple</td>
<td>procedure 83</td>
</tr>
<tr>
<td>statements 81</td>
<td>repeat 88</td>
</tr>
<tr>
<td>types 22</td>
<td></td>
</tr>
<tr>
<td>simple-type constants 58</td>
<td></td>
</tr>
<tr>
<td>Sin function 127</td>
<td></td>
</tr>
<tr>
<td>Single floating-point type 191, 219</td>
<td></td>
</tr>
</tbody>
</table>
repetitive 88
simple 81
structured 85
uses 114
while 89
with 52, 92
static methods 36
calling 38
storing overlaid code 211
Str procedure 127
Strict Var-Strings
   command 270
   option 270
string-type constants 59
strings 59
   character 10
   comparing 73, 74
   concatenation 72
   functions 128
   handling 125
   length byte 221
   maximum length 221
   null 28
   operators 72
   procedures 127
   relaxed parameter checking 270
   types 27, 221
   variables 51
stroked fonts 149, 152
structured
   statements 85
   types 28
      declaring 110
structured-type constants 59
subrange type 25
Succ function 23, 127, 255
Swap function 129, 255
SwapVectors procedure 139, 146
switch compiler directives 261, 262-270
symbols 5
   conditional 274
   CPU 275
   inline assembler 294-297
   local information 267
   scope access
      inline assembler 296
syntax
   extended 270
   inline assembler 283
syntax diagrams, reading 5
System unit 114, 125, 195
   floating-point routines 190
   interrupt vectors and 138
   trapping critical errors 138
   variables in 125-139
T
tag field (of records) 31
terminating a program 246
terms (syntax) 67
Test8087 variable 138
text 152
   color constants 202
   files 131
      devices 136
      drivers 251
      records 223
TextAttr variable 204
TextBackground procedure 202, 206
TextColor procedure 202, 206
TextHeight function 167
TextMode procedure 202, 206
TextRec records 143, 222, 251
TextWidth function 167
tokens 5
transfer functions 126
trapping
   critical errors 138
   critical errors, System unit and 138
   I/O errors 266
   interrupts 248
Trunc function 126, 255
Truncate procedure 131
Turbo Assembler
   8087 emulation and 197
   example program 313
Turbo Pascal
   Editor Toolbox 263
type checking, strings and 270
typecasting, invalid 329
typed
   constants 57
      object type 61
files 138, 222
TypeOf function 229
types 21
array 29, 221
Boolean 24, 218
Byte 23
Char 24, 218
common 24
compatibility 42
declaration 21
  part 16, 44
definition, constant expressions and 26
enumerated 25, 218
file 39
floating-point 191, 219
  Comp 26, 191, 220
  comparing values of 193
Double 26, 191, 220
Extended 26, 191, 220
Single 26, 191, 219
Graph unit 163
host 25
identity 41
Integer 23, 218
Longint 23
mismatches, error messages 324
object 32
  declaring 33
ordinal 22
Pointer 39, 221
procedural 40, 79, 108
real 26
real numbers 219
record 30, 222
set 38, 221
Shortint 23
simple 22
string 27, 221
structured 28
subrange 25
Word 23

heading 115
identifiers 7
indirect references 117
initialization code 184
non-overlay 185
scope of 18
standard
  Crt 132, 136, 199
  Dos 141
  Graph 149
  Overlay 170
  overlays and 171
  System 125
  system 114
syntax 114
version
  mismatch errors 330
  number 118
Unpack procedure 126
UnpackTime procedure 145
  DateTime and 144
unsigned
  constant 66
  integer 9
  number 10
  real 10
untyped
  files 133, 138, 222
  var parameters 107
UpCase function 129
uses statement 114

V
$V compiler directive 270
Val procedure 127
value
  parameters 106, 242
typecasts 78
var
  declaration section 259
  parameters 106, 242
  untyped 107
string checking, compiler switch 270
variables 47
  absolute 49
  arrays 51
  CheckBreak 203

U
$UNDEF compiler directive 274, 276
units
  8087 coprocessor and 195
circular references 118
CheckEOF 203
CheckSnow 203
Crt 203
declaration part 16
declarations 47
defined by Overlay unit 173
DirectVideo 204
DosError 145
dynamic 39, 53, 211
FileMode 133
global 48
Graph unit 165
I/O 129
initializing 57
LastMode 204
local 48
parameters 242
pointer 53, 75
procedural 108
restrictions 109
record 52
references 50
strings 51
TextAttr 204
typecasts 53, 54
WindMax 205
WindMin 205
variant part (syntax) 31
VERSO symbol 275
VGA
modes
emulated 150
video
memory 199
viewports 153
virtual (reserved word) 36
virtual method table 227
pointer 225
initialization 228
virtual methods 36
calling 38, 83, 229
error checking 227
VMT parameter 230

W
WhereX function 205
WhereY function 205
while statements (syntax) 89
WindMax variable 205
WindMin variable 205
Window procedure 200, 206
current coordinates 205
windows 200
with (reserved word)
statement 52
with statements 92
Word
data type 23
word
alignment
automatic 257
Word Align Data command 262
write
procedures 129
statements
8087 coprocessor and 194
BIOS 204
DirectVideo and 204
Write procedure 132
WriteLn procedure 132
8087 coprocessor and 194
DirectVideo and 204

X
$X compiler directive 270
xor operator 71, 154