

Language Reference

for

Virtual Pascal

v2.1

containing

[Lexical elements](#)

[Program structure](#)

[Compilation process](#)

[Untyped constants](#)

[Data types](#)

[Variables and typed constants](#)

[Expressions](#)

[Statements](#)

[Procedures and functions](#)

[Dynamic link libraries](#)

[Linker Module Definition File Reference](#)

[Compiler directives](#)

[Naming conventions](#)

[Error messages](#)

[Index](#)

Copyright © 1996-1999 by fPrint (UK) Ltd. All rights reserved.

All brand names and product names are trademarks or registered trademarks of their respective holders.

fPrint (UK) Ltd
Riverview House
Beavor Lane
London W6 9AR
United Kingdom

Produced in the United Kingdom.

CHAPTER 1	9
LEXICAL ELEMENTS	9
<i>Source file</i>	9
<i>Whitespace</i>	9
<i>Comments</i>	10
<i>Compiler directives</i>	11
<i>Token characters</i>	11
<i>Reserved words</i>	12
<i>Standard directives</i>	12
<i>Identifiers</i>	13
<i>Literals</i>	13
Integer numbers.....	13
Floating point numbers.....	14
Character strings.....	14
<i>Labels</i>	15
<i>Symbols</i>	15
<i>Conditional compilation</i>	16
CHAPTER 2	18
PROGRAM STRUCTURE	18
<i>Program</i>	18
<i>Unit</i>	18
The interface part.....	19
The implementation part.....	19
The initialisation and finalisation parts.....	20
<i>Dynamic link library</i>	20
<i>Uses clause</i>	21
<i>Block</i>	21
<i>Scope</i>	22
Record scope (i).....	23
Object and class scope.....	23
Block scope (ii).....	23
Unit scope (iii).....	23
CHAPTER 3	24
COMPILATION PROCESS.....	24
<i>The linking process</i>	25
<i>Creating an import library</i>	26
<i>Compiling and binding resources</i>	26
CHAPTER 4	27
UNTYPED CONSTANTS	27
CHAPTER 5	28
DATA TYPES.....	28
<i>Ordinal types</i>	28
<i>Integer types</i>	29
Basic integer types.....	29
Discussion: Use32, Use16 and bits.....	30
<i>Character types</i>	31
<i>Enumerated types</i>	31
<i>Boolean types</i>	31
<i>Subrange types</i>	32

4 Contents

<i>Floating point types</i>	32
<i>Pointer types</i>	34
<i>Structured types</i>	37
<i>String types</i>	37
Short strings.....	37
Long strings.....	38
<i>Array types</i>	40
<i>Record types</i>	41
<i>Object and class types</i>	43
Object type declarations	43
Class type declarations	45
Object and class components.....	47
Inheritance	48
Construction of object and class types.....	48
Class references	50
Class methods.....	50
Compatibility rules	51
Components and scope.....	51
Component visibility	52
Static methods	53
Virtual methods	53
Dynamic methods.....	55
Abstract methods.....	55
Message handler declarations	56
Message handler implementations.....	56
Message dispatching.....	57
Method activations	57
Properties.....	58
Access specifiers	60
Array properties.....	60
Index specifiers.....	61
Storage specifiers	62
Property overrides	63
Class-reference types.....	63
Summary of the two object models	64
<i>Set types</i>	64
<i>File types</i>	65
Procedural types	67
Procedural type compatibility.....	68
<i>Type identity</i>	69
<i>Type compatibility</i>	70
<i>Assignment compatibility</i>	70
CHAPTER 6	72
VARIABLES AND TYPED CONSTANTS.....	72
<i>Variable and typed constant declarations</i>	72
<i>Simple-typed constants</i>	73
<i>String-type constants</i>	73
<i>Structured-type constants</i>	74
Array-type constants.....	74
Record-type constants	75
Object-type constants	75
<i>Address constants</i>	76
<i>Procedural type constants</i>	76
<i>Memory allocation</i>	76
<i>Variable references</i>	77

Qualifiers	78
Indices	78
Record field and object component designators	79
Pointers and dynamic variables	79
Variable typecasts	79
MEM arrays.....	80
CHAPTER 7	81
EXPRESSIONS	81
<i>Expression syntax</i>	81
<i>Function calls</i>	83
<i>Set constructors</i>	83
<i>Value typecasts</i>	84
<i>The @ operator</i>	84
<i>Operators</i>	85
Rules of precedence.....	85
Arithmetic operators	86
Bitwise logical operators	86
Boolean logical operators	87
String operator.....	88
PChar operators.....	88
Set operators	89
Relational operators.....	89
<i>Class operators</i>	91
The is operator.....	91
The as operator	91
<i>Port arrays</i>	92
CHAPTER 8	93
STATEMENTS.....	93
Assignment statements	93
Procedure statements	94
Goto statements	94
Compound statements.....	95
<i>Conditional statements</i>	95
if statements.....	95
Case statements	96
<i>Repetitive statements</i>	96
Repeat statement.....	97
While statement	97
For statements.....	98
<i>With statements</i>	99
<i>Exception statements</i>	100
The raise statement	101
The try...except statement.....	102
The try...finally statement.....	103
<i>Assembler statements</i>	104
Labels	105
Prefixes	105
Instructions	105
Pseudo instructions.....	106
Asm directives	107
Operands.....	107
CHAPTER 9	111
PROCEDURES AND FUNCTIONS	111

<i>Procedure declarations</i>	111
<i>Function declarations</i>	112
<i>Near and far declarations</i>	113
<i>Export declarations</i>	113
<i>Forward declarations</i>	113
<i>Calling conventions</i>	114
<i>External declarations</i>	115
<i>Assembler declarations</i>	115
<i>Inline declarations</i>	116
<i>Method declarations</i>	116
Constructors and destructors.....	117
Class methods.....	120
<i>Parameters</i>	120
Value parameters.....	121
Constant parameters.....	121
Variable parameters.....	122
Untyped parameters.....	122
Open parameters.....	123
<i>Open array constructors</i>	124
<i>Type variant open array parameters</i>	124
CHAPTER 10	127
DYNAMIC LINK LIBRARIES.....	127
<i>What is a DLL?</i>	127
The traditional method using export.....	127
Creating DLLs on a per unit basis.....	128
<i>Importing symbols from a DLL</i>	128
Static import.....	128
Dynamic import.....	129
<i>Exporting symbols from a DLL</i>	130
The traditional method.....	130
The export directive.....	130
Using module definition files.....	131
Exporting the entire interface part (OS/2 only).....	131
<i>Types of DLLs</i>	131
Subsystems.....	131
Subroutine Libraries.....	132
Important notes.....	133
Quick DLL examples.....	133
<i>DLLs and unit initialisation code</i>	135
APPENDIX A	136
LINKER MODULE DEFINITION FILE REFERENCE.....	136
<i>Segment attributes</i>	138
<i>CODE</i>	139
<i>DATA</i>	140
<i>DESCRIPTION</i>	141
<i>EXETYPE</i>	141
<i>EXPORTS</i>	141
<i>IMPORTS</i>	143
<i>LIBRARY</i>	144
<i>NAME</i>	145
<i>OLD</i>	145
<i>SEGMENTS</i>	146
<i>STACKSIZE</i>	147

<i>STUB</i>	147
APPENDIX B	149
COMPILER DIRECTIVES.....	149
<i>\$A, &AlignData</i>	149
<i>&AlignCode</i>	150
<i>&AlignRec</i>	150
<i>&Alters</i>	150
<i>&Asm</i>	150
<i>\$B</i>	151
<i>&Cdecl</i>	151
<i>&Comments</i>	151
<i>\$D</i>	152
<i>\$DEFINE</i>	152
<i>&Delphi</i>	152
<i>&Dynamic</i>	153
<i>\$ELSE</i>	153
<i>\$ENDIF</i>	153
<i>&Export</i>	154
<i>&Far16</i>	154
<i>&Frame</i>	154
<i>&G3, &G4, &G5</i>	155
<i>\$H</i>	155
<i>\$I</i>	155
<i>\$I</i>	156
<i>\$IFDEF</i>	156
<i>\$IFNDEF</i>	156
<i>\$IFOPT</i>	157
<i>\$J</i>	157
<i>\$L</i>	157
<i>\$L</i>	157
<i>&Linker</i>	158
<i>&LocInfo</i>	158
<i>\$M</i>	158
<i>\$M</i>	159
<i>&Open32</i>	159
<i>&Optimise, &Optimize</i>	159
<i>&OrgName</i>	160
<i>\$P</i>	160
<i>&PmType</i>	160
<i>&PureInt</i>	161
<i>\$Q</i>	161
<i>\$R</i>	162
<i>\$S</i>	162
<i>&Saves</i>	162
<i>&SmartLink</i>	163
<i>&Speed</i>	163
<i>\$StdCall</i>	163
<i>\$T</i>	163
<i>\$UNDEF</i>	164
<i>&Use32</i>	164
<i>&Uses</i>	164
<i>\$V</i>	164

8 Contents

\$W.....	165
\$X.....	165
\$Z.....	165
&Zd.....	165
APPENDIX C.....	166
THE OPEN DEBUG API (ODAPI).....	166
<i>The debugger DLL</i>	166
<i>The ODAPI System Interface</i>	166
<i>The ODAPI IDE Interface</i>	167
<i>ODAPI Types</i>	168
<i>ODAPI Examples</i>	169
APPENDIX D.....	170
ERROR MESSAGES.....	170
<i>Compiler error messages</i>	170
<i>Run-time error messages</i>	184
APPENDIX E.....	187
NAMING CONVENTIONS.....	187
INDEX.....	188

CHAPTER 1

Lexical elements

This chapter describes the lexical level of the Virtual Pascal Language. From the lexical point of view, a Pascal program consists of a set of small units, known as tokens. They include identifiers, reserved words, constants, literals, comments, etc. Tokens can be grouped together to form more complex structures such as expressions and statements. Tokens are easily identified in the Integrated Development Environment (IDE) editor when syntax highlighting is on - they are displayed in different colours.

Source file

The source code of a program is a simple ASCII text file, which can be created by any text file editor, such as the IDE editor, the E editor supplied with OS/2 or Notepad in Windows. The Virtual Pascal Compiler recognises MS-DOS style text files, where Carriage Return (ASCII 13) and Line Feed (ASCII 10) characters are used as the line delimiter. UNIX style text files, where a single Line Feed (ASCII 10) is used as the line delimiter, are not supported. These must be converted before compilation, e.g. by means of the IDE editor. The length of a line may not exceed 255 characters; using files with lines longer than this limit produces a compiler error. However, if the program is to be compatible with other Pascal compilers (Borland Pascal, Delphi), it is recommended not to create lines longer than 126 characters, which is the maximum line length supported by these.

The source text can be divided into several files. In this case one of the files is the main or primary file and the others can be included through the compiler directive `{$I filename}`. See page 156 for more information about the `$I` compiler directive.

Whitespace

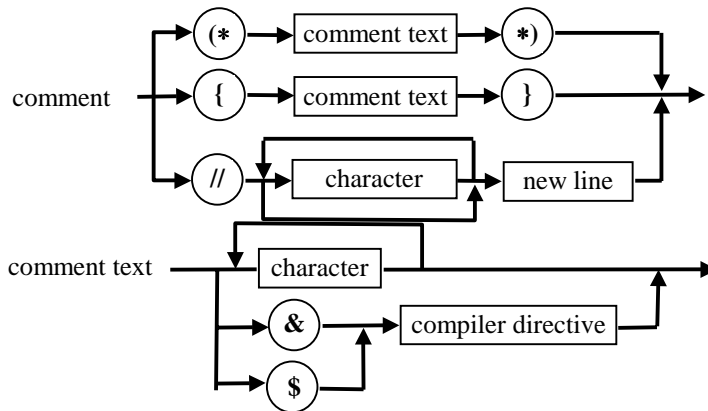
Whitespace is a collective name given to spaces (blanks) and control characters such as horizontal and vertical tabs and new line characters. Whitespaces are used to separate tokens and the compiler discards them. The ASCII characters representing whitespaces can also occur within literal strings, in which case they are protected from the normal parsing process. Control ASCII characters (with codes 0 through 31) and the space character (ASCII 32) are treated as whitespace.

Whitespace control characters

dec	hex	ctrl	esc	dec	hex	ctrl	esc
00	00	^@	NUL	16	10	^P	BS
01	01	^A	SOH	17	11	^Q	HT
02	02	^B	STX	18	12	^R	LF
03	03	^C	ETX	19	13	^S	VT
04	04	^D	EOT	20	14	^T	FF
05	05	^E	ENQ	21	15	^U	CR
06	06	^F	ACK	22	16	^V	SO
07	07	^G	BEL	23	17	^W	SI
08	08	^H	BS	24	18	^X	CAN
09	09	^I	HT	25	19	^Y	EM
10	0A	^J	LF	26	1A	^Z	SUB
11	0B	^K	VT	27	1B	^[ESC
12	0C	^L	FF	28	1C	^\	FS
13	0D	^M	CR	29	1D	^]	GS
14	0E	^N	SO	30	1E	^^	RS
15	0F	^O	SI	31	1F	^_	US

Comments

Comments are pieces of text that are used by the programmers to annotate a program. The compiler ignores comments; they are stripped from the source file before parsing. Comments must adhere to the following syntax:



The following types of comments are recognised by Virtual Pascal:

- Any sequence of characters placed after the symbol pair (*, the comment terminates at the first occurrence of the pair *).
- Any sequence of characters placed after the { symbol, the comment terminates at the first occurrence of the } symbol.
- Any sequence of characters placed after the symbol pair //, the comment terminates at the end of the current line.
- Nested comments are allowed, provided they are of the different types. For example,

{ Use (\$D+*) to enable debug information }*

is a valid comment, while

{ Use {\$D+} to enable debug information }

is only valid in the **{&Comments+}** state, since the comment ends at the first } in the default **{&Comments-}** state.

Note, that only {} comments can be nested in the **{&Comments+}** state; (* *) comments can never be nested.

Compiler directives

Compiler directives are a special type of comment that provide additional information to the compiler and control the compilation process. Compiler directives begin with either {\$ or (*\$ and are followed by the name of the directive. To specify Virtual Pascal specific compiler directives it is possible to use an ampersand (&) instead of the dollar sign (\$). Such a directive is recognised by Virtual Pascal, but is ignored by other Pascal compilers such as Borland Pascal. Throughout the manual, all Borland Pascal directives supported by Virtual Pascal are used with a \$ sign, while all Virtual Pascal specific ones are listed with an & sign. For a complete description of Virtual Pascal compiler directives see the Appendix starting on page 149.

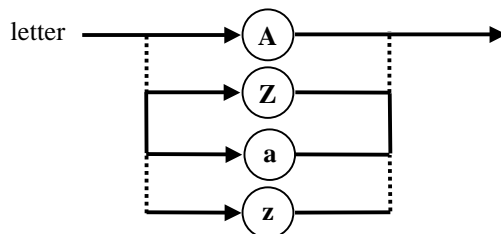
Note

There must be at least one whitespace or a comment between two tokens if both of them are reserved words, identifiers, labels or numbers.

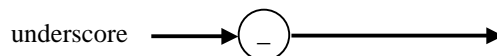
Token characters

Tokens of Virtual Pascal are comprised of characters from the following groups:

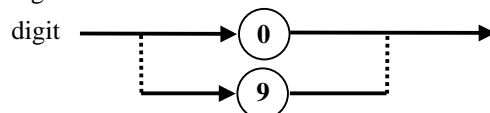
- Letters:



- Underscore:

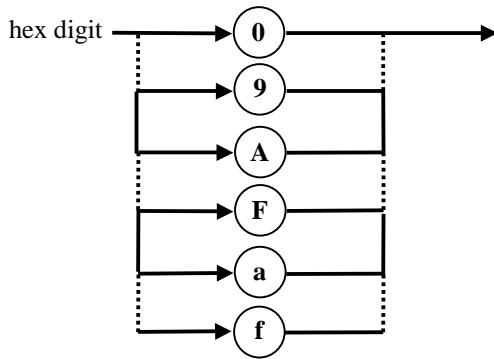


- Digits:



- Hexadecimal digits:

12 Lexical elements



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

- Symbol pairs:
<= >= := .. (* *) (. .)

The (. .) symbol pair is synonym to [and .) is equivalent to].

Reserved words

Reserved words are words reserved for special purposes and must not be used as identifier names. Reserved words are not case-sensitive, so **UNIT**, **Unit** and **unit** are equivalent. The following table lists all reserved words of Virtual Pascal.

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

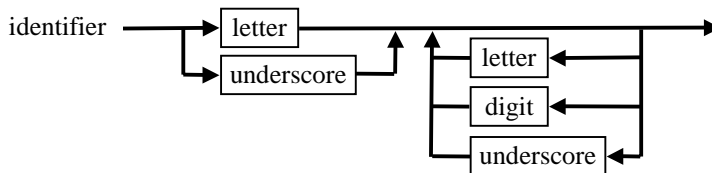
Standard directives

Unlike reserved words, standard directives are reserved only in certain contexts and they can be redefined. However, to avoid confusion, this is not recommended. Standard directives are highlighted as reserved words in the IDE editor. Standard directives are not case-sensitive. Here is the list of standard directives:

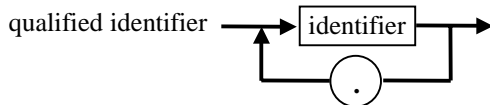
absolute	external	near	published
abstract	far	nodefault	read
assembler	far16	orgname	resident
at	forward	override	stdcall
cdecl	index	pascal	stored
default	inline	private	virtual
dynamic	message	protected	write
export	name	public	

Identifiers

Identifiers are arbitrary names of any length given to variables, procedures, functions, types, objects, etc. Only the first 63 characters of an identifier are significant. Identifiers are composed of letters, digits or underscores and the first character of an identifier must be a letter or an underscore.

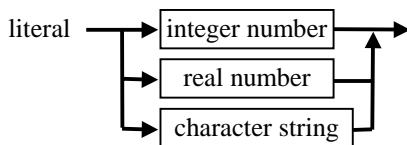


The case of the characters is not significant. Although identifier names are arbitrary (excluding the reserved words), it is not possible to use the same name for more than one identifier within the same scope (see the scope rules on page 22 for details). If identifiers have the same name and are located in different modules, it is possible to qualify the identifier with a module identifier in order to select a specific instance. In this case the module identifier should be followed by a period (.) and the identifier name for example *ModuleName.IdentifierName*. An identifier combined in this way is called a qualified identifier.



Literals

Literals are tokens representing numeric integer and floating point, character and string values.



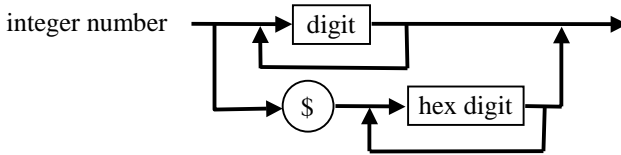
Integer numbers

Integer numbers can be decimal (base 10) or hexadecimal (base 16). In order to distinguish between decimal and hexadecimal numbers, hexadecimal ones must start with a dollar sign (\$).

Decimal values from 0 to 2147483647 are allowed. Constants exceeding this limit produce a compiler error. Decimal constants must use the digits 0 to 9, with leading 0s not being significant.

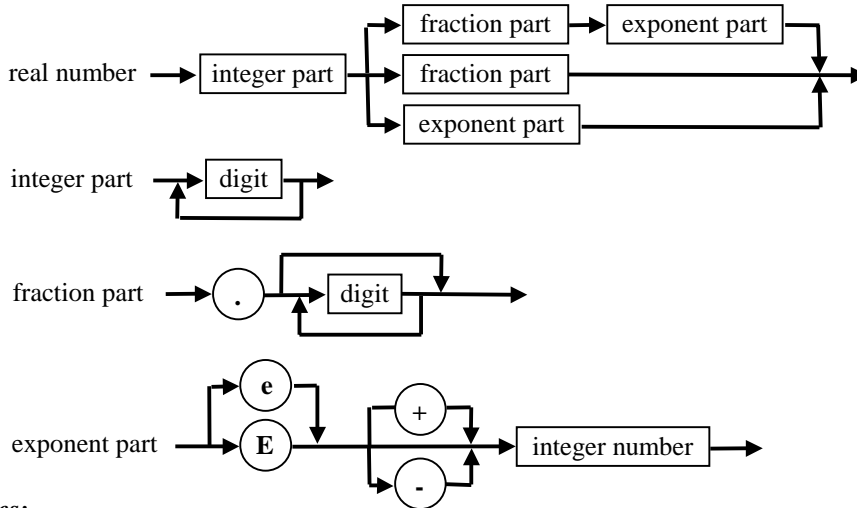
14 Lexical elements

Integer numbers starting with \$ are taken to be hexadecimal. Hexadecimal constants must be in the range \$0 to \$FFFFFFFF. Hexadecimal constants out of this allowed range produce a compiler error.



Floating point numbers

For floating point numbers, the conventional and scientific notations are used. A floating point constant consists of a decimal integer, a decimal point, a decimal fraction (optional), **e** or **E** and a signed integer exponent (optional). A floating point constant must begin with a decimal digit, so **.5** is not allowed. It allows to omit either the decimal point with the optional decimal fraction *or* the letter **e** (or **E**) and the signed integer exponent, but not both. If both of them are omitted, the number is taken to be an integer, which is why whole numbers outside the *Longint* range must contain a non-empty fraction part.

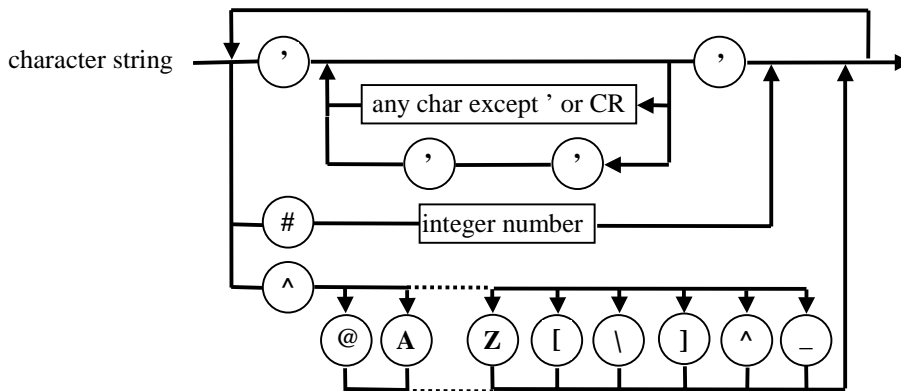


Examples:

Constant	Value	Value
0.	0	0
1.0	1	1
12.34e5	1.234×10^6	1234000
1e-2	1×10^{-2}	0.01
2e3	2×10^3	2000

Character strings

Character strings form a special category of literals used to handle fixed sequences of characters. Character strings are a sequence of zero or more characters enclosed in single quotes.



A string constant that has no characters between the quotes is an *empty* string. It is possible to use any ASCII characters (including whitespace) between quotes, but string constants must totally reside in one source line. In order to include the single quote character within a string constant it should be repeated, for example:

'I'm busy!'

Virtual Pascal allows control characters to be embedded in string constants. The # character followed by an unsigned integer between 0 and 255 denotes a character with the corresponding ASCII code.

The caret character (^) followed by a letter **A** to **Z** or **a** to **z**, or the characters @ [\] ^ denotes the corresponding control character. For example, **^G** denotes the bell ASCII character (#7).

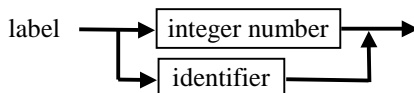
If several control characters are part of a string constant, there must be no whitespace or comment between them.

Example:

****Error** Can't open file QQ.TXT'#\$0D#\$0A^G*

Labels

Labels are special types of tokens that are used to mark statements. Control can be transferred control to a statement marked by a label by means of a **goto** statement. A Label can be represented either as an integer number in the range 0 to 9999 or an identifier.



Symbols

The following symbols and symbol pairs denote the arithmetic, relational and pointer operators:

16 Lexical elements

Symbols	Operation
+	addition, concatenation or union
-	subtraction, difference
*	multiplication, interaction
/	division
=	equal
<	less
>	greater
<=	less or equal
>=	greater or equal
@	pointer operator (taking an address)

The brackets [and] or their synonyms (. and .) indicate single and multidimensional array subscripts. They are also used in the set and open array constructors. The parentheses (and) group expressions, indicate function or procedure parameters, enclose array, record and object constants, etc.

The comma (,) separates the arguments of a function or procedure call, unit names in the uses clause, variable identifiers of the same type in a variable declaration, elements of an array in array constants, expressions in a set constructor, etc.

The colon (:) separates a variable identifier and its type in a variable declaration, record or object names and initialisation expression in record or object constants, indicates labelled statement, etc.

The semicolon (;) is a statement and declaration separator. It is also used to separate initialisation parts of each field in record or object constants.

The period (.) is used to denote a field qualifier.

The caret (^) is used as a pointer qualifier to dereference a pointer.

The := symbol pair denotes the assignment operator.

The .. symbol pair separates the lower and upper bounds of a subrange.

Conditional compilation

Virtual Pascal supports conditional compilation. Conditional compilation is based on the evaluation of a certain condition depending on which one or another part of the code is compiled. There are two types of conditions: the first one checks whether a conditional symbol is defined or undefined, the second one checks whether a switch compiler directive is turned on or turned off. Conditional compilation is implemented by constructs which resemble Pascal **if** statements. The following compiler directives are used in the first case:

- **{IFDEF *Symbol*}** the condition is *True* if *Symbol* is defined
- **{IFNDEF *Symbol*}** the condition is *True* if *Symbol* is undefined

and these directives are used in the second case:

- **{IFOPT *SwitchDir*+}** the condition is *True* if the directive *SwitchDir* is enabled
- **{IFOPT *SwitchDir*-}** the condition is *True* if the directive *SwitchDir* is disabled.

The part of the code which should be compiled is defined through the following constructs:


```
{$IFxxx}  
...  
{$ENDIF}
```

The source text between these two directives will be compiled if the condition, specified in `{$IFxxx}` is *True*, otherwise it will be ignored.

```
{$IFxxx}  
...  
{$ELSE}  
...  
{$ENDIF}
```

If the condition, specified in `{$IFxxx}` is *True*, then the source text between `{$IFxxx}` and `{$ELSE}` will be compiled, otherwise the part of the source text between `{$ELSE}` and `{$ENDIF}` will be compiled.

Conditional compilation constructs can be nested as deeply as needed.

Conditional symbols for conditional compilation can be defined in two different ways:

- initial conditional defines, specified using the `/D` switch of the command-line compiler or in the Options|Compiler|Conditional defines input box within the IDE.
- in the text of the program, using the directive `{$DEFINE Symbol}` to define a conditional symbol and `{$UNDEF Symbol}` to undefine it.

Conditional symbols are constructed as Pascal identifiers, they must start with a letter or an underscore followed by any combination of letters, underscores or digits. They can be of any length, but only the first 63 letters are significant. However, conditional symbols have no relation to Pascal identifiers. They cannot be evaluated in a program just as any other entities from a program can not be used in conditional compilation constructs.

Virtual Pascal always defines the following conditional symbols:

- **CPU86** and **CPU386** indicates that the CPU belongs to the 80x86 processor family.
- **CPU87** indicates that the coprocessor is available. Even if there is no hardware coprocessor installed, it will be emulated by OS/2.
- **OS2** indicates that the target operating system is OS/2 and is set when the target platform is set to OS/2. This symbol can be used when writing portable code which will run under different operating systems.
- **WIN32** indicates that the target operating system is 32-bit Windows and is set when the target platform is set to Win32.
- **USE32** indicates that the compiler produces 32-bit object code. This can be used to check it when writing 16-bit/32-bit portable code, for example while writing assembler code.
- **VER21** indicates the version of the compiler. Future versions will have their own predefined conditional symbols.
- **VIRTUALPASCAL** indicates that you are using the Virtual Pascal compiler.

Program structure

The following chapters describe the syntax of the Virtual Pascal Language. The syntax of the programming language provides a set of rules, defining the legal order of tokens and how they can be combined to make up a program.

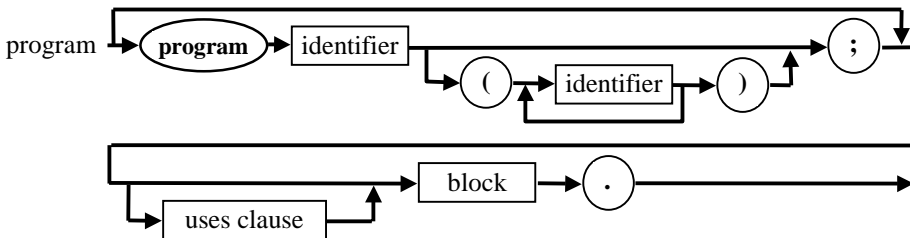
Standard Pascal features no separate compilation units and all the text of a program must be written in one module, called the program module. Later, modular programming appeared and separate compilation units, known as *units*, were introduced, allowing large applications to be divided into a set of logically related modules. Units are compiled separately. The final executable file is made by statically linking a program module by combining all the units that the program module uses.

Modern operating systems such as Windows 95, Windows NT and OS/2 allow the use of dynamic linking as well as static linking. Dynamically linked modules are combined into a dynamic-link library (DLL), and other applications are allowed to use part of its variables, procedures and functions. The actual linking of a program that uses several dynamic-link libraries takes place dynamically either during the loading of the program into memory or during execution of the program.

Virtual Pascal supports all of the above-mentioned types of compilation modules. In the description below, the term *module* is assumed to be either **program**, **unit** or **library**.

Program

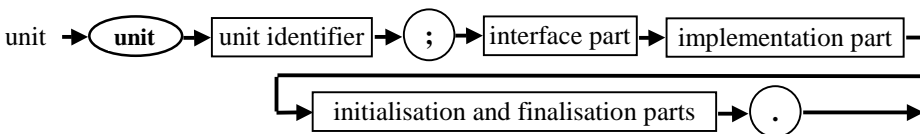
The syntax diagram below shows the main components of a Virtual Pascal program: a program heading, an optional uses clause and a block .



Unlike standard Pascal, the program header is optional and it is not required to specify the standard input and output files in the program header. These files are declared in the *System* unit and can always be used, irrespective of whether they are declared in the program header. If no program name is given, a default name of *Program* is assumed.

Unit

Units are separate compilation modules, allowing a large application to be divided into a set of logically related parts. The following diagram shows the main components of a unit:



The unit identifier must be unique and must reside in a Pascal source file with an identical name (not counting the .PAS extension). If it is not, the compiler will not be able to find the source and/or binary file during compile time. Although the length of the unit name may exceed 8 characters on HPFS, NTFS or VFAT partitions, to retain compatibility with other platforms and file systems, it is recommended not to declare units with names longer than 8 characters.

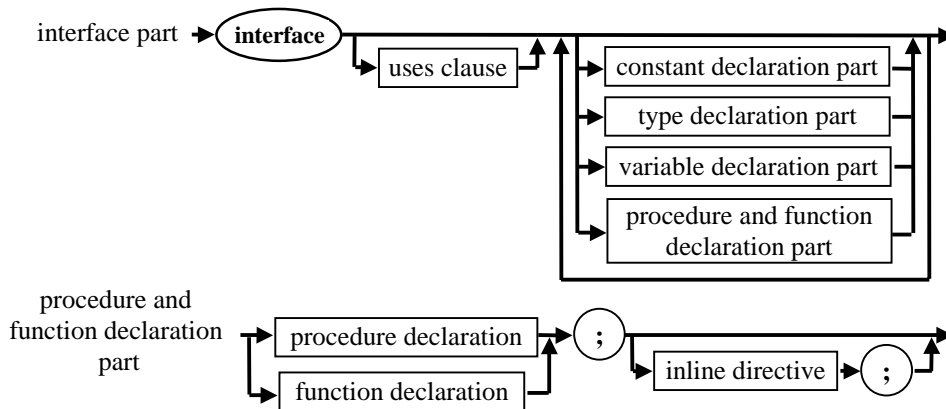
As with other identifiers, unit names may not contain spaces.

The interface part

Any constants, types, variables, procedures and functions that are declared in the interface part are accessible not only from within the unit, but also for any unit, program or library which uses the unit. For this reason, they are called *public*.

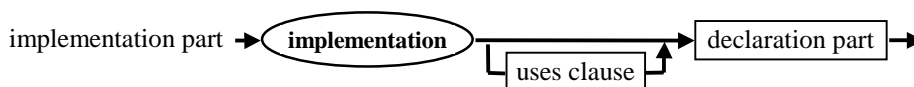
The interface part contains only procedure and function *headings* which are in fact forward declarations. The actual defining declaration with a statement part should be present anywhere in the implementation part of the unit. The exceptions to this rule are **inline** procedures and functions that contain their statement part just after the heading definition in the interface part of the unit (See page 116).

The following diagrams show the syntax of the interface part:



The implementation part

The implementation part contains the actual implementation of all procedures and functions declared in the interface part. Constants, types, variables, procedures and functions that are declared only in the implementation part are accessible only from the unit itself; they are not visible from a module using it and are called *private*. Here is the diagram for the implementation part:



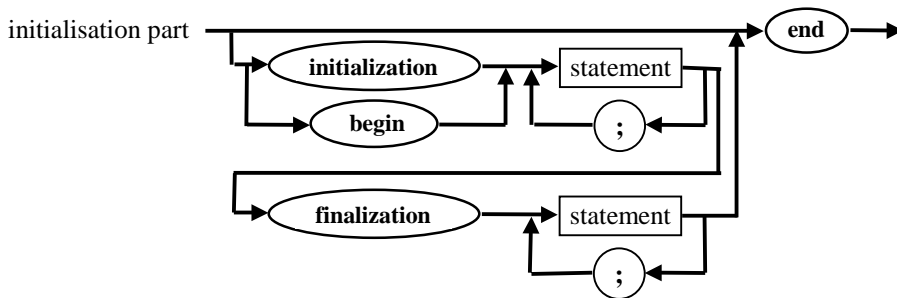
Note, that it is possible for a unit only to define types, constants and **inline** procedures and functions. In this case, the implementation part may be empty and consist just of the reserved word **implementation**. See also the section about pure interface units on page 161.

The initialisation and finalisation parts

The initialisation part contains statements that will be executed upon program start-up, before the statement part of the program receives control. The initialisation code of different units are executed in the order in which the units appear in the program's **uses** clause. The finalisation part contains statements that are executed when the program is shut down. The finalisation code of different units is executed in the reverse order of execution of the initialisation code.

Internally, Virtual Pascal implements this by generating a call to the *AddExitProc* procedure at the beginning of every initialisation statement, adding the finalisation statements into the list of exit procedures to be executed when the program terminates. For this reason, the finalisation part must be present only if an initialisation part is present.

The finalisation part must be prepared to handle partially initialised data. This can happen in various cases, for example when the initialisation part of a unit invokes the *Halt* or *RunError* standard procedures to terminate the program because of an unrecoverable error – the finalisation code must handle this case correctly.

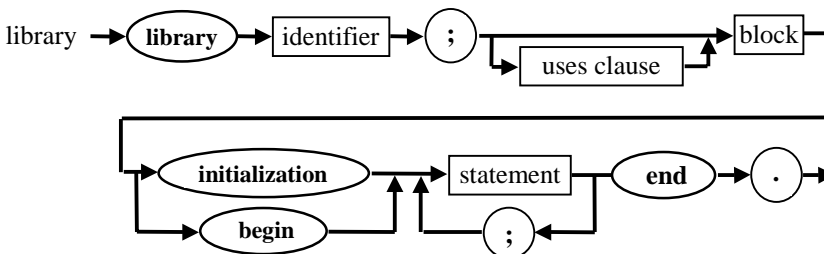


If no initialisation and finalisation is needed for a unit, its initialisation and finalisation parts consist of the reserved word **end** only. Use non-empty initialisation and finalisation parts only when it is really necessary, because it reduces the chances of smart linking out unused code. Even if the program only uses untyped constants and types declared in the unit, the initialisation and finalisation parts (including all variables, procedures and functions used by them) will be linked into the program.

For compatibility with Borland Pascal, the reserved word **begin** can be used to start the initialisation part.

Dynamic link library

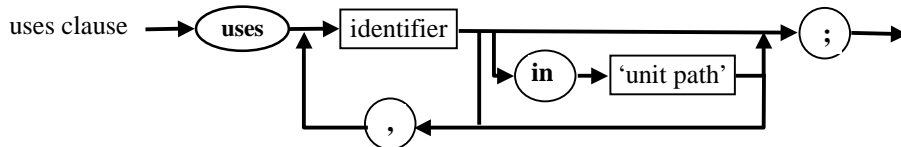
A dynamic link library is a dynamically linked executable module that allows other applications to use parts of its variables, procedures and functions. When a program is running and it uses a DLL's variables, procedures or functions, the references in the program are dynamically linked to the corresponding entry points in the DLL. The following diagram defines the structure of a DLL:



A DLL has the same structure as a program, except that the reserved word **library** must be present. For a detailed guide on how to write dynamic link libraries, refer to the chapter *Dynamic link libraries* on page 127.

Uses clause

The **uses** clause lists all units that are used by a **program**, a **library** or a **unit**.



The **uses** clause may not necessarily include all units that are actually linked into the program or library, since units can be included indirectly by themselves being used by units present in the **uses** clause. When including a unit in the uses clause, all declarations in the interface part of that unit become available for use in the program.

The *System* unit, containing all predefined (built-in) variables, procedures and functions, is always implicitly used as if it was included as the first unit in the **uses** clause (in the **interface uses** clause of the unit). It does not need to be listed explicitly in the **uses** clause.

In the **{&Use32+}** state, the *Use32* unit is also implicitly included in the **interface uses** clause (See also page 164).

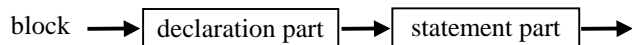
To explicitly specify the full path and file name of the unit, the **in** reserved word can be used, followed by the name of the unit in quotes. Delphi uses this syntax when including units defining forms.

Example:

```
uses
  SysUtils, VPUtills, MyUnit,
  MyFuncs in 'c:\vp\source\myfuncs.pas';
```

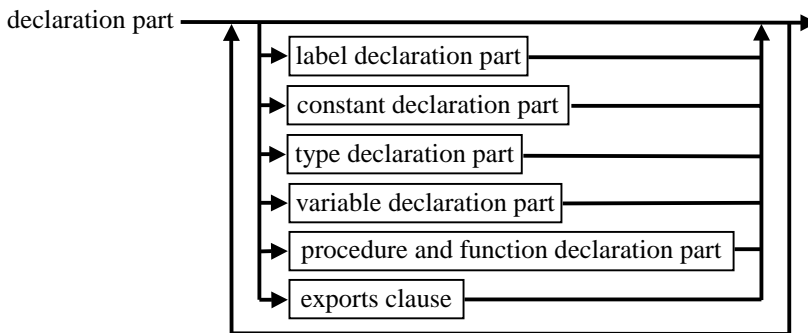
Block

A *block* is part of a program, unit, procedure, function or method declaration, containing a list of declarations and a statement part. Virtual Pascal allows the use of any number of declaration sections in any order. Here is the syntax of a block:

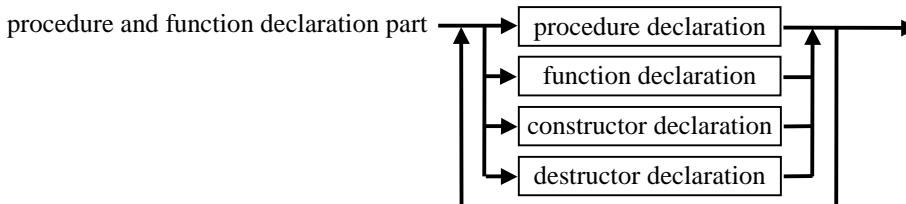
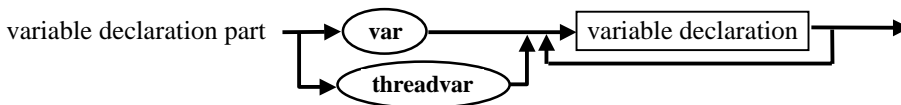
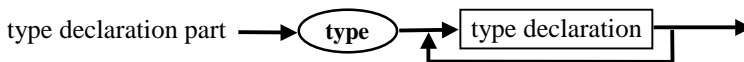
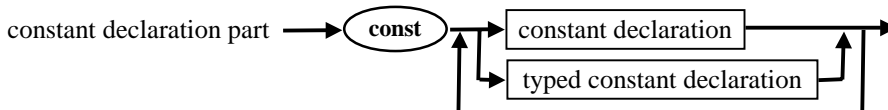
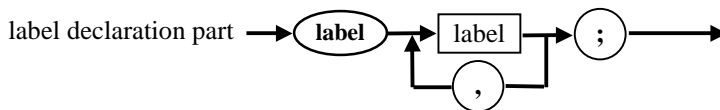


The declaration part is used to declare program objects: labels, types, constants, variables, procedures and functions. Since Virtual Pascal is a one pass compiler, i.e. it generates code by parsing the source code of a program only once, every identifier or label must be declared before it is actually used.¹

¹ The exception to this rule is the declaration of pointer types. Pointer types can be declared as pointing to not-yet-defined identifiers. The identifier pointed to must be defined elsewhere in the same type declaration part.

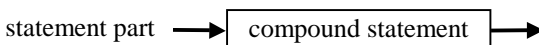


All declarations in the declaration part are local to the block in which they are declared. The syntax of each declaration part is shown below:



An **exports** clause specifies which procedures or functions are to be exported by a dynamic link library. See the description on how to export symbols from a DLL on page 130.

The statement part consists of a compound statement which implements the required functionality of the program, procedure or function.



Scope

The scope of an identifier or label is that part of the program in which the identifier (or label) can be used to access the entity it identifies (or statement it marks).

The rules of scope define the order in which the compiler looks for an identifier or a label. When an identifier or a label is encountered, it is searched for starting from the innermost scope and ending in

the outermost one. All identifiers must be unique within a particular scope. This ensures that no ambiguity arises during the searching process.

There are three categories of scope: (i) *record*, *class* and *object*; (ii) *block* (or *local*); and (iii) *unit* scope. These depend on how and where identifiers are declared. *Record*, *class* and *object* scopes are the innermost ones, followed by *block* scope and outermost *unit* scope.

Record scope (i)

The scope of a field identifier with *record* scope starts at the declaration point and ends at the end of the record-type definition. It includes field designators and **with** statements used with the given record-type.

Object and class scope

The scope of a component identifier with *object* and *class* scope starts at the point of declaration and ends at the end of the object or class-type definition. The scope includes all descendants, all blocks of all method declarations of an object or class type, field, method and property designators, in addition to **with** statements acting on the given object or class-type.

Block scope (ii)

The scope of an identifier or label with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as an *enclosing* block).

If an enclosing block contains an inner block, an identifier or label declared in the enclosing block can be re-declared within the inner block. After the end of the inner block, the identifier or label will have the same meaning as it had before.

Unit scope (iii)

The scope of an identifier declared in the **interface** part of a unit extends over all programs, units or libraries which use this unit. Each unit in the **uses** clause introduces an outer scope which encloses all scopes of other units in the **uses** clause, listed later.

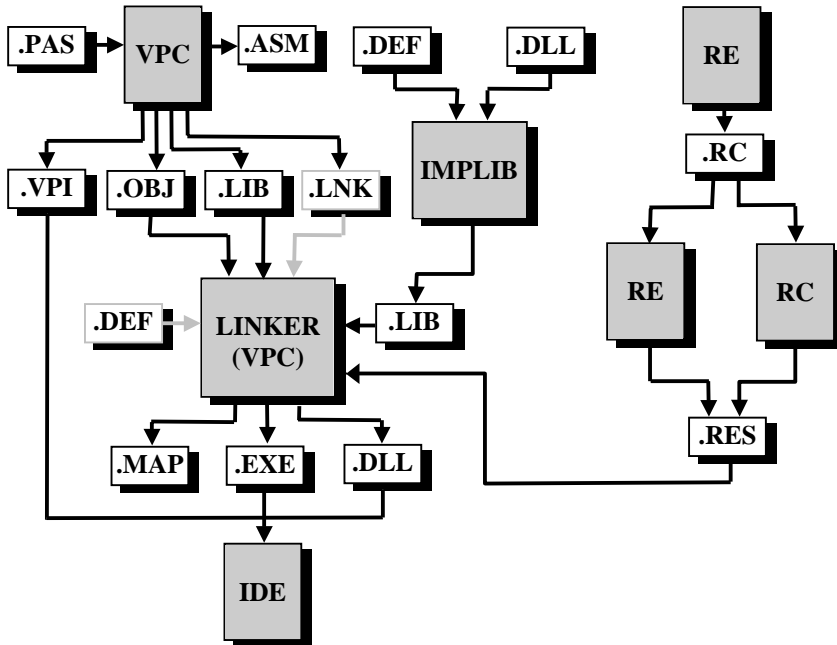
This means that if two or more identifiers with the same name are located in different units and both are included in the **uses** clause, the identifier which is declared *last* in the list of units will be visible. Since the *System* is always included implicitly as the first unit in the **uses** clause, it has the outermost scope. (One way of remembering this is: If the scope rule had worked the “other way”, it would have been impossible to redefine *System* unit functions).

To override the scope rules and select an identifier from a particular unit, qualify its name with a module name. For example, *Dos.DosError* will select the *DosError* identifier in the unit named *Dos*.

The scope of identifiers declared in the **implementation** part includes the **implementation** block of that unit only, from the point at which it is defined.

Compilation process

The following picture illustrates the process of building an application using Virtual Pascal. When using the built-in linker, the steps involving the linker take place transparently to the user.



Initially, the source code for the program and all used units is compiled by the Virtual Pascal compiler, VPC. The output of this is standard OMF-386 (Object Module Format) object (.OBJ) or library (.LIB) files. Note, that if a unit has no code (it is simply used to define constants and types), no object code is generated and there is no .OBJ or .LIB file for it.

A .LIB file is generated for a unit if smart linking is enabled (`{&SmartLink+}` state), otherwise, an .OBJ file is produced. The term *smart linking* means that unused variables, typed constants, procedures, functions and objects are not linked into the executable, thus reducing the size of the final program. To make this possible, each `const` and `var` declaration section, procedure and function is separated into its own object module (all stored inside the resulting .LIB file). When smart linking is enabled, the compiler cannot generate assembler code for the unit, and DLLs compiled with smart linking enabled cannot be symbolically debugged from the Virtual Pascal IDE. Unless this is required there is no reason to disable smart linking.

In the `{&SmartLink-,Asm+}` state, the compiler generates a readable 386 assembly source (.ASM) file in addition to the object code. This file can be compiled by MASM version 6.0 or later and TASM version 3.0 or later.

In the `{SD+}` or `{&Zd+}` state, the Pascal source lines are included in the assembler source as comments, thus showing the source lines alongside the corresponding CPU instructions generated by Virtual Pascal.

The .VPI file also generated by the compiler is a Virtual Pascal Interface file. A VPI file contains precompiled interface symbol information and is the only proprietary version-dependent file format

used by the Virtual Pascal compiler. In spite of this, it is still possible to distribute the unit object code without making the source available. Please refer to the description of the `{&PureInt+}` directive on page 161 for directions on how to do this.

Unlike compilers for other high level languages, Virtual Pascal units can not be compiled separately, but are always compiled as a group. In order to keep track of dependencies between units, the compiler treats them as a group, which is why *make* and *build* facilities are built in to the compiler and accessible from the IDE via the Compile menu.

When *building* a module, the compiler automatically recompiles the unit and all units used by it, including implicitly used ones.

When *making* a module, the compiler checks whether any dependent units have been changed since the last compile and recompiles them if necessary. The compiler checks the current module and all dependent units for the following conditions:

- if the source file for a unit has been modified since the .VPI file was created, the unit is recompiled;
- if the unit source file includes (Using the `{SI}` statement) one or more include files and one or more of them is newer than the .VPI file, the unit is recompiled;
- if the interface of a unit has been changed, all units using it are recompiled.

One difficulty arises when two or more units use each other. The compiler is able to recompile a dependent unit that uses it, only when its own **interface** part already has been compiled. For this reason, mutually dependent units may not use each other in the interface parts. It is possible for one unit to use the other in the **interface** part, but the other one must use the first one in the **implementation** part.

More complex situations may appear with more than two mutually dependent units and the compiler checks for a circular path in the **interface** part **uses** chains of all mutually dependent units. If it finds one, a *'Circular unit reference'* error is reported. To avoid this, it is recommended to include used units in the **implementation** part **uses** clause whenever possible.

The compiler also generates a .LNK file, which is primarily of use when using an external linker. In this case, the generated .LNK file is passed to the linker program. A .LNK file is a text file containing a list of object and library files comprising the module, the module definition file name (if the compiler found one) and a map file name to be generated when applicable. The .LNK file can be examined to see a list of all units that are linked into the executable. By examining the extension (.OBJ/.LIB) of the files listed, the **&SmartLink** setting used to compile each unit can be determined.

The linking process

Virtual Pascal produces *only* standard object records (no proprietary extensions), which can be linked by any 32-bit OMF linker capable of generating OS/2 Linear eXecutable (LX) or Win32 PE files. Virtual Pascal includes a highly efficient built-in linker capable of linking for both these targets and using an external linker is rarely, if ever, required.

The object and library files produced by the compiler are linked together to form the final executable file. External object or library files and import library files must be specified using the `{L FileName}` compiler directive (refer to page 157 for details).

The module definition file (.DEF) is an optional file containing detailed information about the executable. If present, the module definition file must have the name of the primary program or library file (with a .DEF extension) and must be located in the same directory. Alternatively,

module definition statements can be included directly in the source code of the program, using the **{&Linker}** directive described on page 158.

When using an external linker, the content of a **{&Linker}** directive will be extracted from the source text and stored in a .DEF file of the same name, overwriting another .DEF file of the same name.

The linker can generate a map file containing information about segments by setting the IDE Option|Linker|Map file setting to *Segments only*. More detailed information can be obtained by setting it to *Detailed*. The detailed information includes a list of public symbols and their addresses, imported symbols, detailed segment information and a list of all line numbers and their addresses in code.

With a map file available efficient bug finding is possible. If a customer reports an error, it is easy to find the error location if the original map file is available. This does not affect the size of the executable file.

For a more advanced approach to the problem of locating run-time errors, please refer to the information on the **{&LocInfo}** switch directive, described on page 158.

Creating an import library

When compiling a dynamic link library file, the IDE can automatically create an import library for it. This has the same name as the primary file of the library, has a .LIB extension and is placed in the directory that is listed first in the Options|Directories|Library directories input box.

An import library contains information about all symbols exported by the DLL and can be used with a DLL to tell the compiler which symbols are defined in it.

Two default import libraries, OS2.LIB and IMPORT32.LIB, contain the import declarations of the OS/2 and Win32 API functions, respectively. These libraries should not be specified using **{&L FileName}** directives; the correct one is automatically included based on the destination platform chosen in the Compile menu.

In the IDE, the Options|Linker|Generate import library radio buttons specify which file (module definition or DLL executable) should be taken as the source to create the import library. If the *None* radio button is selected, an import library is not generated when creating a DLL.

Compiling and binding resources

Resources include icons, cursors, bitmaps, menus, dialogue boxes, fonts, string tables and other user-defined data. Virtual Pascal is capable of converting several kinds of resources (.RES files) to a format appropriate for the target platform and linking them into the executable.

To generate the resource files, a resource editor is usually required; resource compilers and editors are included with all Borland Pascal products; OS/2 is delivered with a default resource compiler, RC.EXE.

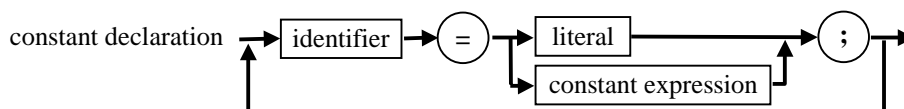
If a resource file with the same name as the main program file is found in the same directory (but with a .RES extension), it is automatically linked into the executable.

Additionally resource files can be included (from either the program file or a unit) by using the **{&R ResFile}** directive.

CHAPTER 4

Untyped constants

A literal may be given a name. A named literal is called an untyped constant or simply a constant. The value of the constant is defined once and can not be changed. In addition to literals, a *constant expression* can be used to define the value of an untyped constant. Untyped constants are defined in the constant declaration section.



A *constant expression* is an expression that can be evaluated at compile time. Its factors must be previously defined constants, constant addresses (obtained via the address operator @, or one of **Addr**, **Ofs**, **TypeOf**, **TypeInfo**) or calls to the following standard functions:

Abs	Length	Ord	Succ
Addr	Lo	Pred	Swap
Chr	Low	Ptr	Trunc
Hi	Odd	Round	TypeOf
High	Ofs	SizeOf	TypeInfo

Examples:

const

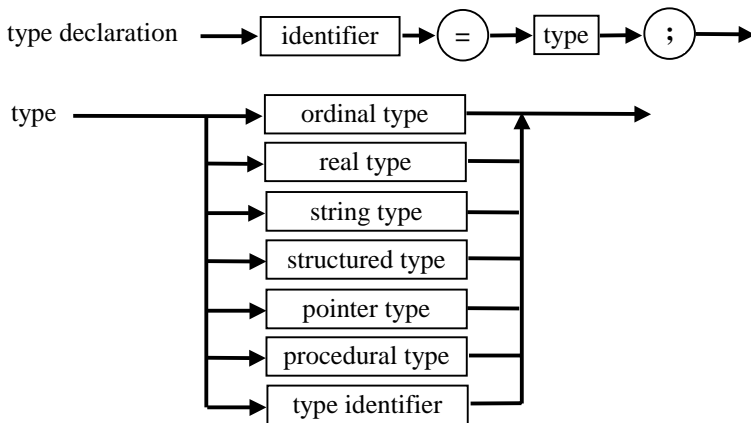
```

A = 0;
B = 1000;
AB = Abs(A - B);
Average = AB div 2;
space = ' ';
readme = ' These are constants: ';
length = SizeOf(readme);
file = readme + space + '1 2 3 4 5';
Set1 = [0..100];
Set2 = [101..150];
UnionSet = Set1 + Set2;
    
```

Data types

This chapter contains a description of the Virtual Pascal language data types. The term *type* is a fundamental concept of programming languages. Program objects, such as constants, variables, procedures and functions, expressions, etc., always have an associated type. The type defines the set of values that a program object can have as well as the operations that can be performed on it. Virtual Pascal provides a set of basic fundamental types as well as a mechanism for creating new user-defined types.

Types are declared in the type declaration section, which has the following format:



Ordinal and real types are also known as simple types.

This chapter contains only a brief description of operations and standard functions. A detailed description of standard procedures and functions is given in the *Runtime Library Reference* which is included as part of the online help useable from within the Virtual Pascal IDE. More information about operators can be found in the *Expressions* chapter on page 81.

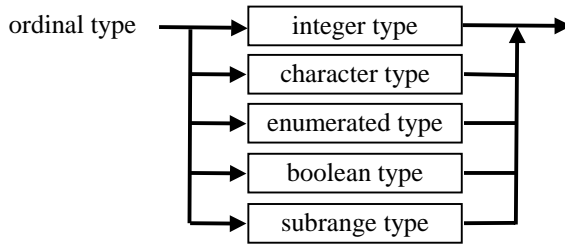
Ordinal types

Ordinal types are types whose values can be enumerated. For each value of an ordinal type, there is a corresponding numeric ordinal number returned by the *Ord* standard function. The *Succ* and *Pred* standard functions return the next and the previous value of a given ordinal argument, respectively. The *Low* and *High* standard functions return the lowest and the highest value for a given ordinal argument.

The following relational operators can be applied to values of any ordinal type:

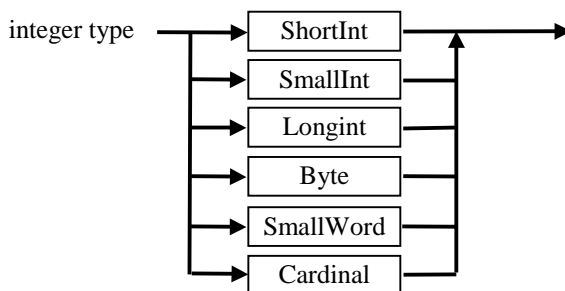
= > < > <= >=

They compare ordinal numbers of two ordinal type values; = denotes equality, > “larger than” and < “less than”.



Integer types

Integer types are ordinal types that denote a specific subset of whole numbers. The ordinal number of an integer type value is the value itself. Virtual Pascal provides a number of basic predefined integer types.



The size and range of the basic integer types is shown in the following table.

Table 2-1. Basic integer data types

Type	Range	Format	Size (bits)
<i>ShortInt</i>	-128..127	Signed	8
<i>SmallInt</i>	-32768..32767	Signed	16
<i>Longint</i>	-2147483648..2147483647	Signed	32
<i>Byte</i>	0..255	Unsigned	8
<i>SmallWord</i>	0..65535	Unsigned	16
<i>Cardinal</i>	0..2147483647	Unsigned	31

In addition to the basic types, Virtual Pascal provides the types *Integer* and *Word*. They are defined in the *System* unit as follows:

type

Integer = *SmallInt*;

Word = *SmallWord*;

The primary aim of Virtual Pascal is to be a 32-bit cross-platform compiler, compatible with 16-bit Borland Pascal for DOS as well as 16- and 32-bit Borland Delphi Object Pascal. However, one main obstacle to achieving the level of compatibility required is that the 3 compilers use different sizes of the basic integer type. The following paragraphs outline the problem and the various solutions provided by Virtual Pascal.

Basic integer types

The basic integer types defined in the *System* unit of Virtual Pascal are compatible with 16-bit Borland Pascal. However, using 16-bit integer variables on a true 32-bit processor is inefficient and

we recommend including the *Use32* unit in the **uses** clause of every unit. This unit redefines the 16-bit integer types *Integer* and *Word* as the 32-bit *Longint* type. In Borland Pascal, the *SmallInt* and *SmallWord* types are not defined; in Delphi, only the *SmallInt* type is defined. To increase cross-compiler usability and compatibility, the *SmallInt* and *SmallWord* types are defined in the *Use32* unit if it is compiled by any compiler other than Virtual Pascal; these are defined in the *System* unit of Virtual Pascal. To add to the confusion, 32-bit Delphi defines *Word* as a 16-bit unsigned type.

Discussion: Use32, Use16 and bits

The use of the *Use32* unit ensures that all basic types of Virtual Pascal have the same size and range regardless of the compiler used to compile the program (Borland Pascal, Virtual Pascal or Borland Delphi). Where the size of integer variables is significant, for example when saving/retrieving information from binary files, the basic integer types with a well-defined size (such as *SmallInt*) should be used. In all other cases, the *Integer* and *Word* types should be used: since these change size depending on platform and compiler used, this maximises performance.

The **{&Use32+}** directive, introduced in Virtual Pascal v2.0, automates the procedure described above. When this directive is defined, the compiler implicitly includes the *Use32* unit in the **uses** clause just after the *System* unit, before any other units are included. By default, the **{&Use32}** directive is disabled but should be enabled for programs written in Borland Pascal.

When the **{&Use32 +}** directive is enabled, it is possible to revert to the 16-bit integer types in selected units, either by changing the setting to **{&Use32 -}** at the beginning of the unit, or by including the *Use16* unit in the **uses** clause. The *Use16* unit effectively cancels the type re-definitions done by the implicitly included *Use32* unit.

As opposed to Borland Pascal, Delphi Object Pascal was designed with both a 16-bit and a 32-bit version of the compiler in mind. In Delphi, the concept of fundamental and generic types has been introduced. The fundamental types in Delphi Object Pascal are *ShortInt*, *SmallInt*, *Longint*, *Byte* and *Word*. The ranges of these fundamental types are independent of the underlying CPU and operating system and do not change across different implementations. They have the same meaning in Virtual Pascal as in Delphi.

The generic integer types are *Integer* and *Cardinal*, the ranges and format of which depend on the underlying CPU and operating system. For the 16-bit version of the Delphi compiler they are 16-bit, for the 32-bit version they are 32-bit. In all Delphi programs, the *Integer* and *Cardinal* types are usually used, resulting in more efficient integer operations for the underlying CPU and operating system. For this reason it is not necessary nor recommended to include *Use32* into the **uses** clause of a Delphi program. Instead, the Virtual Pascal *SysUtils* unit is used to redefine the meaning of the type *Integer* to be equivalent to the *Longint* type. As a result, including *SysUtils* in the **uses** clause makes Virtual Pascal programs compatible with the 32-bit Delphi compiler. The *SysUtils* unit is included automatically in the interface **uses** clause by the Delphi environment. Make sure to include *SysUtils* unit in the interface **uses** clause of all other units written manually. The **{&Use32}** directive should not be used for Delphi programs as the *Word* type is 16-bit in all versions of Delphi and is redefined to be 32 bits by the *Use32* unit.

Signed types are stored in the 2's complement format with the sign in the most significant bit.

There are two predefined integer constants:

- *MaxInt* defines the largest possible *SmallInt* (32767)
- *MaxLongint* defines the largest possible *Longint* (2147483647).

Note

The *MaxInt* constant is redefined by both *Use32* and *SysUtils* to be equal to *MaxLongint*; whereas the *System* and *Use16* units define it as shown above.

The following binary arithmetic operators,

+ - * **div** **mod**

are associated with integer types. The '-' (minus) operator may be used as an unary operator to change the sign of an operand. The '/' (division) binary operation can be used on integer operands, but the result of the operation is always of type *Extended*.

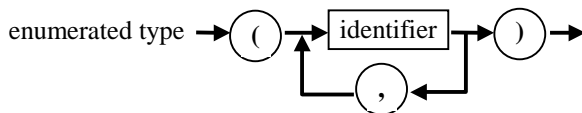
Character types

A character type is an ordinal type used to store ASCII characters. Virtual Pascal provides one predefined character type, *Char*. Its size is 1 byte. The *Chr* standard function converts an integer value into a character with the corresponding ASCII value. The *Ord* function returns a character's ASCII value. A character string consisting of just one character is used to specify a constant of type character.



Enumerated types

An enumerated type is an ordinal type that defines a list of values by enumerating the identifiers denoting the possible values. The ordinal number of the first identifier is **0**, of the second **1**, of the third **2**, etc.



The identifiers in the type definition of an enumerated type become constants of the enumerated type. The first constant has ordinal number 0, the second has ordinal number 1 etc., for example:

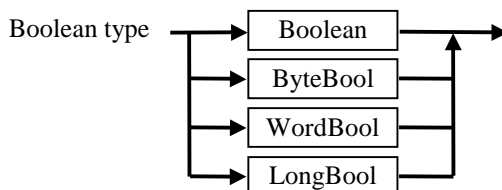
type

Metal = (Fe, Cu, Ag, Au, Pl); // Ord(Fe) = 0, Ord(Cu)=1,..Ord(Pl)=4

If the *Word Sized Enumerated* compiler directive is disabled **{Z-}**, enumerated types occupy 1 byte when the number of constants of the enumerated type is less than or equal to 256 and 2 bytes if the number of constants is larger than 256. In the **{Z+}** state, enumerated types always occupy 4 bytes of memory.

Boolean types

Boolean types are ordinal types that have only two values: *False* ($Ord(False)=0$) and *True* ($Ord(True)=1$).



The *Boolean* type can be regarded as an enumerated type with some special features and the following definition:

32 Data types

type

Boolean = (*False*, *True*);

The following table lists all predefined boolean types:

Type	Size(bytes)	Only <i>True</i> and <i>False</i> are allowed
Boolean	1	Yes
ByteBool	1	No
WordBool	2	No
LongBool	4	No

The *Boolean* type does not allow values with ordinal numbers other than 0 and 1. The *ByteBool*, *WordBool* and *LongBool* types are provided for compatibility with C language booleans, which are considered to be *False* when their ordinal number is 0 and *True* when the ordinal number is non-zero. When these boolean types are used in a context where a *Boolean* value is required (e.g. in accessing element of an array with a boolean dimension type), the compiler automatically converts non-zero values to *True*.

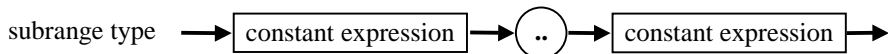
In an expression, the following relational operators produce results of type *Boolean*:

= <> > < >= <= in

The logical operators **not** (unary), **and** and **or** (binary), work by testing for 0 (*False*) or non-zero (*True*), but always return a result with an ordinal number of 0 or 1.

Subrange types

A subrange type is a range of values taken from an ordinal type called the base type. Two constants defining the lower and upper bounds declare the subrange.



Both expressions must be of the same ordinal type and the first one must be less than or equal to the second. The first expression must not start with an open parenthesis symbol '(' since this starts an enumerated type declaration.

Subrange types have all the properties of the base type, but the values are limited to the defined range.

The size of a subrange variable is equal to the size of the predefined integer type with the smallest range that covers the entire subrange.

Examples:

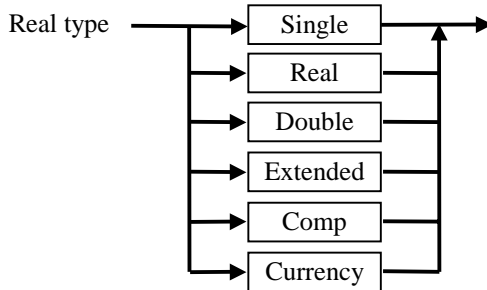
type

Hour = 1..24;

PreciousMetal = Ag..Pl;

Floating point types

A floating point (or “real”) type has a set of values that is that subset of real numbers which can be represented in a floating-point notation with a fixed number of digits.



The normalised range of each type is shown in the following table.

Type	Normalised range	Digits	Size(bytes)
<i>Single</i>	$1.18 \times 10^{-38} \dots 3.4 \times 10^{38}$	7-8	4
<i>Real</i>	$2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$	11-12	6
<i>Double</i>	$2.23 \times 10^{-308} \dots 1.79 \times 10^{308}$	15-16	8
<i>Extended</i>	$3.37 \times 10^{-4932} \dots 1.18 \times 10^{4932}$	19-20	10
<i>Comp</i>	$-9.2 \times 10^{18} \dots 9.2 \times 10^{18}$	19-20	8
<i>Currency</i>	-922337203685477.5808.. 922337203685477.5807	19-20	8

Virtual Pascal always uses the 80x87 coprocessor for floating point operations. *Real* is supported for compatibility with Borland Pascal only and should be avoided whenever possible, both for performance and for accuracy reasons. Every time an operation on a *Real* value is encountered, it is converted to *Extended* and back, which means that the *Real* type only should be used where the size of the data type is of prime importance.

The *Comp* type is a 64-bit integer capable of holding an integer value between $-2 \times 10^{63} + 1$ and $2 \times 10^{63} - 1$. It is regarded as a floating point type since it can not be used in contexts where integer and ordinal types are accepted.

The *Currency* type is a fixed-point data type that can be used for monetary computations. It is stored as a scaled 64-bit integer with the four least-significant digits implicitly representing four decimal places. When referenced in expressions or used in assignment statements with other real types, values of type *Currency* are automatically scaled by dividing or multiplying by 10,000. Since numbers stored in the *Currency* format are exact representations, operations on *Currency* values are not subject to rounding errors.

34 Data types

The internal representation of the floating point types are as follows:

Data formats	Range	Precision	Most significant byte (highest addressed byte)												
			7	0	7	0	7	0	7	0	7	0	7	0	
<i>Comp</i>	10^{18}	64 bits	[] two's complement 63												
<i>Currency</i>	10^{18}	64 bits	[] two's complement 63												
<i>Single</i>	$10^{\pm 38}$	24 bits	S	biased exp	significand										
			31	23											
<i>Double</i>	$10^{\pm 308}$	53 bits	S	biased exponent	significand										
			63	52											
<i>Extended</i>	$10^{\pm 4932}$	64 bits	S	biased exponent	I	significand									
			79	64	63 ^Δ										
<i>Real</i>	$10^{\pm 38}$	39 bits	S	significand								biased exp			
			47									7			

S = sign bit (0 - positive, 1 - negative)

Δ = position of implicit binary point

I = integer bit of significand; stored in temporary real, implicit in single and double

exponent bias (normalised values) :

single : 127 (7FH)

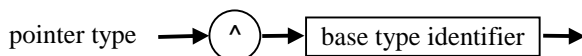
double : 1023 (3FFH)

extended: 16383 (3FFFH)

real: $(-1)^S (2^{E-BIAS})(F_0 F_1 \dots)$

Pointer types

A pointer type is a special type that is used to store an address of a variable of a specified type, called the based type.



The base type identifier can be any type identifier, even one that has not yet been declared. In this case, the declaration of the base type should be present somewhere in the same **type** declaration section. This feature allows the declaration of pointers to a structured type that contains a pointer to itself, for example:

type

PNode = ^TNode;

```

TNode = record
  Next: PNode;
  Info: Integer;
end;

```

A pointer variable does not represent a variable of the base type itself, it only holds its address (points to it). When there is no variable of the base type associated with a pointer, a special value denoted by the reserved word **nil** is used. **nil** is compatible with all pointer types. A **nil** pointer value is an address that is guaranteed to be different from any valid pointer and is encoded as an address with the value zero.

A pointer variable is stored as a double word (4-byte value) representing a 32-bit flat memory offset. The segment (selector) part of the address is not available and is always assumed to be *flat*.

The predefined type *Pointer* defines a generic pointer that does not have any base type, which means that it is a pointer to anything. Values of type *Pointer* are compatible with all other pointer types.

The predefined type *PChar* is declared in the *System* unit as follows:

```

type
  PChar = ^Char;

```

Unlike all other user defined pointers to type *Char*, *PChar* pointers have special properties, which take effect only when extended syntax is enabled (**{{X+}}** state). They are used to hold addresses of *null-terminated strings*. These are strings consisting of a sequence of non-null characters with a terminating NUL (#0) character. Unlike the *ShortString* type (see the description of the *ShortString* type on page 37) null-terminated strings have no length byte and can be of any length. They are stored in special character arrays called *zero-based character arrays* (refer to page 40 for more information on array types).

The following relation operators can be applied to all pointer types:

```
= <>
```

Additional operators can be applied to *PChar* pointers only and only when extended syntax is enabled:

```
< > <= >= + -
```

The *New* standard procedure takes a pointer variable as parameter, allocates dynamic memory for a variable of the pointer base type on the heap and returns the address of it in the pointer variable. The standard procedure *GetMem* takes a pointer variable and a block size as parameters and returns the address of a newly allocated memory block of the given size in the pointer variable. The *Ptr* standard function creates a pointer with a specified pointer value. Note, that unlike the 16-bit implementation of Object Pascal, *Ptr* in Virtual Pascal accepts only one argument - the flat memory offset. The standard procedures *Inc* and *Dec* can be used with pointer types to increment and decrement pointer values by the specified number of base type variables. This can be helpful when performing operations on array elements, e.g.: incrementing a pointer to an array element by one will cause the pointer to point to the next array element.

Inc and *Dec* can not be used with pointers of type *Pointer*, as the compiler does not have size information associated with a *Pointer*. The address operator *@* returns the address of a variable, procedure or function entry point. All pointer types can be dereferenced by writing the dereference pointer symbol *^* after the pointer value. This operation results in a variable of the base type with an address as defined by the value of the pointer.

When dereferencing a value of type *Pointer*, the result is an untyped value that can be used when passing an untyped **var** parameter as a function parameter, for example.

The following features are applicable to *PChar* pointers only.

- In the **{SX+}** state (when extended syntax is enabled) a constant string expression can be assigned to a *PChar* pointer. For example:

```
var
  Letters: PChar;
begin
  Letters := 'ABCDEFGF';
end;
```

After this assignment, *Letters* will contain the memory address of a null-terminated copy of the string constant expression.

- If the formal parameter in a procedure or function call is of type *PChar*, it is possible to use a constant string expression as the actual parameter. In this case the compiler will also generate a null-terminated copy of the string constant expression and the *PChar* pointer will hold the memory address of this copy. For example for the procedure:

```
procedure Error(ErrStr: PChar);
the following call is valid:
Error( ' Unresolved reference ' );
```

- Constant string expressions can also be used as initialisers for simple or structured typed constants containing *PChar* types. For example:

```
const
  Name: PChar = Queen ' + 'Elizabeth ' + ' II';
```

Note

Unlike Borland Pascal and Borland Delphi v1, Virtual Pascal imposes no length restrictions on string constants.

- A *PChar* type pointer can be indexed as a *zero-based character array* (See the Array type for more information about *zero-based character arrays*). For example:

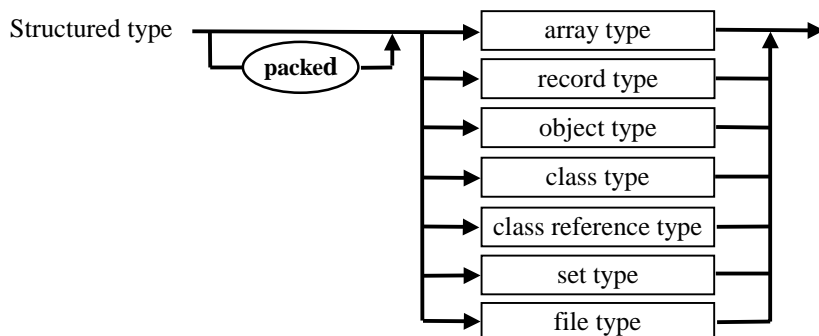
```
var
  Symbols: array[0..10] of Char
  Signs: PChar;
  Item: Char;
  Letter: Char;
begin
  Signs := Symbols;
  Item := Signs[0];
  Letter := Signs[1];
end;
```

In this case the variable *Item* will be assigned the first and the variable *Letter* the second element of the array *Symbols*. The memory address of *Signs*[0] is the same as the value of the pointer *Signs*. *Signs*[0] is essentially an equivalent to *Signs*[^], *Signs*[1] is the same as (*Signs*+1)[^] and so on.

- In the `{SX+}` state (when extended syntax is enabled) the *Assign*, *Rename*, *Val*, *Write* and *Writeln* standard functions can be used with *PChar* type pointers and zero-based character arrays.

Structured types

Structured types are special types that consist of one or more simple or structured component. Except for the *AnsiString*, *ShortString*, *Text* and **file** types, structured types are user defined types. The following sections contain a detailed description of Virtual Pascal structured types.



The reserved word **packed** can be specified before the declaration of a structured type. It instructs the compiler to use a more efficient way of storing the components of the structured type. In the current implementation of Virtual Pascal, the setting defined by the `{&AlignRec+}` directive can be overridden by specifying the **packed** reserved word.

String types

Virtual Pascal supports two types of strings: short strings and long strings. These can be mixed in assignments and expressions and the Virtual Pascal compiler automatically generates the code required to perform any string type conversions that may be required.

The short string type is a structured type consisting of a sequence of characters. A short string has a dynamic length and a maximum length between 1 and 255 that must be specified at compile time. Both Borland Pascal and all versions of Borland Delphi support the short string type.

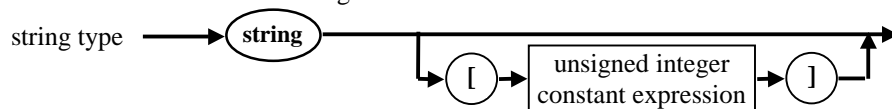
The long string type is a structured type that contains a sequence of characters, whose length is only limited by available memory. It is allocated dynamically during program execution. The long string type is not available in Borland Pascal or in 16-bit Borland Delphi v1.x but is supported by 32-bit versions of Borland Delphi and by Virtual Pascal.

The state of the `{SH-}` compiler directive determines whether the reserved word **string** denotes a short string or a long string. In Virtual Pascal, the default state of the directive is `{SH-}`.

Irrespective of the state of these compiler directives, the predefined identifiers *ShortString* and *AnsiString* (defined in the *System* unit) can be used to explicitly select the desired string type.

Short strings

The declaration of a short string is as follows:



The unsigned integer expression constant denotes a maximum length. If it is not specified, a maximum length of 255 is assumed (in the **{SH-}** state; in the **{SH+}** state, a **string** declaration not specifying a maximum length denotes an *AnsiString*).

The actual dynamic length of the string is returned by the *Length* standard function.

The standard function *High* returns the declared maximum length of the string.

The individual character elements of a string can be indexed. Indices begin with 0. The first character (index=0) contains the string length, although the standard function *Length* should be used wherever possible as it functions with both short and long strings. Variables of short string type occupy the maximum declared size + 1 byte.

Long strings

The declaration of a long string is as follows:

long string type \longrightarrow **string** \longrightarrow

The **string** reserved word is only equivalent to an *AnsiString* in the **{SH+}** state.

Long strings are allocated dynamically at run-time and have no declared maximum length, except for a theoretical upper length of 2GB – or 2 billion characters. In reality, this means that only available memory and disk space limits the length of the string.

The operations necessary to manage the allocation and deallocation of memory for long strings are handled entirely by the compiler and require no user code. A long string is really a pointer to a string of characters and it is **nil** if the string is empty. When the string is not empty, the long string variable points to a sequence of 0-terminated characters making up the value of the string. In addition, the compiler maintains a reference count and the dynamically allocated length of the long string at offsets -4 and -8 of the base string, respectively.

The reference count for a long string is maintained by the compiler and is incremented when it is assigned a new value (if it already had a value, the reference count of that string is decremented). When the reference count of a long string reaches zero, the memory dynamically allocated for the string is freed. This logic means that string handling can become more efficient: When a long string is assigned a value, only the 32-bit pointer value is copied rather than the entire string and several variables can reference the same string without using additional memory.

When a long string is modified and the reference count of the string is greater than one, a copy of the string is created before it is modified; the reference count of the original string is decremented. This technique is called copy-on-write and ensures that data is not inadvertently modified.

Since global variables never go out of scope, long strings declared as global variables are only freed when the program terminates.

Long string variables are always initialised to be empty (**nil**) when they are created. When they go out of *scope*, for example when exiting a function or procedure containing local long string variables, or when an object containing long string fields is destroyed, the reference count of the string is decremented.

Long strings do not have their length stored as a single byte at offset 0 (as do short strings); to get the length of a long string, the *Length* function should be used. To set the length of a long string

explicitly, the *SetLength* standard procedure should be used. *SetLength* allows both short and long strings to be passed as the parameter.

As mentioned above, long string variables are always 0-terminated. When assigning a value to a long string, the compiler automatically appends a #0 at the end of the string, although this trailing #0 is not counted when retrieving the length of the string.

Because of the automatic 0-termination of long strings, they can be directly typecast to *PChar*; typecasting a long string to *PChar* is guaranteed to return a pointer to a null-terminated string, even if the string is empty. When a long string is typecast to *PChar*, it is valid in the current statement only and should be considered to be *read only*. Consequently, a long string typecast to a *PChar* value should be used in an expression or as a parameter value only.

It is possible to modify a string typecast to *PChar* only when the string is non-empty and unique. Calling the *SetLength*, *SetString* or *UniqueString* standard procedures guarantee that a long string is unique (i.e. has a reference count of 1).

When long string variables are declared as fields of a record type, they cannot form part of any variant part of the record. To do this would potentially prevent the compiler from performing the automatic household tasks associated with long strings.

Example:

```

type
  NumStr = string[10];
const
  S1000: NumStr = '1000';
var
  MyAnsiStr : AnsiString;
  MyShortStr : ShortString
begin
  MyShortStr := 'Test';
  MyAnsiStr := MyShortStr + ' of AnsiString';
  WriteLn( MyShortStr > MyAnsiStr );
end.

```

Strings of short and long types can be mixed in expressions and assignment statements and can be passed by value as parameters. When passing a string as a **var** parameter, the type of the parameter must match the type used in the parameter declaration. A short string can be explicitly converted to a long string by typecasting it (as in *AnsiString(S)*) and vice versa (*ShortString(S)*).

The following operations can be performed on string types:

+ = <> < > <= >=

The relation between two strings is established by applying the relation to characters on corresponding positions. If all characters of the first string are equal to the corresponding characters of the second string, but the first string is shorter, it is considered to be smaller.

For example,

'ABC' > 'ABB', 'ABC' < 'ABCD'.

The + operator performs concatenation of two strings, for instance,

'Virtual' + 'Pascal' = 'Virtual Pascal'

Open strings are a special case short of string variables. Variable parameters, declared using the standard *OpenString* identifier or the reserved word **string** in the **{SP+,H-}** state are taken to be open

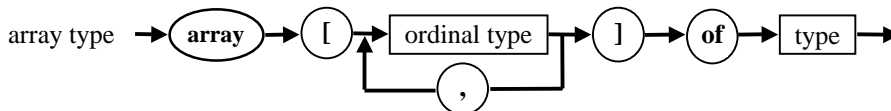
short string parameters. Unlike normal variable short string parameters, information about the maximum declared size of the actual parameter is available. It is passed automatically as an additional parameter to the procedure or function just after the open string parameter address. The *High* standard function returns this value and the *SizeOf* standard function returns this value plus one.

When passing a long string to an *OpenString* parameter, the string is truncated to 255 characters and the *High* standard function returns 255.

Short string variables of any maximum declared length can be passed as actual parameters to open string parameters. Using open strings makes it possible to control the indexing of an actual parameter in the **{*R+*}** state.

Array types

An array is a structured type consisting of a finite number of elements of a simple or structured type. The declaration of an array type is as follows:



The ordinal type denotes the number of indices and therefore the number of elements of the array. Unlike Borland Pascal, where an array index can not be of *Longint* type, Virtual Pascal considers all ordinal types to be valid index types.

All array elements must be of the same type - the *element type*. The element type can be any type, even a structured type.

To access an element of an array, specify the array identifier followed by the index (or indices) of the desired element enclosed in square brackets. Here are some examples of array types:

const

```
MaxLen = 100;
```

type

```
Dimension = 1..200;
```

```
TextLine = array[0..MaxLen-1] of Char;
```

```
Vector = array[Dimension] of Integer;
```

The element type of an array may itself be an array. For example:

type

```
Matrix = array[Dimension] of array[Dimension] of Integer;
```

or the declaration of *Vector* may be used in the declaration of *Matrix*:

type

```
Matrix = array[Dimension] of Vector;
```

Matrix can also be declared using the more convenient form of a multidimensional array:

type

```
Matrix = array[Dimension, Dimension] of Integer;
```

The *Low* and *High* standard functions applied to the array's type identifier return the smallest and the largest value of the array index, respectively.

Two forms of character arrays have special features. The first one has the following syntax:

array[*n1*..*n2*] of Char;

where $n1 \leq n2$ and the array size is less than 256 bytes (that is $n2 - n1 < 256$). It is called a *packed string* type. Typed constants of a *packed string* type can be initialised by a constant string expression.

The second form has the syntax:

array[0..*n*] of Char;

where $n > 0$ and is called a *zero-based character array*. These can be used to store *null-terminated strings* (See the Pointer type for more information about null-terminated strings). Typed constants of a *zero-based character array* type can be initialised by a constant string expression the length of which is less than the dimension of the array.

In the **{SX+}** state (when extended syntax is enabled) a zero-based character array can be used instead of a *PChar* value. In this case the compiler takes the address of the first element of the array and uses it as the value of a *PChar* pointer. For example:

var

Symbols: **array**[0..10] of Char;

Signs: PChar;

begin

Signs := *Symbols*;

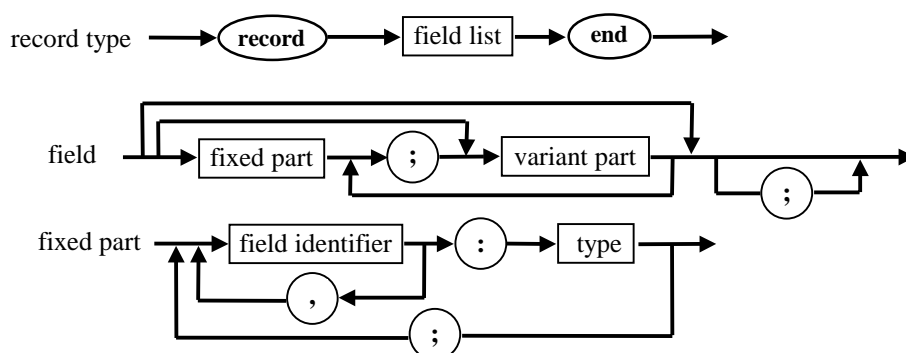
end;

The *Signs* variable will point to the first element of the array *Symbols* after the assignment statement.

When extended syntax is enabled (**{SX+}** state), zero-based character arrays can be used in place of string parameters to the *Read*, *Readln*, *Write*, *Writeln*, *Str*, *Val*, *Assign*, *ChDir*, *MkDir*, *RmDir* and *Rename* standard procedures.

Record types

A record is a structured type that consists of a finite number of elements that may be of different types. The following diagram describes the syntax of a record type:



As the syntax for a field list shows, a record may contain a fixed part, a variant part, or both. If it contains both, the fixed part must be specified before any variant parts. The fixed part contains a list of elements that are the same each time the record is accessed.

The name of the elements must be unique within the record. To select an element of a record type, both the name of the record type and the name of the element should be specified, separated by a period. This is an example of a record type:

type

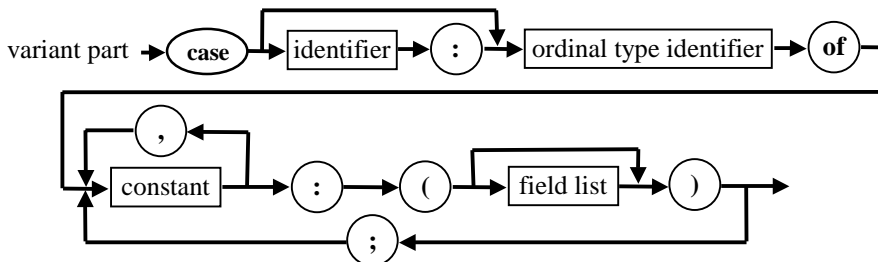
```
DayOfWeek = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
WorkInfo = array[DayOfWeek] of Boolean;
PWorkInfo = ^WorkInfo;
```

```
ScheduleRec = record;
```

```
Day: Mon..Sun;
Hours: 0..23;
Minutes: 0..59;
Sum: Extended;
Winfo: PWorkInfo;
```

```
end;
```

Records of the same type do not necessarily contain the same elements. They can have more than one field list or *variant* and a criterion should be assigned, according to which the program selects the field list needed at a given time. Only one *variant* can be active at any one time, and only fields of the currently active *variant* can be accessed. The size of the *variant* part is the size of the largest *variant*. The declaration of the *variant* part is the following:



In most cases it is convenient to use an additional fixed part field (a *tag field*) in a record with variants to define which of the variants will be selected. The tag field must be of an ordinal type.

Each variant is identified by at least one constant that should be of ordinal type and be compatible with the tag field type. All constants must be distinct.

The variant part of a record must not contain long string, or *AnsiString*, field types.

An example of a record with variants:

type

```
Coordinate = record
xcoor, ycoor: Double;
```

```
end;
```

```
Shape = (Circle,Line);
```

```
Figure = record
```

```
case tag : Shape of
Circle: (Centre: Coordinate; Radius : Double);
Line:(Xcoeff, Ycoeff, Constant : Double);
```

```
end;
```

The syntax of a record type allows the use of a record type **record end;**, called an *empty* record. Since an empty record does not have any fields, its size is 0. An empty record is usually used to mark a specific field without allocating any memory for it.

The fields of a record are stored sequentially in memory with the first field stored at the lowest memory address. The variant parts share the same memory space and each variant starts at the same

memory address. If **{&AlignRec+}** is specified, the next field of the record will be aligned according to its size as follows:

- Byte sized fields are not aligned;
- Word sized (2 byte) fields are aligned at a word boundary;
- All other fields are aligned at a double word (4 byte) boundary.

In **{&AlignRec-}** state, no alignment is performed and record fields are simply placed at the next available offset. In the **{&AlignRec}** state, the same can be achieved by specifying the reserved word **packed** before the record definition.

Object and class types

Virtual Pascal supports two object models: The “old” object model and the “new” class model.

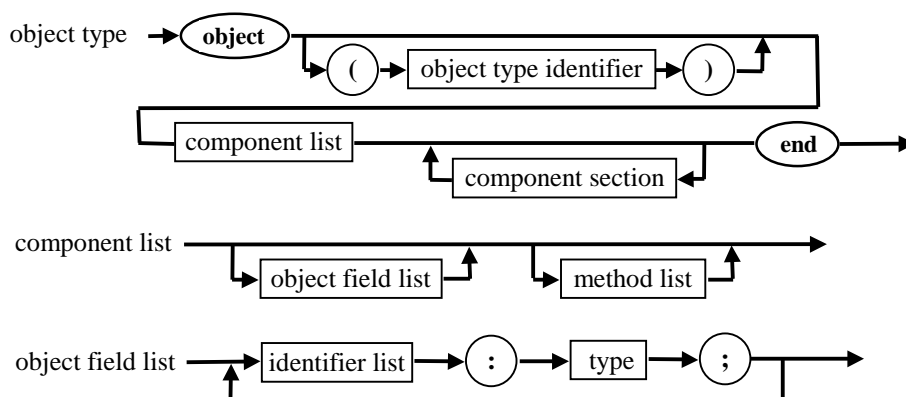
The old object model was introduced in Borland Pascal v5.5 and uses the reserved word **object** for the declaration of object types. The new object model first appeared in Delphi v1 and uses the reserved word **class** in the declaration of object types. The two object models have a lot in common and are described together. To make it clear which object model is used we will call old style object types *objects* and new object types *classes* or *class instance types*.

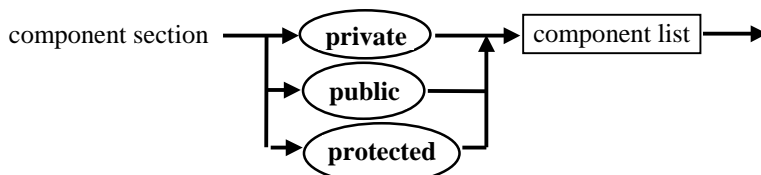
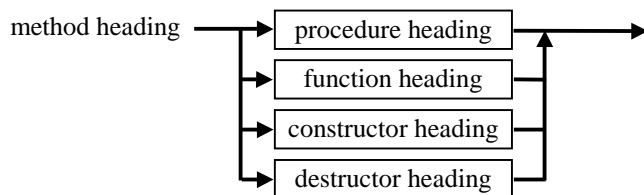
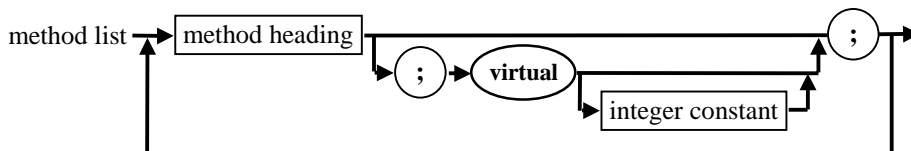
Each object and class type represents a unique set of components (the collection of members). Each component of an object or class is either a *field*, a *method* or for class types a *property*. A field stores data of a particular type, a method performs operations on the object and a property defines special attributes of a class and allows controlled access to its attributes. Two special types of methods, constructors and destructors, are used to construct (i.e. allocate memory for and initialise) and destruct (i.e. free memory used by) objects and classes.

An object or class type must be declared globally and can not be declared within the declaration part of a procedure or function. The declaration of an object or class type is very similar to that of a record with the addition that methods and properties can be declared.

For a summary of differences between the two object models, refer to the table on page 64.

Object type declarations





Example (from the interface section of OBJECTS.PAS):

```

type
{ TObject base object }

PObject = ^TObject;
TObject = object
  constructor Init;
  procedure Free;
  destructor Done; virtual;
end;

{ TStream }

PStream = ^TStream;
TStream = object(TObject)
  Status: Integer;
  ErrorInfo: Integer;
  constructor Init;
  procedure CopyFrom(var S: TStream; Count: Longint);
  procedure Error(Code, Info: Integer); virtual;
  procedure Flush; virtual;
  function Get: PObject;
  function GetPos: Longint; virtual;
  function GetSize: Longint; virtual;
  procedure Put(P: PObject);
  procedure Read(var Buf; Count: Word); virtual;
  function ReadStr: PString;
  procedure Reset;
  procedure Seek(Pos: Longint); virtual;
  function StrRead: PChar;
  procedure StrWrite(P: PChar);

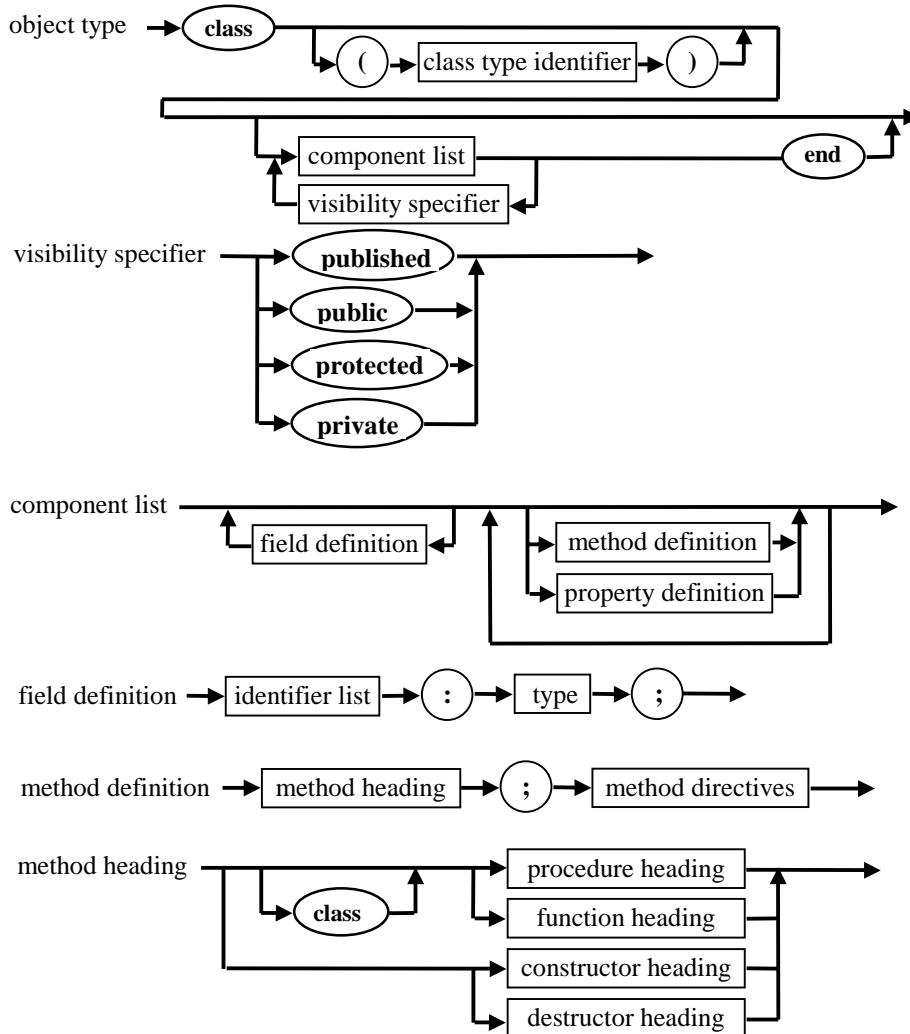
```

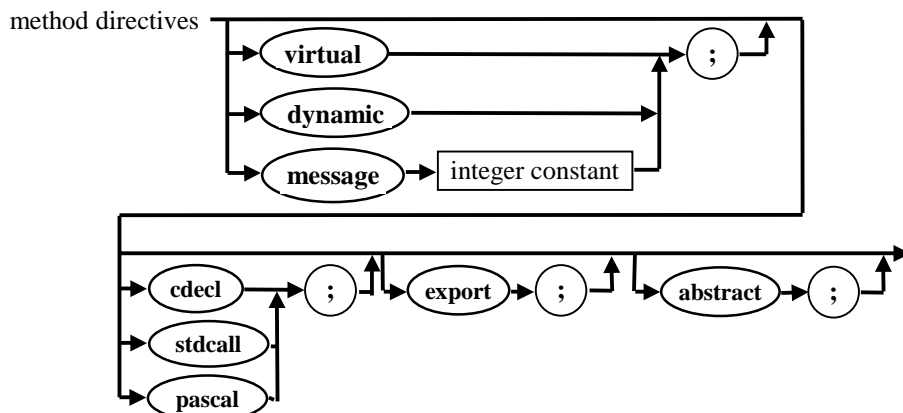
```

procedure Truncate; virtual;
procedure Write(var Buf; Count: Word); virtual;
procedure WriteStr(P: PString);
end;

```

Class type declarations





Example (from the interface section of *CLASSES.PAS*):

```

type
{ TFiler }

TValueType = (vaNull, vaList, vaInt8, vaInt16, vaInt32, vaExtended,
  vaString, vaIdent, vaFalse, vaTrue, vaBinary, vaSet);

TReaderProc = procedure(Reader: TReader) of object;
TWriterProc = procedure(Writer: TWriter) of object;
TStreamProc = procedure(Stream: TStream) of object;

TFiler = class(TObject)
private
  FStream: TStream;
  FBuffer: Pointer;
  FBufSize: Cardinal;
  FBufPos: Cardinal;
  FBufEnd: Cardinal;
  FRoot: TComponent;
  FAncestor: TPersistent;
  FIgnoreChildren: Boolean;
public
  constructor Create(Stream: TStream; BufSize: Cardinal);
  destructor Destroy; override;
  procedure DefineProperty(const Name: string;
    ReadData: TReaderProc; WriteData: TWriterProc;
    HasData: Boolean); virtual; abstract;
  procedure DefineBinaryProperty(const Name: string;
    ReadData, WriteData: TStreamProc;
    HasData: Boolean); virtual; abstract;
  procedure FlushBuffer; virtual; abstract;
  property Root: TComponent read FRoot write FRoot;
  property Ancestor: TPersistent read FAncestor write FAncestor;
  property IgnoreChildren: Boolean read FIgnoreChildren write FIgnoreChildren;
end;

{ TParser }

```

```

TParser = class(TObject)
private
  FStream: TStream;
  FOrigin: Longint;
  FBuffer: PChar;
  FBufPtr: PChar;
  FBufEnd: PChar;
  FSourcePtr: PChar;
  FSourceEnd: PChar;
  FTokenPtr: PChar;
  FStringPtr: PChar;
  FSourceLine: Integer;
  FSaveChar: Char;
  FToken: Char;
  procedure ReadBuffer;
  procedure SkipBlanks;
public
  constructor Create(Stream: TStream);
  destructor Destroy; override;
  procedure CheckToken(T: Char);
  procedure CheckTokenSymbol(const S: string);
  procedure Error(MessageID: Word);
  procedure ErrorFmt(const Ident: string; const Args: array of const);
  procedure ErrorStr(const Message: string);
  procedure HexToBinary(Stream: TStream);
  function NextToken: Char;
  function SourcePos: Longint;
  function TokenComponentIdent: string;
  function TokenFloat: Extended;
  function TokenInt: Longint;
  function TokenString: string;
  function TokenSymbols(const S: string): Boolean;
  property SourceLine: Integer read FSourceLine;
  property Token: Char read FToken;
end;

```

Object and class components

The declaration of a field specifies the field's identifier and data type. If a particular field of a class type is declared, each instance of that object or class will contain it.

Within a class type the declaration of a method is equivalent to a **forward** declaration of the method and must be resolved by a defining declaration of the method somewhere later in the same module.

In addition to fields and methods, class types (not object types) can also have properties. Properties are similar to object fields, but unlike them do not allocate any space. They are used to provide a pure implementation-independent interface for reading and writing a value of a particular type. The actual reading and writing may be performed directly to a host field of the class or via special access methods for reading and writing.

Properties can be used in expressions and be passed as value parameters, but cannot be passed by reference (**var** parameters).

Inheritance

An object or a class can be derived from an existing object/class type. The type from which it is derived is called the *immediate ancestor* of the new type. The immediate ancestor of the immediate ancestor is called an *ancestor* of the new type, as are all of *its* ancestors. The new type is called a *descendant* of its ancestors.

A descendant class can define new components in addition to those inherited from an ancestor type, but cannot remove the definition of components defined in an ancestor class. In the class model, a descendant class can redefine components defined by its ancestors. While this effectively hides the inherited component, the **inherited** keyword can be used to access the original component.

The *domain* of an object or class type consists of itself and all of its descendants.

If no ancestor type is specified in the definition of an *object* type, it has no ancestor type. However, all *class* types have the predefined class type *TObject* as the ultimate ancestor. If a declaration of a class type does not specify an ancestor type, the class type will inherit from *TObject*. *TObject* is declared in the *System* unit as follows:

type

```
TObject = class;  
TClass = class of TObject;  
TObject = class  
  constructor Create;  
  class function ClassInfo: Pointer;  
  class function ClassName: ShortString;  
  class function ClassNameIs(const Name: string): Boolean;  
  class function ClassParent: TClass;  
  function ClassType: TClass;  
  procedure CleanupInstance;  
  procedure Dispatch(var Message);  
  function FieldAddress(const Name: ShortString): Pointer;  
  procedure Free;  
  class function InheritsFrom(AClass: TClass): Boolean;  
  class procedure InitInstance(Instance: Pointer): TObject;  
  class function InstanceSize: Longint;  
  class function MethodAddress(const Name: ShortString): Pointer;  
  class function MethodName(Address: Pointer): ShortString;  
  { Virtual Methods: the order is significant }  
  procedure DefaultHandler(var Message); virtual;  
  class function NewInstance: TObject; virtual;  
  procedure FreeInstance; virtual;  
  destructor Destroy; virtual;  
end;
```

It is very important to mention that object and class types must not be mixed with each other. A class type can not have an ancestor of an object type and vice versa. The two object models exist independently of each other, although they both can be used in the same program.

Construction of object and class types

The declaration of an object or class type creates a unique type of which *instances* can be allocated. All instances of an object or class type share a single copy of every method (i.e. executable code), but have their own copies of the fields (i.e. data).

Instances of object types can be either static or dynamic. A static object variable is declared in the same way as variables of any other type, whereas a dynamic object is allocated at run time. As for other dynamic variables, the dynamic allocation is performed by a call to the *New* standard procedure, which should be passed a variable of type pointer to the object as the parameter. If the object type has a constructor method, it should be passed to *New* as a second parameter. Dynamic objects are disposed of by calling the *Dispose* standard procedure. If a destructor method is defined for the object type, it should be specified as the second parameter to *Dispose*. Note, that although object types without constructor or destructor methods are allowed they rarely occur, since constructors are used to initialise the object and destructors are used to perform any cleanup actions required.

Example

```

type
  POldStyleObject = ^TOldStyleObject;
  TOldStyleObject = object
    SomeField: Integer;
    constructor Init;
    destructor Done; virtual;
    ...
  end;
var
  XDyna: POldStyleObject;
  XStatic: TOldStyleObject;
begin
  XDyna := New(POldStyleObject, Init );      { or New(XDyna, Init); }
  XDyna^.SomeField := 2;
  ...
  Dispose( XDyna, Done );

  XStatic.Init;
  XStatic.SomeField := 2;
  ...
  XStatic.Done;
end.

```

Instances of class types are always allocated dynamically on the heap, without calling the *New* and *Dispose* standard functions when constructing and destroying class instances. Instead, the constructors and destructors are called directly. Class types always have the default constructor *Create* and destructor *Destroy* declared in *TObject*. A variable of a class instance type is an *implicit* pointer containing the address of the class instance and should not be declared as a pointer to the class type but simply as the class type. A variable of a class type contains either **nil** or a *reference* to an object of the class type. If it contains **nil**, it does not reference a class; otherwise, it points to the memory block allocated to the class instance. Two class type variables can refer to the same object, although only one of them should then be destroyed by calling the destructor method. To access components of a class type variable, the \wedge qualifier is implied and should not be specified.

TObject defines a method *Free*, which normally should be called instead of the *Destroy* destructor. *Free* checks if the class variable is **nil** and calls *Destroy* if it is not.

Example

```

type
  TNewStyleObject = class   { Implicitly has TObject as ancestor }
    SomeField: Integer;

```

```
    ...
  end;
var
  X : TNewStyleObject;
begin
  X := TNewStyleObject.Create;
  X.SomeField := 2;
  ...
  X.Free;
end.
```

Mutually dependent class types can be declared by using a forward class declaration. A forward declaration of a class type makes it possible to use the name of it in other class definitions before it has been fully defined and consists of a class name followed by the reserved word **class**. Later within the same type declaration block, the definition of the class must appear. For an example of a forward declaration see the declaration of the class *TObject* above.

Class references

If an object or a class type has any virtual methods, constructors or destructors, a structure uniquely identifying the object type (a Virtual Method Table, VMT) is created. Since class types always have *TObject* as the ultimate ancestor and *TObject* defines virtual methods, a constructor and a destructor, class types always have an associated VMT. The structure of the VMT is different for object and class types and is dependent on the compiler version. Class types store much more information than objects in their associated VMTs and this information can be accessed via some of the methods declared in *TObject*. This interface will not change in future versions of the compiler, with the possible exception of adding more methods.

The class model introduces another new concept - class reference types. Since class types always have an associated VMT, the VMT can uniquely identify *every* class type defined. The term *class reference* denotes a value of a class-reference type, which contains the address of the VMT for a class type. Class reference types are used to construct objects whose actual type is not known at compile time and this makes it possible to implement virtual constructors - something that was not possible in the old object model. *TClass* (in the block defining *TObject* above) declares a class reference type. Refer to page 63 for more information on class reference types.

Class methods

The defining declaration of a method always contains an implicit parameter with the name *Self*. *Self* represents the instance for which the method was activated. In addition to normal methods, class types may declare so-called *class methods*, which operate on the class reference instead of an instance of the class. It is possible to call a class method without constructing a particular instance of the class, although it is possible to call class methods from an instance of a class. The *ClassName* method of *TObject* is an example of a class method; it returns the name of the class as a *ShortString*.

To define a class method, the method definition must start with the reserved word **class**. In the defining declaration of the class method, the *Self* identifier represents the class for which the method was activated and does not represent an object reference. Thus, it is possible to use *Self* to call constructors and other class methods from within a class method, but it is not possible to use it to reference fields, properties or normal methods of the class.

Example

```
var
```

```

X: TObject;
Y: TClass;
begin
  X := TObject.Create;
  Writeln('X is a class instance of type ',X.ClassName);
  // Outputs 'X is a class instance of type TObject' on screen
  X.Free;

  X := TSomeClass.Create;
  Writeln('X is a class instance of type ',X.ClassName);
  // Outputs 'X is a class instance of type TSomeClass' on screen
  X.Free;

  Y := TSomeClass;
  Writeln('Y refers to the class named ',Y.ClassName);
  // Outputs 'Y refers to the class names TSomeClass' on screen
end.

```

Compatibility rules

An instance of an object type (or a pointer to an object type) can be assigned an instance of any of its descendent types (or pointer to any descendent type). A class type is assignment-compatible with any ancestor class type. During run time a class-type variable can reference an instance of the class type itself, an instance of any descendant type, or be **nil**.

Components and scope

The scope of a component identifier starts at the point of declaration and ends with the enclosing block. A component identifier is visible to all objects or classes derived from the given object or class type, to all methods of the object or class type, to field, method and property designators, as well as **with** statements that perform operations on variables of the class type.

It is possible to *redeclare* a component identifier declared in an object or class type in the block of a method declaration (by defining a local variable, for example). To access the component of the object or class rather than the local identifier, the implicit *Self* parameter can be used, for example *Self.SomeComponent*.

A method component identifier can be redeclared in an object or a class derived from the given type; in addition the class model allows a field and a property to be redeclared in a descendant class. The inherited component will be hidden after the redeclaration and can be accessed using the reserved word **inherited**.

Example

```

type
  TMyClassOne = class(TObject)
    FieldA: Integer;
    FieldB: Integer;
  end;
  TMyClassTwo = class(TMyClassOne)
    FieldA: AnsiString;
    function Test: Integer;
  end;

function TMyClassTwo.Test: Integer;

```

```
var
  FieldB: Boolean;
begin
  // Assign a value to local FieldB based on TMyClassOne FieldB field
  FieldB := (inherited FieldB > 0);
  // Assign a value to FieldA string based on TMyClassOne.FieldB field
  Self.FieldA := IntToStr(inherited FieldA);
  // Assign the function result based on the value of the values found
  Result := Length(FieldA) - ord(FieldB);
end;
```

Component visibility

A visibility specifier in the component section declaring the identifier defines the visibility of a component identifier. Visibility specifiers provide four levels of visibility: **published**, **public**, **protected** and **private**.

Public components

Component identifiers declared with the **public** attribute are visible to any code that has access to the object as a whole and do not have any particular restrictions on their visibility.

Published components

Component identifiers declared with the **published** attribute have no special restrictions on their visibility either. A published component acts as if declared as **public**, with the addition that run-time type information (RTTI) is generated for the component. Published component declarations are valid only in class types; no run-time type information is generated for object types. Run-time type information enables an application to dynamically query information about fields, properties and methods of a class type at run-time.

The **\$M** compiler directive controls the generation of run-time information for a class. Refer to page 158 for more information about the **\$M** compiler directive. A class type can have **published** sections only if it is compiled in the **{\$M+}** state or is derived from a class that was compiled in the **{\$M+}** state. Component identifiers which are declared right after class type heading have the **published** visibility attribute if the class type is compiled in the **{\$M+}** state and **public** in the **{\$M-}** state.

Field components can be declared as **published** only if they are of a class type. Fields of all other types must be declared as **public**, **protected**, or **private**.

Properties defined as **published** must be of an ordinal type, a real type (*Single*, *Double*, *Extended*, *Comp* or *Currency*, but not *Real*), a string type, a small set type (an integer sized set), a class type, or a method pointer type. A property defined as **published** can not be an array property.

Protected components

Component identifiers declared with the **protected** attribute are invisible to code that is not part of the implementation of methods of the class or its descendants. If a component of a class is for use only in the implementation of derived classes it should be declared as **protected**.

Private components

Component identifiers declared with the **private** attribute are invisible outside the module in which the class type is declared. Within the module containing the class type declaration, component identifiers can be accessed as if they were declared as **public**. If two related class types are declared in the same module, each of them will be able to access the private components of the other one, but these private components will be hidden from other modules.

While Borland uses **private** fields and methods widely in the Visual Component Library, we recommend the use of **protected** instead in order to increase usability of defined object types.

Static methods

All methods declared in an object or class are *static* by default. Static methods act just like normal procedures and functions. When a static method is called, the compile-time class or object type used in the method call indicates which method implementation should be invoked and the address of the method is determined at compile time. Since the compiler is able to determine the address of the method, it links the method directly and the dispatching of static methods is very quick compared to dispatching virtual and dynamic methods.

A static method does not change when inherited by another type. If a new type is derived from an object or class type containing a static method, it shares the same method located at the same address as the ancestor's type.

Static methods can not be overridden. If a static method with the same name as an inherited static method is declared in a descendant type, the inherited static method is replaced completely and cannot be accessed from the descendant object.

Virtual methods

To declare a method as *virtual*, the directive **virtual** should be added to the end of the method declaration. A virtual method is called just as any other method, but its dispatching mechanism is more flexible. When a virtual method is called, the run-time class or object type used in the method call indicates which method implementation should be invoked and the address of the method cannot be determined at compile time. The object looks up the address of the method at run time.

The **virtual** directive in a method declaration creates an entry in the VMT of the object or class. The VMT holds the addresses of all virtual methods in a class type and is used to determine the address when a virtual method is invoked.

An object type can override any virtual method declared by the ancestor type. The overridden method must be declared as **virtual** and must have exactly the same name, exactly the same number and order of parameters, their names and types and the same function result value, if any.

For class types, it is possible to override a virtual method in a descendant class type by adding the **override** directive to the end of the method declaration. The order, types and names of parameters and the type of the function result (if any) in the declaration of an override of a virtual method must be identical to those in the declaration of the ancestor's method. The scope of an override method includes the class type itself and all its descendants.

If a virtual method is declared in a descendant class type, with the same name as an inherited virtual method by specifying **virtual** instead of **override**, a new virtual method (hiding the inherited virtual method rather than overriding it) is defined. This is not possible in the old-style object model and is generally not advisable as it reduces the readability of the code (as illustrated by the example below). New virtual methods of an object type must have a name that is different from any virtual methods declared in any ancestor type.

Example 1 – using the class model

```
type
  TMyObjectOne = class
    destructor Destroy; override; // Overrides an existing virtual method
    function MyDynamic: Integer; virtual; // Define a new virtual method
  end;
```

```
TMyObjectTwo = class(TMyObjectOne)
  function MyDynamic: ShortString; virtual; // Redefines the method
end;
var
  MyObject: TMyObjectOne;
  MyInt: Integer;
  MyStr: ShortString;
begin
  // Call TMyObjectOne.Create (really calls TObject.Create)
  MyObject := TMyObjectOne.Create;
  // Call MyDynamic, which returns an Integer
  MyInt := MyObject.MyDynamic;
  ...
  // Call Free, which in turn calls TMyObject.Destroy
  MyObject.Free;

  // Call TObject.Create
  MyObject := TMyObjectTwo.Create;
  // Call MyDynamic, which returns a ShortString
  MyStr := MyObject.MyDynamic;
  ...
  // Call TMyObject.Free, which in turn calls TObject.Destroy
  MyObject.Free;
end;
```

Example 2 –using the object model

```
type
  PMyObjectOne = ^TMyObjectOne;
  TMyObjectOne = object
    function MyStatic: Integer;
    function MyDynamic: Integer; virtual;
  end;
  PMyObjectTwo = ^TMyObjectTwo;
  TMyObjectTwo = object(TMyObjectOne)
    function MyStatic: Integer;
    function MyDynamic: Integer; virtual;
  end;
var
  MyObject: PMyObjectOne;
begin
  MyObject := New(PMyObjectTwo);
  // Call TMyObjectOne.MyStatic
  WriteLn( MyObject.MyStatic );

  // Call TMyObjectTwo.MyDynamic
  WriteLn( MyObject.MyDynamic );

  // Free memory used by object
  Dispose( MyObjec );
end;
```

Dynamic methods

Dynamic methods differ from virtual methods only in the way dynamic method calls are dispatched at run time. A dynamic method does not contribute an entry in the VMT table, instead it assigns a number to the method and stores the address of the associated code in the Dynamic Method Table (DMT). Dynamic methods are very useful when object or class types declare a large number of virtual methods and there are a lot of descendants that override just a few of them. In the implementation of dynamic methods, the code size is smaller than the code size for the virtual methods, but the dispatching of dynamic methods is somewhat slower than the dispatching of virtual methods.

Dynamic methods of an object type are declared like virtual methods with the addition of a dynamic method index after the **virtual** standard directive. The dynamic method index must be a non-zero integer constant expression of type *Longint* and must be unique among the dynamic method indices of any ancestors.

const

```
cm_First = 0;  
cm_Exit = 20;
```

type

```
TMyObject = object(TObject)  
  procedure CMExit(var Msg: TMessage); virtual cm_First + cm_Exit;  
end;
```

As for virtual methods, the heading of an override of a dynamic method must list exactly the same identifier, parameters and function result type. It must also include the **virtual** standard directive with the same dynamic index.

To declare a method as dynamic in a class type, it is necessary to specify the **dynamic** directive instead of **virtual** at the end of the method declaration. It is not necessary to assign a dynamic index number manually; the compiler automatically assigns a unique number.

type

```
TMyClass = class (TObject)  
  procedure CMExit(var Msg: TMessage); dynamic;  
end;
```

Abstract methods

Abstract methods are used to define pure interface methods without providing any implementation. Using these, it is possible to declare just the heading of a method without a defining declaration; abstract methods are available only for class types. A virtual or dynamic method can be declared as **abstract**. Descendant class types that want to use it must override and fully define it.

To declare a virtual or dynamic method as abstract, the **abstract** directive should be added to the end of the method declaration. The **abstract** directive is only valid when a method is first introduced.

An abstract method can be overridden just as a virtual or dynamic method and it is not possible to call the ancestor's method from the implementation of the overriding method. An exception will be generated at run time if an abstract method is called through a class that has not overridden the method; if the *SysUtils* unit is not used, a run-time error 211 is generated instead.

Usually, abstract methods are used to introduce virtual methods in generic classes that are not meant to be instanced and where descendant classes are expected to implement the abstract methods. Since the method is defined (just not implemented) in the generic class, other methods of the generic class can call the abstract method as it would any other method.

type

```
TGenericClass = class (TObject)
  procedure SomeMethod; virtual; abstract;
end;
TSpecificClass = class (TGenericClass)
  procedure SomeMethod; override;
end;
```

Message handler declarations

Message handler methods are used to implement responses to *dynamically dispatched messages*. Message handling is available only in the class model.

A message handler method is always a procedure declared with the **message** directive in conjunction with an integer constant between 0 and *MaxLongInt*, specifying a *message ID*. A message handler method takes a single parameter, which must be any **var** parameter.

A typical message-handling method declaration looks like this:

type

```
TMyControl = class(TCustomControl)
  ...
  procedure WMSize(var Msg: TWMSize); message wm_Size;
end;
```

A message-handling method can be overridden by declaring a new method in a descendant class type with the same message index as the method it overrides. Using the **override** directive is not allowed in this instance. The name of the method and the type of the parameter do not have to match those of the overridden method.

Message handler implementations

The implementation of a message handler method looks exactly as that of a regular method.

Within the implementation of a message handler method, the implementation of an inherited method can be called. Since the name and parameter of the method may change, the **inherited** statement can be used by itself, without specifying the name of the ancestor method to call the inherited method. The invoked inherited method will be the first message handler with a matching index found in the most derived ancestor class. If the ancestor type does not declare a message handler for a particular message index, the **inherited** statement calls the *TObject* virtual method *DefaultHandler*. Since the *DefaultHandler* is available for any class, it is safe to call **inherited** within a message handler method, no matter whether the parent classes implement a handler for the message or not.

Example

type

```
TMyEditControl = class(TCustomControl)
  ...
  procedure WMSize(var Msg: TWMSize); message wm_Size;
end;
TMyOtherEditControl = class(TMyControl)
  ...
  procedure AnotherName(var Parm); message wm_Size;
end;
```

```
procedure TMyEditControl.WMSize(var Msg: TWMSize);
```



```

begin
  inherited;           // Calls TCustomControl or other ancestor handler
  ...
end;

procedure TMyOtherEditControl (var Parm);
begin
  inherited;           // Calls the TMyControl.WMSize method
  ...
end;

```

Message dispatching

Message methods are rarely called directly. Instead the message handler methods are usually dispatched to a class using the *Dispatch* method inherited from *TObject*. The declaration of *Dispatch* is:

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

The *Message* parameter is a record, the first field of which must be of *Cardinal* type and contain the message index for the method to be dispatched. In addition to this field, any number of extra fields can be defined to hold message specific information.

When *Dispatch* is called, it first checks whether the list of message handlers declared for the class itself contains a handler for the given message index. If not, *Dispatch* checks the list of message handlers of the ancestor type, ancestor's ancestor type and so on until it either finds a matching handler or reaches *TObject*. In the latter case, *Dispatch* will invoke the *DefaultHandler* method declared in *TObject*. The declaration of the *DefaultHandler* is:

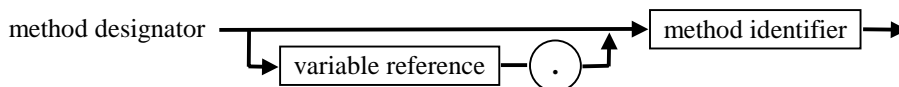
```
procedure DefaultHandler(var Message); virtual;
```

The *DefaultHandler* does nothing - it simply returns and it is possible to override *DefaultHandler* to implement a different default handling of messages. *DefaultHandler* is also called in case a message handler executes an **inherited** statement for which no inherited message handler is defined.

The most common use of message methods is in message handler loops in OS/2 or Windows programs, where they are used to transfer control based on messages sent to the program by the operating system.

Method activations

A function or procedure statement, consisting of a *method designator* followed by an actual parameter list, activates a method:

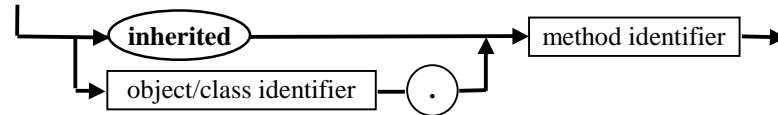


The variable reference denotes either an instance of an object or class type, or a class reference. The method identifier denotes a method of that object or class type. Only class methods and class constructors can be called by means of a class reference. An instance of an object or class becomes an implicit actual parameter of the method, corresponding to a formal parameter named *Self*. For class methods, *Self* holds a class reference (the address of the VMT for the class).

Within a **with** statement referencing an object or a class, a method designator can consist of a method identifier only, since the **with** statement will supply the implicit *Self* parameter for the method call.

Within a method, a qualified method designator can be used to denote a particular method. This type of activation is known as *qualified method activation*.

qualified method designator



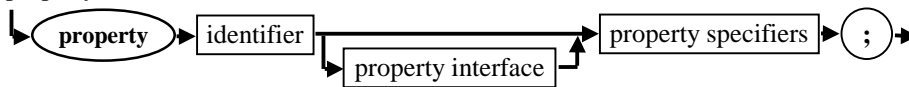
The object or class type specified in the qualified method activation must denote an object or class of the method or any of its ancestors. In this case even virtual and dynamic methods behave as if they were static, as the method to be invoked can be determined at compile time by the qualified identifier specified. A qualified method identifier can also be used as an operand of an @ address operator to query the address of the method's entry point.

Properties

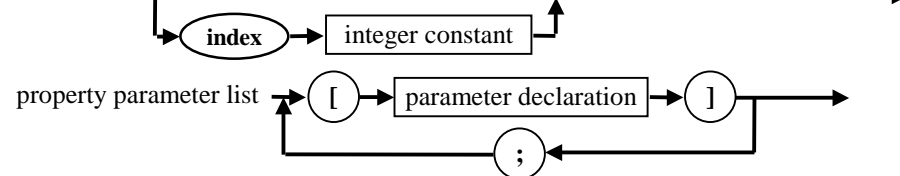
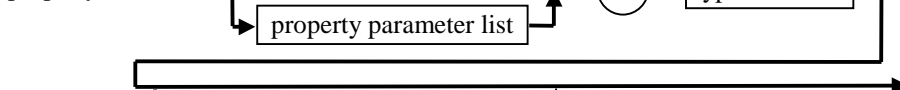
In addition to fields and methods, class types can also have properties. A property definition in a class declares a named attribute and the actions associated with reading and writing the attribute.

Properties can be considered to be an extension of fields that allows the operations for getting the field value (reading) and assigning a new value to it (writing) to be redefined in descendant classes. Unlike fields, properties do not allocate any space in the class instance, as they only provide an interface for accessing the *attribute* of a specified type by associating *actions* with reading and writing the attribute.

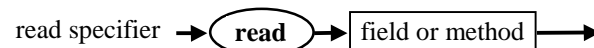
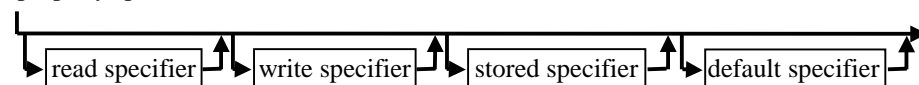
property definition

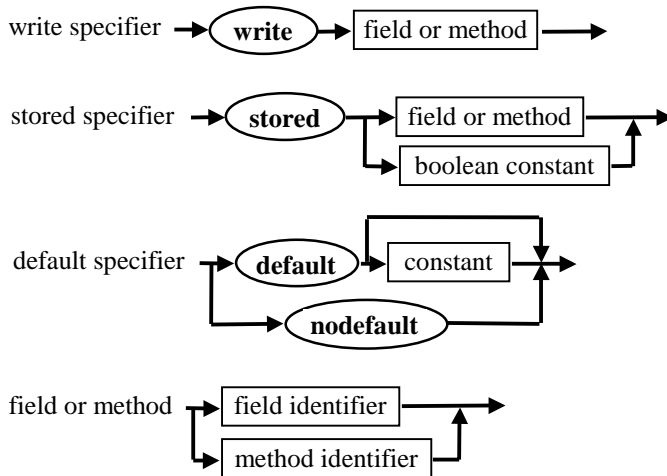


property interface



property specifiers





The declaration of a property requires:

- The name of the property
- The type of the property
- Methods to read and/or set the value of the property

A property can be of any type that a function can return (since the implementation of the property can use a function).

Example

type

```

TPosControl = class(TCustomControl)
  FPosition: Integer;
  procedure SetPosition(Value: Integer);
  property Position: Integer read FPosition write SetPosition default 0;
end;

```

The read and write specifiers of a property can refer directly to a field of the class that is of the same type as the property. Alternatively, the specifiers can refer to access methods that are called when the value of the property is accessed or set.

When field specifiers are used, reading and/or writing the property value is equivalent to reading and writing the field value itself, with the important difference that the implementation can be changed to use an access method later without having to change any code using the class.

For simple properties, the **read** part of the property is usually a field reference and the **write** part is accessed by an access method, as is the case in the example above.

In a program, properties acts like fields of a class. If used as a factor in an expression, a property returns its value. It is also possible to assign a new value to a property by means of an assignment statement. However, a property can not be passed as a **var** parameter and it is illegal to take the address of a property using the @ operator, even if both the **read** and **write** specifiers list a field identifier. This ensures that any descendants of the class type may change one or both access specifiers of the property to list a method.

Access specifiers

The **read** and **write** parts of a property define the way property data can be accessed. The syntax for property declarations allows the **read** and **write** parts of a property declaration to specify either an access method instead or a class field.

If a class field is specified as the access specifier, the field type must be the same as the property type.

If a **read** specifier lists a method, it must be a function that takes no parameters and returns a value of the same type as the property. The **read** method implements the logic required to produce the value of the property. If a **read** specifier is not declared, the property is *write-only*. Because it is an error to use a *write-only* property as a factor in an expression, write-only properties are only rarely useful.

If a **write** specifier lists a method, it must be a procedure that takes a single value or constant parameter of the same type as the property. The value passed in the parameter should be used to set a new value of the property and the **write** method should take care of any manipulation needed to put the appropriate values into internal storage. If a **write** specifier is not declared, the property is *read-only*. An attempt to write to a *read-only* property causes a compiler error to be reported. If both a **read** and a **write** specifier is declared, the property is a *read-write* property.

Array properties

Array properties can be defined as properties that have multiple values of the same type referred to by one or more indices.

The declaration of an array property should specify the name of the property, the name(s) and type(s) of the index (indices) as well as the type of the elements.

Example:

type

```
TCursorControl = class
```

protected

```
// ... some fields are probably defined here
```

```
function GetCursors(Index: Integer): HCursor;
```

```
procedure SetCursors(Index: Integer; NewValue: HCursor);
```

published

```
property Cursors[Inx: Integer]: HCursor read GetCursors write SetCursors;
```

```
end;
```

The declaration of a property can specify indices of any type (even *String*), not just ordinal types like the index type in an **array** type declaration.

The **read** and **write** parts of an array property declaration must be methods and must not directly refer to class fields.

The **read** method of an array property must be a function that takes a set of parameters of the same number, the same type and in the same order as the indices specified in the property declaration. The function must return a value of the same type as the property elements.

The **write** method for an indexed property is a procedure that takes a set of parameters of the same number, the same type and in the same order as the indices specified in the property declaration along with an additional value or constant parameter of the same type as the property elements.

In a program, an array property can be treated just as if it was an array, with the exception that only individual elements of an array property can be referenced. Referencing the entire range of the property is not allowed.

An array property can be declared as the *default array property*. This allows the array property to be accessed without specifying its name - the class instance followed by the indices enclosed in square brackets is enough.

To declare a default array property the **default** directive should be appended to the definition of the array property:

```

type
  THash = class
    function DoSearch(const Key: String): String;
    property Search[const Key: String]: String read DoSearch default;
  end;
...
var
  Hash: THash;
  S1,S2: String;
begin
  S2 := Hash.Search[S1];    // These assignments are equivalent
  S2 := Hash[S1];
end;

```

A class can have only one default property. When a class type declaration contains a default array property, all descendants of the class type inherit the default array property and the default array property can not be redeclared or hidden in descendant classes.

Index specifiers

Index specifiers are used by properties that share the same access method. To declare a property with an index specifier, the **index** directive followed by an integer value with the range from -2147483647 to 2147483647 should be added to the property declaration.

Example:

```

type
  TStringNum = class
    protected
      FNumber: Integer;
      function GetStr(Index: Longint) : String;
      procedure SetStr(Index: Longint; Value: String);
    published
      property HexStr: String index 0 read GetStr write SetStr;
      property DecStr: String index 1 read GetStr write SetStr;
      property Number: Integer read FNumber write FNumber;
    end;

function TStringNum.GetStr(Index: Longint) : String;
var
  FormatType: String[2];
begin
  case Index of
    0 : FormatType := '%x';

```

```
    I : FormatType := '%d';  
end;  
Result := Format( FormatType, [FNumber] );  
end;  
  
procedure TStringNum.SetStr(Index: Longint; Value : String);  
begin  
    if Index = 0 then  
        Value := '$'+Value;  
        FNumber := IntToStr(Value);  
    end;  
end;
```

The **read** and **write** parts of the declaration of a property with an index specifier must be methods and may not list class fields.

When referencing a property with an **index** specifier, the index specifier is passed to the access method as an additional parameter. The index parameter should be a value parameter of type *Longint* and must be added to the parameter list of the access methods of a property. The **read** method must have it as the last parameter and the **write** method must have it as the second to last parameter.

Storage specifiers

A non-array property definition can include a *storage specifier*: **stored**, **default** or **nodefault**. The purpose of these specifiers is to set a special field in the run-time type information for the class. It is up to the program to choose what to do with this information. The Delphi Visual Component Library uses this information for saving and loading property values in a form file.

By default, all properties in the **published** part of the object declaration are stored. It is possible to choose not to store a particular property at all, or designate a function for determining at run-time whether to store the property.

To control whether a property is stored, the **stored** directive should be added to the property declaration, followed by *True*, *False*, or the name of a *Boolean* field or method.

The **default** and **nodefault** directive can be used only with properties of ordinal types or small set types.

The **default** directive is used to declare a default value for a property. The default value is a constant of the same type as the property and is used by Delphi to determine whether or not to store a property in a form file. If a default value is not specified, the property will always be saved.

When redeclaring a property, the property can be declared to have no default value, even if the inherited property specified one. To declare a property as having no default value, the **nodefault** directive should be added to the property declaration. If neither **default** nor **nodefault** is specified in a property definition, the default setting of **nodefault** is assumed.

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

To determine whether the **stored** attribute of a property is set, the *IsStoredProp* function declared in the *TypeInfo* unit can be used.

Property overrides

A class type inherits all properties of its ancestor types. To change the visibility, access specifiers or storage specifiers of an inherited property in a derived class, the property should be overridden. An overriding declaration of a property does not include the property interface.

To change just the visibility of a property, the reserved word **property** followed by an inherited property name should be specified. Note, that overrides can only make properties less restricted, not more restricted. In other words, a **protected** property can be made **public**, but redeclaring it as protected can not hide a property declared as **public** in an ancestor class.

When overriding a property, it is possible to declare new access specifiers, new default values and a new storage specifier.

Example:

```

type
  TPublicStringNum = class(TSomeClass)
  public
    property HexStr;
    property DecStr;
    procedure SetNumber(Value: Integer);
    property Number write SetNumber;
  end;

```

In this example, the visibility of the *HexStr* and *DecStr* properties is raised and the **write** access specifier of the *Number* property is changed to be a method (from whatever it was in the ancestor class, where it might also have been a method).

Class-reference types

Class-reference types are used to perform operations directly on class types, as opposed to class types, which operate on class instances. Class reference types are available only when using the class model. The syntax of a class reference type declaration is:

class reference type → (class) → (of) → [object type identifier] →

A class reference type is used by a virtual constructor to create objects whose actual type is unknown at compile time and by a class method to perform an operation on a class whose actual type is unknown at compile time. It can be used as the right hand side operand of an **is** operator to perform a dynamic type check with a type that is unknown at compile time, can be used as the right operand of an **as** operator to perform a checked typecast on a type that is unknown at compile time, and can be used to call class methods

A class reference type is assignment-compatible with any ancestor class reference type, so it can reference the class or any class derived from the class it was defined for.

A class type identifier functions as a class reference type value. A class reference type variable can be **nil**, in which case it does not reference a class. A value of a class reference type is encoded as a pointer to the VMT for the class. The *ClassType* standard method declared in *TObject* is used to receive a class reference value from a class instance, like this:

```

type

```

```

TMyObject = class;
TMyClass = class of TMyObject;
...
var
  One: TMyObject;
  Two: TObject;
  C: TMyClass;
begin
  One := TMyObject.Create;
  C := One.ClassType; // Get the class of object One
  if C is TMyClass then // Always true, but this is an example :-)
    Two := C.Create // Create another object of the same type
  else
    Two := TObject.Create // Create another object of type TObject
end;
```

When a constructor is invoked on a variable reference of a class reference type, it is used to create an object, the type of which is unknown at compile time. Constructors that are invoked through class reference types are usually **virtual**. If this is the case, the constructor implementation called depends on the run-time class type selected by the class reference.

Summary of the two object models

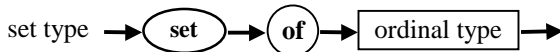
The following table summarises the most important features of the two object models.

Feature	object types	class types
Default ancestor <i>TObject</i>	No	Yes
Static instances	Yes	No
Dynamic instances	Yes	Yes
Dynamic instances must be dereferenced	Yes	No
Abstract methods	No	Yes
Message handlers	No	Yes
Dynamic methods	Yes	Yes
Virtual constructors	No	Yes
Class methods	No	Yes
Properties	No	Yes
Class reference types	No	Yes
Guarded type checking using is	No	Yes
Protected typecast using as	No	Yes
New instance is automatically cleared	No	Yes
Run-time type information	No	Yes
private , public and protected components	Yes	Yes
published components	No	Yes
Compatible with Borland Pascal	Yes	No
Compatible with Borland Delphi	Yes	Yes

Set types

A set is a collection of elements of the same ordinal type, called the base type. If **S** is a set of elements of type **T**, then any element of type **T** is either a member of **S** or is not a member of **S**. Values of a set type are subsets of values of the base type. A set may have all the members of the base type or it may have no members, in which case it is called an *empty* set and is written [].

The following diagram defines the syntax for a set type:



A set value is stored as a string of bits. Each bit in the string corresponds to a value of the base type. If a bit is set, the corresponding element is in the set. The maximum size of a set variable is 32 bytes, which limits the base type of any set to 256 elements and limits the lower and upper bounds to the 0..255 range.

This means that an enumerated base type must not contain more than 256 values and integer base types must be a subrange of the *Byte* type. Type *Char* is also a valid base type for a set. The number of bytes occupied by a set, the byte number of a particular element and the bit number within that byte, are calculated using the following formulas:

$$\begin{aligned} \text{Size} &= (\text{High}(\text{BaseType}) \text{ div } 8) - (\text{Low}(\text{BaseType}) \text{ div } 8) + 1 \\ \text{ByteNo} &= (\text{Ord}(\text{Element}) \text{ div } 8) - (\text{Low}(\text{BaseType}) \text{ div } 8) \\ \text{BitNo} &= \text{Ord}(\text{Element}) \text{ mod } 8 \end{aligned}$$

Some examples of set types:

type

```
CharSet = set of Char;
Alphabet = set of 'A'..'Z';
WeekPlan = set of DayOfWeek;
```

The following operators can be applied to set type values:

+ * - = <> <= >= in

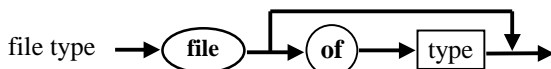
The standard procedures *Include* and *Exclude* can be used to include or exclude an element of the base type. They are implemented as an optimised form of the following assignment operators:

Include(SetVar, Element) \Leftrightarrow SetVar := SetVar + [Element];

Exclude(SetVar, Element) \Leftrightarrow SetVar := SetVar - [Element];

File types

File types are special types that are used to access external files or devices. A file is a linear sequence of components of a specific type, called the based type. The base type can be any type except a file, object or class type or can be a structured type not containing fields of any of these types.



Using standard procedures and functions, all basic input and output (I/O) operations on file types can be performed, such as opening, reading or writing, setting the file position, closing, renaming, deleting, etc.

Pascal files fall into three categories:

Typed files

If the reserved word **of** and a component type are present in the declaration of a file type, the file is assumed to be typed and it is only possible to read and write components of the declared component type. The *Seek* standard procedure and the *FileSize* and *FilePos* standard functions assume the file position and file size to be calculated in component type unit, i.e. not necessarily in bytes.

Untyped files

Untyped files are declared using the reserved word **file**. No component type is associated with untyped files. Untyped files are used for low level access to binary files and allow only the most basic I/O operations of reading and writing blocks of data to be performed. The base blocksize can be specified when the file is opened by the *Reset* standard function. If the base size is not specified in the call to *Reset*, it assumes a default value of 128 bytes.

Text files

Text is a predefined text file type. Text files are used to access MS-DOS style text files, where Carriage Return (ASCII 13) and Line Feed (ASCII 10) characters are used as line delimiters. *Text* is declared in the *System* unit as follows:

type

Text = **file of Char**;

Unlike other typed files with *Char* as the base type, *Text* files have special features:

- It is possible to Read and Write elements that are not of type *Char*. These will automatically be converted from/to character representation.
- Text file I/O is buffered using either an internal buffer of 128 characters contained in the file variable or an external one attached by means of the *SetTextBuf* standard procedure.
- Special driver functions are used for opening, closing, flushing, reading and writing and offer a flexible and powerful mechanism for creating a custom user defined low level I/O interface.

The following table lists all standard procedures and functions that can be used with each of the file types. They are combined into groups according to the function performed:

Name	Typed	Untyped	Text	Description
<i>Assign</i>	+	+	+	Associates the name of an external file with a file variable
<i>Append</i>	-	-	+	Opens an existing file for appending
<i>Reset</i>	+	+	+	Opens an existing file
<i>Rewrite</i>	+	+	+	Creates a new file and opens it
<i>Erase</i>	+	+	+	Deletes an external file
<i>Rename</i>	+	+	+	Renames an external file
<i>Read</i>	+	-	+	Reads one or more records/values from a file
<i>ReadLn</i>	-	-	+	The same as <i>Read</i> , but skips to the beginning of the next line after the read
<i>Write</i>	+	-	+	Writes one or more values to a file
<i>WriteLn</i>	-	-	+	The same as <i>Write</i> and appends a line delimiter
<i>BlockRead</i>	-	+	-	Reads one or more records from a file
<i>BlockWrite</i>	-	+	-	Writes one or more values to a file
<i>Flush</i>	-	-	+	Flushes the buffer of a text file
<i>Seek</i>	+	+	-	Moves the current position to the specified record
<i>SeekEof</i>	-	-	+	Returns <i>True</i> if end of file is reached (ignores spaces at the end)
<i>SeekEoln</i>	-	-	+	Returns <i>True</i> if end of line is reached (ignores spaces at the end)
<i>Eof</i>	+	+	+	Returns <i>True</i> if end of file is reached
<i>Eoln</i>	-	-	+	Returns <i>True</i> if end of line or end of file is reached

Name	Typed	Untyped	Text	Description
<i>FileSize</i>	+	+	-	Returns the size of the file
<i>FilePos</i>	+	+	-	Returns the size of the file
<i>Truncate</i>	+	+	-	Truncates a file at the current file position
<i>SetTextBuf</i>	-	-	+	Associates an external I/O buffer with a file
<i>Close</i>	+	+	+	Closes an open file
<i>IOResult</i>	+	+	+	Returns 0 if last I/O operation was successful or a non-zero error code otherwise

Internally, file types are stored as records. Typed and untyped file records occupy 332 bytes with the following structure:

type

```
TFileRec = record
  Handle: Longint;
  Mode: Longint;
  RecSize: Longint;
  Private: array [1..28] of Byte;
  UserData: array [1..32] of Byte;
  Name: array [0..259] of Char;
end;
```

Text files occupy 460 bytes with the following layout:

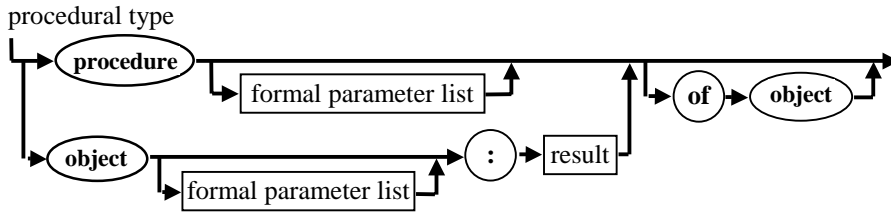
type

```
TTextBuf = array [0..127] of Char;
TTextRec = record
  Handle: Longint;
  Mode: Longint;
  BufSize: Longint;
  BufPos: Longint;
  BufEnd: Longint;
  BufPtr: ^TextBuf;
  OpenFunc: Pointer;
  InOutFunc: Pointer;
  FlushFunc: Pointer;
  CloseFunc: Pointer;
  UserData: array [1..32] of Byte;
  Name: array [0..259] of Char;
  Buffer: TTextBuf;
end;
```

These records are declared in the *Dos* unit as *FileRec* and *TextRec* and in *WinDos* and *SysUtils* units as *TFileRec* and *TTextRec*.

Procedural types

Procedural types are pointers, which enable the use of procedures and functions as parameters both in procedure and function calls and as variable values. The syntax diagram for a procedural type:



A procedural type declaration is similar to a procedure or function heading without a procedure or function identifier.

Procedural types may be of two forms: *global procedure pointers* and *method pointers*.

A *global procedure pointer* can be thought of as an address of a global procedure or function, usually in the code segment, where the executable code of the procedure or function is stored; that is, the address to which control is transferred when the procedure or function is called. The declaration of a global procedure pointer does not contain the clause **of object**. Examples of global procedure pointer types follow:

type

```
TMathProcedure = procedure (var X, Y, Z: Integer );
```

```
TStatFunction = function( var X, Y: Extended): Extended;
```

```
TReportFunc = function( var Min, Max: Extended; SF: StatFunction): Extended;
```

A *method pointer* is as an address of a procedure or function method of a **class**. It is implemented as *two* pointers, one of which stores the address of a procedure or function method of a class, the other contains the address of a corresponding class instance. The declaration of a method pointer must contain the clause **of object**. For example:

type

```
MathMethod = procedure of object;
```

```
MathProcedure = procedure (P: MathObject ) of object;
```

A variable of a procedural type can have one of the following values:

- The **nil** value.
In this case the procedural variable is considered to be unassigned and using it as a procedure or function call will result in an error. The standard function *Assigned* returns *True* if a procedural variable has been assigned a value and *False* if it has not.
- Pointer to a variable of a procedural type.
- A procedure or function identifier (for global procedural pointers).
- A method identifier (for method pointers).

Procedural type compatibility

Procedural types are compatible if they use the same calling convention and their parameters are of the same number, in the same order and of the same types; the result types of functions must also be identical.

A value of **nil** is compatible with any procedural type.

Global procedure and method pointer types are always mutually incompatible, i.e. a global procedure or function can not be a value of a method pointer variable and a method can not be assigned to a global procedure pointer variable.

A nested procedure or function is one declared within another procedure or function. Nested procedures and functions, standard procedures and functions and **inline** procedures and functions can not be used as procedural values.

To be used as a procedural value, a standard procedure or function (declared in the *System* unit) must be enclosed in a user-defined procedure or function.

Example

```

type
  TUpCase = function(ch: Char): Char;

function TestFunc (_UpCase: TUpCase; ch: Char): Char;
begin
  if Assigned(_UpCase) then
    Result := _UpCase(ch)
  else
    Result := ch;
end;

function MyUpCase (ch: Char): Char;
begin
  Result := System.UpCase(ch)
end;
var
  TheFunction : TUpCase;
begin
  // Output 'A' on screen
  Writeln( TestFunc( MyUpCase, 'a' ));
  // Output 'A' on screen
  TheFunction := @MyUpCase;
  Writeln( TestFunc( TheFunction, 'a' ));
  // Output 'a' on screen
  Writeln( TestFunc( nil, 'a' ));
end.

```

Type identity

Type identity is required between an actual and a formal variable parameter in procedure and function calls.

Two types are identical if one of the following is true:

- They are declared with the same type identifier
- The first type is declared to be equivalent to a type identical to the second type

The second condition means that the first type does not have to be declared directly to be equivalent to the second type. For example, the following declaration:

```

T1 = Integer;
T2 = T1;
T3 = T2;

```

makes T_1 , T_2 and T_3 identical types. The type declarations:

```

T4 = array [0..3] of Byte;

```

$T_5 = \mathbf{array} [0..3] \mathbf{of} \mathit{Byte}$;

do not make T_4 and T_5 identical because $\mathbf{array} [0..3] \mathbf{of} \mathit{Byte}$ is not a type identifier.

Two variables are of identical types if they are declared in the same declaration:

```
var
  V1, V2: record
    X, Y: Integer;
end;
```

Type compatibility

Compatibility between two types is required, for example, in expressions or in relational operations. Type compatibility is a precondition of assignment compatibility.

Two types are compatible if at least one of the following conditions is true:

- Both types are the same.
- Both types are floating point types.
- Both types are integer types.
- One type is a subrange of the other.
- Both types are subranges of the same host type.
- Both types are set types with compatible base types.
- Both types are packed string types with an identical number of components.
- One type is a string type and the other is either a string type, packed string type, or *Char* type.
- One type is *Pointer* and the other is any pointer type.
- Both types are class types or class reference types and one type is derived from the other.
- One type is *PChar* and the other is a zero-based character array of the form $\mathbf{array}[0..N] \mathbf{of} \mathit{Char}$ (applies only when extended syntax is enabled - the $\{\$X+\}$ state).
- Both types are pointers to identical types (applies only when type-checked pointers are enabled - the $\{\$T+\}$ state).
- Both types are procedural types with identical result types, an identical calling convention, an identical number of parameters and one-to-one identity between parameter types.

Assignment compatibility

Assignment compatibility is necessary when a value is assigned a value and is also required between an actual and a formal variable parameter in procedure and function calls.

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 conditions are true:

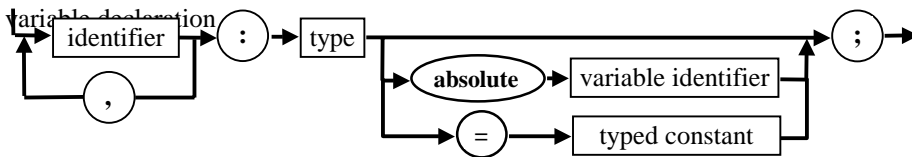
- T_1 and T_2 are identical types and neither is a file type or a structured type that contains a file-type component at any level of structuring.
- T_1 and T_2 are compatible ordinal types and the values of type T_2 falls within the range of possible values of T_1 .

- 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 floating point type and T_2 is an integer type.
- T_1 and T_2 are string types.
- T_1 is a string type and T_2 is a *Char* type.
- T_1 is a string type and T_2 is a packed string type.
- T_1 and T_2 are compatible, packed string types.
- T_1 and T_2 are compatible set types and all the members of the value of type T_2 fall within the range of possible values of T_1 .
- T_1 and T_2 are compatible pointer types.
- T_1 is a class type and T_2 is a class type derived from T_1 .
- T_1 is a class reference type and T_2 is a class type derived from T_1 .
- T_1 is an object type and T_2 is an object type derived from T_1 .
- T_1 is a pointer to the object type O_1 and T_2 is a pointer to the object type O_1 or to any object type derived from O_1 .
- T_1 is a *PChar* and T_2 is a string constant (applies only when extended syntax is enabled - the **{SX+}** state).
- T_1 is a *PChar* and T_2 is a zero-based character array of the form **array[0..N] of Char** (applies only when extended syntax is enabled - the **{SX+}** state).
- 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 calling convention, an identical number of parameters and one-to-one identity between parameter types.

Variables and typed constants

Variable and typed constant declarations

A variable is an identifiable region of memory that holds a value (or a set of values) that can change. Each variable has an associated identifier and type (also known as a data type). All variables must be declared in a variable declaration that has the following syntax:



An identifier list contains a single identifier or a list of identifiers, separated by commas. An identifier, specified in the identifier list, can be used to access the variable it identifies within the block containing the declaration. If an enclosing block contains an inner block, an identifier declared in the enclosing block can be redeclared within the inner block. In this case, the new variable will use this identifier, but after the end of the inner block, the identifier will represent the variable declared in the enclosing block.

The type associated with a variable can be any valid standard data type or a user-defined type. A user-defined type identifier should be declared in a **type** declaration part prior to it being used or may be a new type declaration.

In the **{&Delphi+}** state, a global variable can be given an initial value by specifying an equal sign followed by a constant expression, using the same syntax as described for typed constants below. Local variables (defined inside a procedure, function or method), cannot have an initial value; upon entry the value of all local variables is undefined.

An example of a variable declaration part:

var

I, J, M, N: Integer;

Value: Extended;

B: ^Byte;

Statement: array[0..100] of Char;

Sentence: string[150]

X: Integer = 8;

Table: TMatrix;

DSchedule: TScheduleRec;

ValRec: record

SignificandLo: Longint;

SignificandHi: Longint;

Exponent: SmallWord;

end absolute Value;

Point: record

X, Y: Integer;

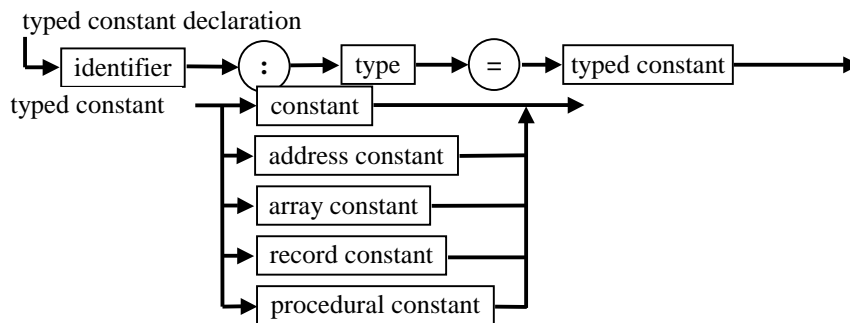
end = (X: 2; Y: 10);

The reserved word **absolute** followed by a variable identifier is used for declaring *absolute* variables. These are variables residing at the same memory address as another variable. This is called an *absolute-equivalence*. If an **absolute** clause is present, only one identifier must be specified.

Note

Unlike Borland Pascal, variables at absolute addresses are not supported, since absolute addresses have no meaning in protected mode.

Typed constants can be thought of as variables with an initial value and can be used irrespective of the state of the **{&Delphi}** directive, as they are compatible with both Borland Pascal and Borland Delphi. They are initialised once at the beginning of a program and can be used as normal variables. In addition to the type, a typed constant declaration contains the initial value:



Simple-typed constants

A simple-typed constant declaration defines a simple type and the initial value of the constant. For example:

const

LowBound: Integer = 1;

Delta: Extended = 0.01;

Typed constants can be initialised by normal constant expressions or by *constant-address expressions*. A constant-address expression is an expression which takes the address, offset, or segment of a typed constant, a global variable, a procedure or a function. For example:

var

TextLine: array[0..255] of Char;

const

TextLineOfs: Longint = OfS(TextLine);

Typed constants can not always be used instead of untyped constants. For example, it is not allowed to use a typed constant in the declaration of other constants or types.

String-type constants

A string-type constant declaration either defines the maximum length of the string and its initial value (when declaring a *ShortString* constant) or specifies the *AnsiString* type followed by an initial value. For example:

const

```
TextString: string[4] = 'text';  
LongString: AnsiString = 'This is a typed long string constant';
```

Structured-type constants

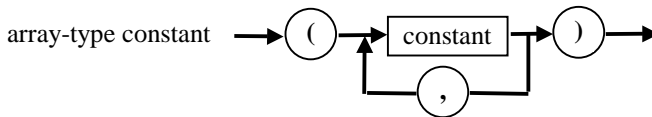
A structured-type constant declaration defines the values of the components of a structured-type constant. Virtual Pascal supports the declaration of array, record, object, class and set-type constants.

const

```
MyRec : record x, y: Integer; end  
= (x: 2; y: 8);
```

Array-type constants

The following diagram shows the declaration of an array-type constant:



Example

const

```
BitMasks = array [0..7] of Byte = ($01,$02, $04, $08, $10, $20, $40, $80);
```

When a multidimensional-array type constant is initialised, the constants of each dimension should be enclosed in parentheses and separated by commas. The innermost constants correspond to the rightmost dimensions.

Example

type

```
TCcoef = array[1..2, 1..2] of Integer;
```

const

```
Coef: TCcoef = ((1, 2), (3, 4));
```

that is equivalent to the following assignments at the beginning of the program:

```
Coef[1,1] := 1;  
Coef[1,2] := 2;  
Coef[2,1] := 3;  
Coef[2,2] := 4;
```

An array-type constant can have components of any type except file types. Packed string type constants can be initialised both by single characters and by a string constant expression.

Example

const

```
WildChars: array [0..1] of Char = ('*', '?');
```

or using a string constant expression:

const

```
WildChars: array [0..1] of Char = '*?';
```

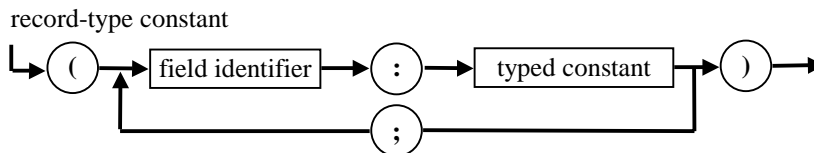
In the **{SX+}** state, a zero-based character array can be initialised by a string constant expression whose length is less than the dimension of the array. In this case the remaining characters will be set to null characters (#0) and the array will contain a null-terminated string.

Example

```
const
  FileName = array[0..259] of Char = 'Long_File_Name.PAS';
```

Record-type constants

The following diagram shows the declaration of a record-type constant:



Example

```
type
  TPoint = record
    X, Y: Integer;
end;
TPersonRec = record
  Name: string[40];
  Surname: string[40];
  Age: Integer;
end;

const
  InitPos: TPoint = (X:1; Y: 10);
  PersonRec: TPersonRec = (Name: 'Mary'; Surname: 'Smith'; Age: 30);
```

A record-type constant can not have components of a file type. The record fields must be initialised in the same order as they were declared in the record-type. For records with a variant part, only the active variant can be initialised and if there is a tag field its value must also be defined.

Object-type constants

Object-type constants have the same syntax as record-type constants. Method components are not initialised.

Here is the example of object-type constants (see page 43 for the earlier object-type declaration):

```
const
  DataStream: TDataStream = (Status:0; ErrorInfo: 1);
```

If an object type contains virtual methods, its constants can be initialised without a constructor call.

Address constants

An address constant is initialised by a constant-address expression.

Example

```
type
  TFontName = string[32];
  PFontName = ^TFontName;
```

const

```
Times: TFontName = 'Tms Rmn';
PTimes: PFontName = @Times;    // Or Addr(Times)
```

In the `{ $\$X+$ }` state, when extended syntax is enabled, a `PChar` typed constant can be initialised with a string constant. In this case the pointer will contain the address of a null-terminated string that is a copy of the string constant.

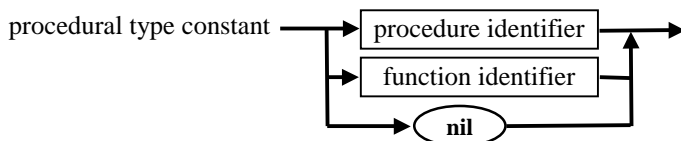
Example

const

```
CityStr: PChar = 'London';
```

Procedural type constants

A procedural type constant is initialised by an identifier of a procedure or function that is assignment-compatible with the constant type or by the value `nil`.



Example

type

```
TPrintString = procedure(const Str: string);
```

```
procedure ConsoleOutput(const Str: string);
```

begin

```
WriteLn(Str);
```

end;

const

```
PrintString: TPrintString = ConsoleOutput;
```

Memory allocation

Variables declared in a program or library block, or in the interface or implementation parts of a unit, are called *global* variables and are stored in the uninitialised data segment named `BSS32`.

Typed constants are stored in the initialised data segment called `DATA32`.

Variables declared in a procedure, function or method block are called *local* variables. They are allocated on the stack of the procedure or function when it is called and are disposed of after the execution of the procedure or function. The stack segment holding local variables is called `STACK32` and the total size of local variables allocated by a procedure or function when it runs can not exceed the size of the stack segment. The size of the stack is set by the `{ $\$M StackSize$ }` compiler

directive. *StackSize* value may not be less than 8 KB to allow for the minimum requirements of the operating system and defaults to 16KB. In the **{&Optimize+}** state, if the address of a local variable is not used, the compiler can allocate it in a CPU register instead of in a memory location.

The **\$\$** switch compiler directive is used to control stack overflow checking in a program. In the default **\$\$+** state, when stack overflow check is turned on, the compiler generates special code on entry to each procedure or function, checking the amount of stack currently available. Stack overflow errors are usually very hard to track, so switch stack checking off only after the code has been thoroughly tested.

Variables in a variable declaration section started by the reserved word **threadvar** are stored as thread local storage (TLS) variables. Thread local storage is created for each thread running, so each thread has its own instance of every **threadvar** variable declared. By redeclaring critical global variables as **threadvar**, old code can be made re-entrant; this is important in a multithreaded environment. For example, the *InOutRes* variable is declared in the *System* unit of Virtual Pascal is declared as **threadvar**. This change is transparent to user code, meaning that the *IOResult* function can be relied upon even in multi-threaded programs. When accessing a variable declared as **threadvar**, it can not be accessed directly using code written in the built-in assembler, but must instead be accessed by means of a Pascal wrapper routine.

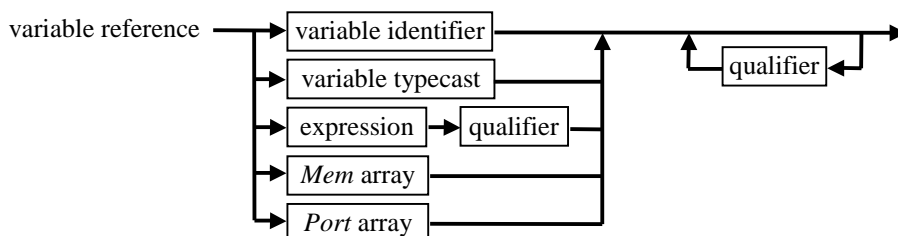
Since Virtual Pascal generates pure 32-bit code, there is no 64KB restriction on the size of any segment. In theory, the total size of all segments can not exceed 4GB, although operating systems may impose limits smaller than this (for example, to maintain compatibility with 16-bit applications, OS/2 restricts the total size of all segments by a 512 MB limit). Some of the available address space is used for common code and data of DLLs, meaning that the actual maximum size is somewhat smaller, depending on the environment.

In the **{&AlignData+}** state, all word-sized (2-byte) global variables and typed constants are aligned at a word boundary. If the size of the global variable is greater than 3 bytes, it is aligned at a double word boundary. In the **{&AlignData-}** state, no alignment is performed. Aligning can significantly increase execution speed, at the cost of memory and hard disk space.

Variable references

In a context where a variable is expected, a variable reference can be substituted. Variable references allow both simple variables, variable type casts, selected components of a structured variable or a dereferenced pointer to be specified.

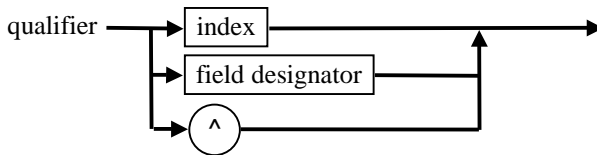
A variable reference has the following syntax:



An expression can be used in a variable reference to calculate the value of a pointer to a dynamic variable. To produce the actual variable reference, the pointer value must be dereferenced (*PChar* pointers can also be indexed in **\$\$X+** state), so the syntax demands the use of a qualifier after the expression.

Qualifiers

Qualifiers can be used in variable references to denote a component of a structured or string-type variable or to dereference or index a pointer value.



Here are some examples:

BitMasks[*I+J*]

The index *I+J* denotes the *I+J*th component of the array-type variable *BitMasks*.

ValRec.*SignificandLo*

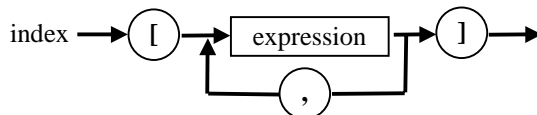
The field designator *SignificandLo* specifies a component of a record-type variable *ValRec*.

B[^]

The caret (^) symbol is used to access the value of a dynamic variable that is pointed to by the pointer-type variable *B*.

Indices

Indices are used to access specific elements of array- or string-type variables. The following diagram shows the syntax for an index:



To access an element of an array-type variable, the variable reference followed by the index (or indices) of the desired element should be specified.

Example

Statement[*M+N*]

Coef[*I*][*J*]

Coef[*I*, *J*]

Each index expression corresponds to one dimension of the array. The number and type of expressions should correspond to the number and type of ordinal types in the array-type declaration.

To access a specific character of a string variable, the variable reference should be followed by a single index expression. The value of the index for a string variable can range from 0 to *N*, where *N* is the declared length of the string, for example:

Sentence[*I*]

specifies the *I*th character of the string *Sentence*. It is of the type *Char*.

For short strings, the character at index 0 contains the current (dynamic) length of the string. Although a string can be indexed beyond its dynamic length, the result of this operation is unpredictable.

For both short and long strings, the *Length* function returns the current length of the string.

In the $\{ \$X+ \}$ state, it is possible to index a value of *PChar* type. It can have only one index expression which should be of *Cardinal* type. The index expression is added to the *PChar* type pointer before it is dereferenced to produce a *Char* type variable reference.

Record field and object component designators

To access a field of a record or a component of an object, class or class reference variable, the variable reference should be followed by a field designator specifying the field or component to access. A component designator that designates a method is called a *method* designator.

The syntax for a field designator is the following:



Example

ValRec.SignificandHi
PersonRec.Name
PPoint^.Xcoor

Within a **with** statement, fields of a record and components of an object, class or class reference variable may be referred to by a designator only: the compiler supplies the variable reference to the base variable. When accessing several fields or components, using a **with** statement can also reduce code size and slightly speed up execution.

Example

```

with Ppoint^ do
begin
  Xcoor := 3;
  Ycoor := 8;
end;
  
```

Pointers and dynamic variables

A pointer variable is used to hold an address of a dynamic variable. When no dynamic variable is associated with a pointer, a special value denoted by the reserved word **nil** is used. It is encoded as the address with value zero and does not point to any variable.

The symbol (^), when it is written after a pointer variable, means that the dynamic variable pointed to by the pointer variable, is dereferenced.

Example

B^
DSchedule.WInfo^

To create a dynamic variable, the standard procedures *New* or *GetMem* can be used.

Variable typecasts

Variable references of one type can be converted to variable references of another type using the typecasting mechanism. The diagram below shows the variable typecast definition:



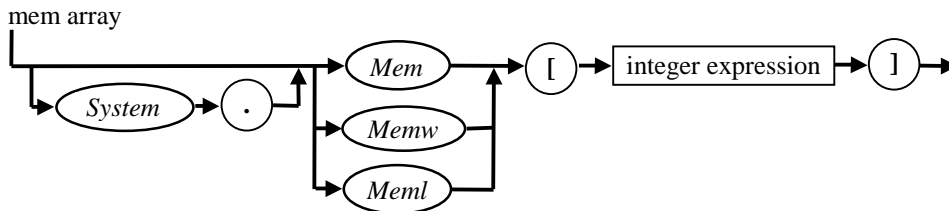
The type identifier denotes the type which the variable reference will have after the typecasting conversion. The size of the type, denoted by the type identifier and the size of the variable, referred to by the variable reference, must be the same.

The syntax allows the use of qualifiers after variable typecasts. For example:

```
var
  F: Text;
begin
  ...
  WriteLn('File handle = ', TextRec(F).Handle);
  ...
end.
```

MEM arrays

Mem arrays are used to directly access memory. Here is the syntax diagram for the *mem* array:



Virtual Pascal has three built-in memory arrays: *Mem*, *MemW* and *MemL*. The components of *Mem* have type *Byte*, the components of *MemW* have type *SmallWord* and the components of a *MemL* have type *Longint*. An *i*-th component of the memory array specifies a *Byte*, *SmallWord* or *Longint* variable located at the absolute address *i*. Unlike Borland Pascal, the segment part of an address is neither required, nor allowed in Virtual Pascal and the integer expression defines a flat memory address.

In OS/2, some 16-bit API calls like *VioGetBuf* return 16:16 type pointer instead of the 32-bit 0:32 pointers used by VP. Before accessing the memory pointed to by such a pointer, it should be converted to a flat memory pointer by calling the *SelToFlat* procedure located in the *System* unit.

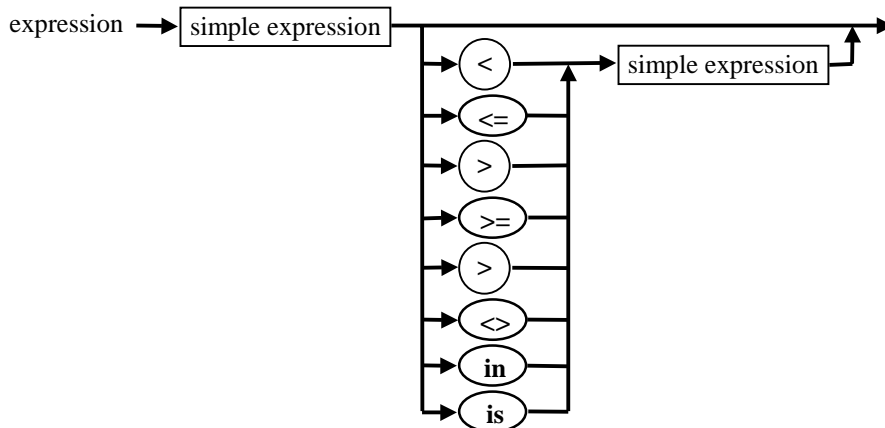
CHAPTER 7

Expressions

An *expression* is a combination of *operators* and *operands* specifying a computation. Operators are either *binary* or *unary*. *Binary* operators take two operands and are entered in infix notation, which means that they are written between operands. *Unary* operators take one operand which must follow the operator.

Expression syntax

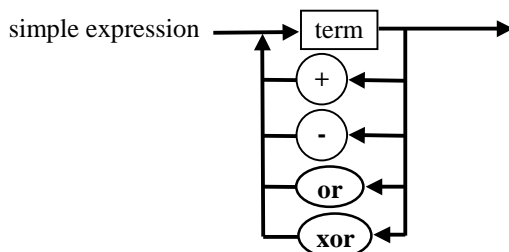
The syntax of expressions is described by the following diagrams:



An expression consists of *simple expressions* and relational operators. These are examples of expressions:

$A < B$
 $(X+I) <> (Y+Z)$
 Red **in** ColourSet
 MyClass **is** TComponent

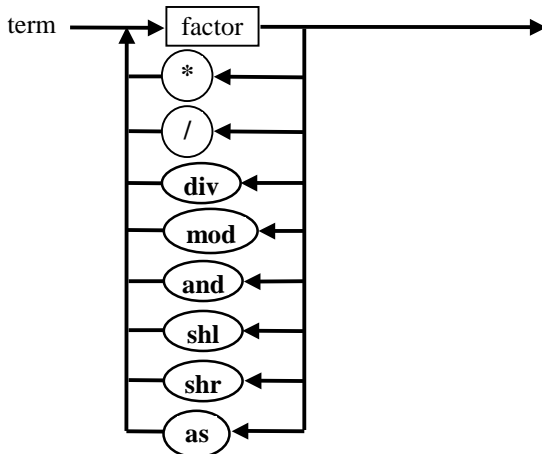
The syntax for a simple expression is the following:



A simple expression is a combination of *terms*, adding operators and bitwise **and** and **or** operators. For example, the simple expression

$X+Y*Z$

is the sum of two terms and the second term has two *factors*. Terms are made up of factors and multiplication operators:



These are examples of terms:

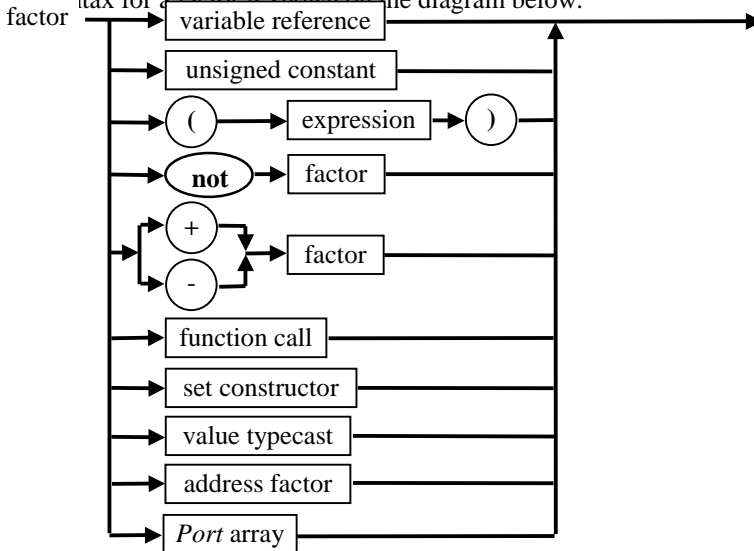
$A * B$

$A \text{ div } (B + C)$

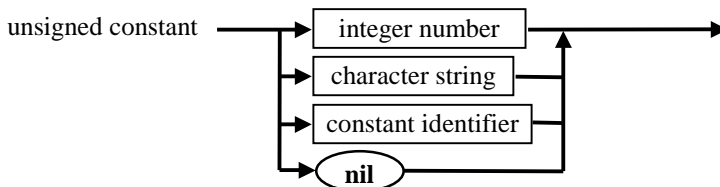
$Z \text{ shl } 1$

$\text{MyObject as TComponent}$

The syntax for a factor is shown on the diagram below:



An unsigned constant has the following syntax:



A function call is used in expressions to invoke a function and receive the result returned by the function. Set constructors are used in expressions to denote set type values. Value typecasts change

the type of a value. An address factor calculates the address of a variable, procedure, function or method.

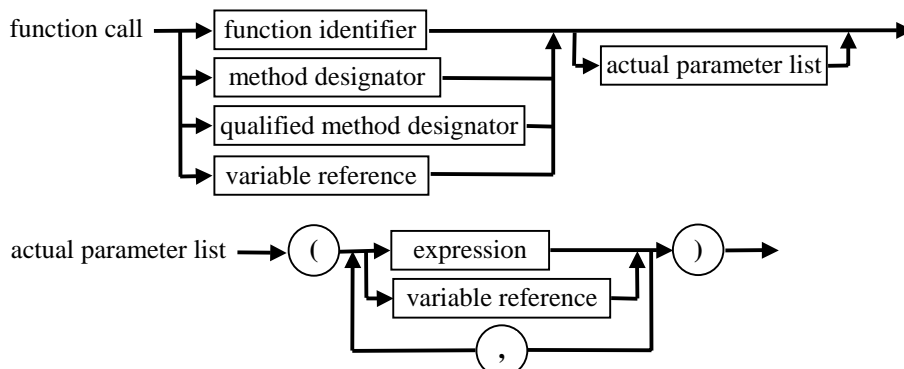
Here are some examples of factors:

A
 2
 (*B+C*)
Exp(-X)
 @*A*

where *A* is variable reference, 2 is an unsigned constant, (*B+C*) is a sub-expression, *Exp(-X)* is a function call and @*A* is a pointer containing the address of the variable *A*.

Function calls

The syntax for a function call is shown in the diagram below:



A function call invokes a function as specified by a function identifier, a method designator, a qualified method designator, or a variable reference of a procedural type. For each formal parameter in the function definition there must be an actual parameter of a corresponding type in the function call. Refer to page 120 for a description of parameter passing rules.

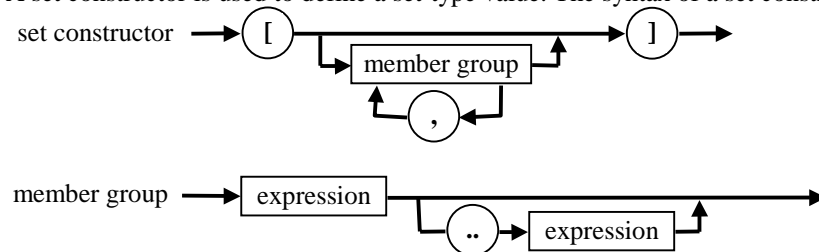
If extended syntax is enabled ($\{\$X+\}$ state), a function can be used in place of a procedure in a procedure call statement. When used in this way, the value returned by the function call is ignored.

Some examples of function calls follow:

*Exp(2*X+Y)*
Succ(A)
UserFunc(Z, 10)

Set constructors

A set constructor is used to define a set-type value. The syntax of a set constructor is:



Each expression denotes a value of the set. All values in a particular set constructor must be of the same ordinal type. A set can have no members at all, in which case it is called *empty* and has the notation `[]`. An empty set is compatible with all other set types.

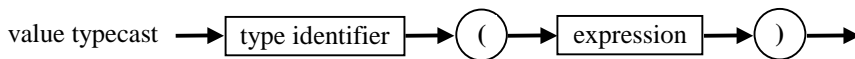
Example

```
['A'..'Z', 'a'..'z', '_']
[1..10, 50..100, A div 2]
```

Value typecasts

A value typecast is used to change the type of an expression. It is applicable to ordinal and pointer types only.

The following diagram shows the syntax of a value typecast:



Unlike variable typecasts, value typecasts convert values (as opposed to variables) and therefore cannot be used in variable references.

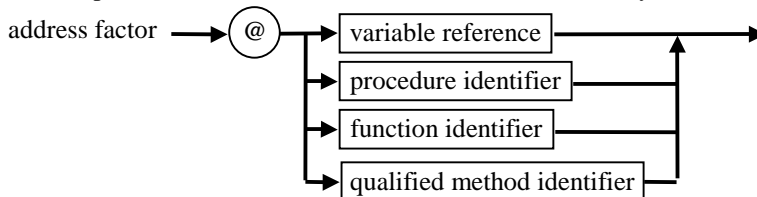
The size of the specified type can be different from that of the expression. If the size of the type is less than the type of the expression, the expression will be truncated; if it is longer, the expression will be extended, with the sign of the expression preserved.

Some examples of value typecast follow:

```
Char(137)
Metal(J)
Longint(@J)
```

The @ operator

The @ operator is used in an address factor, which has the syntax:



The @ operator is used to calculate the address of a variable, procedure, function or method. When the @ operator is applied to a method, the method should be designated by a qualified method identifier.

When applied to a variable reference, the @ operator returns a pointer to the variable. The **\$T** switch compiler directive controls the types of pointer values generated by the @ operator. In the **{\$T-}** state, the result type of the @ operator is always an untyped pointer that is compatible with all pointer types. In the **{\$T+}** state, when the @ operator is applied to a variable of type *T*, the type of the result is $\wedge T$, which is compatible only with other pointers to this type of variable.

When applied to a procedure, function or method identifier, the @ operator prevents the compiler from invoking the routine and returns a pointer to the entry point of the function, procedure or method. The result type of the @ operator is always an untyped pointer that is compatible with all pointer types.

When applied to a procedural variable, the address operator converts the argument into an untyped pointer variable that contains an address. The @ operator can also be used to assign an untyped pointer value to a procedural variable. For example, consider the following fragment from the *WinCrt* unit,

```

type
  FnWp = function(Wnd: HWnd; Msg:Wong; Mp1, Mp2: MParam): MResult;
  ...
var
  OldFrameWndProc: FnWp;
  ...
begin
  ...
  @OldFrameWndProc:=WinSubclassWindow(CrtWindow, Frame, FrameWndProc);
  ...
end

```

The *WinSubclassWindow* function returns as a result an untyped pointer value, which can be assigned to a procedural variable as shown above. After execution of the assignment statement, *OldFrameWndProc* does *not* contain the address of the *WinSubclassWindow* function - but the address of the procedure returned by it.

A double address operator (@@) applied to a procedural value returns the memory address of the procedural variable (not the address stored in it).

Operators


Operators are tokens that trigger some computation when applied to operands in an expression. There are arithmetic operators, logical operators, string operators, character-pointer operators, set operators, relational operators and the @ operator.

Rules of precedence

Expressions are evaluated according to certain rules of precedence, that depend on the operators used and the presence of parentheses.

The following table shows the precedence of the operators:

Operators	Precedence
@, not	High
*, /, div , mod , and , shl , shr , as	
+, -, or , xor	
=, <>, <, >, <=, >=, in , is	Low



Operators with higher precedence are performed before operators with lower precedence. Operators of the same precedence are performed from left to right. The use of parentheses changes the order of evaluation: expressions enclosed in parentheses are evaluated first.

Example

$X + Y < 0$	<i>first evaluates $X+Y$, then compares with 0</i>
$X \text{ shr } 2 + Y \text{ xor } 5$	<i>evaluates the left side of the + operator, adds Y and finally applies the xor operator to the result</i>
$X \text{ div } (Y+3)$	<i>first evaluates $Y+3$, then divides X by the result</i>
not A and C	<i>is True, if A is False and C is True</i>
not (A and C)	<i>is True, if A is False and C is False</i>

Arithmetic operators

The following table describes the arithmetic operators, the possible types of operands and the resulting type of the operations.

Operator	Operator type	Operation	Operand types	Result type
+	Binary	addition	integer type	<i>Longint</i>
			real type	<i>Extended</i>
+	Unary	sign identity	integer type	<i>Longint</i>
			real type	<i>Extended</i>
-	Binary	subtraction	Integer type	<i>Longint</i>
			Real type	<i>Extended</i>
-	Unary	sign negation	Integer type	<i>Longint</i>
			Real type	<i>Extended</i>
*	Binary	multiplication	Integer type	<i>Longint</i>
			Real type	<i>Extended</i>
/	Binary	division	Integer type	<i>Extended</i>
			Real type	<i>Extended</i>
div	Binary	integer division	Integer type	<i>Longint</i>
mod	Binary	remainder	Integer type	<i>Longint</i>

The + operator can also be used as a string or set operator and the +, -, * operators can also be used as set operators. If *A* and *B* are sets of the same type, *A* * *B* denotes a set containing only those elements present in both *A* and *B*.

An operand type can be a subrange of an ordinal type. In this case all operations with the operand are performed as with the ordinal type operand.

If both operands are of an integer type, they are converted to *Longint* before the operation and the result type of the operation is always *Longint*.

If one or both operands of a / operator are of an integer type, they are converted to *Extended* before the operation and the result of the operation is always *Extended*. The second operand must be non-zero, otherwise a run-time error occurs or an exception is raised.

op1 div op2 is the quotient of *op1/op2* rounded towards zero to a value of an integer type. *op2* must be a non-zero value. The use of **div** with a second operand of zero results in an error.

The **mod** operator is used to obtain the remainder when dividing one operand by the other.

$$op1 \text{ mod } op2 = op1 - (op1 \text{ div } op2) * op2$$

The result of **mod** has the same sign as *op1*; *op2* must be non-zero. The use of **mod** with a second operand of zero results in an error.

Bitwise logical operators

The following table describes the bitwise logical operators, the possible type of the operands and the result type of the operation.

Operator	Operator type	Operation	Operand types	Result type
not	Unary	bitwise negation	integer type	integer type
and	Binary	bitwise and	integer type	integer type
or	Binary	bitwise or	integer type	integer type
xor	Binary	bitwise xor	integer type	integer type
shl	Binary	shift left	integer type	<i>Longint</i>
shr	Binary	shift right	integer type	<i>Longint</i>

Each bit in the result value of the bitwise logical operations is determined as shown in the table below:

Bit in <i>op1</i>	Bit in <i>op2</i>	not <i>op1</i>	<i>op1</i> and <i>op2</i>	<i>op1</i> or <i>op2</i>	<i>op1</i> xor <i>op2</i>
0	0	1	0	0	0
1	0	0	0	1	1
0	1	1	0	1	1
1	1	0	1	1	0

The result of the **not** operator is of the same integer type as the operand.

The result type of **and**, **or** and **xor** operators is the common type of both operands. The common type of the two operands is the basic integer type with the smallest range that includes all values of the integer types of both operands.

The result of *op1* **shl** *op2* is the value of *op1* left-shifted by *op2* bits. The result of *op1* **shr** *op2* is the value of *op1* right-shifted by *op2* bits. The type of the result of both operators is the type *Longint*.

Examples

```
10 shr 1 = 5
10 shl 2 = 40
3 xor 26 = $03 xor $1A = $19 = 25
10 and 6 = 2
10 or 6 = 14
```

Boolean logical operators

The table below describes the boolean logical operators, the possible types of the operands and the result type of the operations.

Operator	Operation	Operand types	Result type
not	negation	Boolean type	<i>Boolean</i>
and	logical and	Boolean type	<i>Boolean</i>
or	logical or	Boolean type	<i>Boolean</i>
xor	logical xor	Boolean type	<i>Boolean</i>

<i>op1</i>	<i>op2</i>	not <i>op1</i>	<i>op1</i> and <i>op2</i>	<i>op1</i> or <i>op2</i>	<i>op1</i> xor <i>op2</i>
<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>

Virtual Pascal supports two models of code generation for the **and** and **or** operators: short-circuit evaluation in the **{\$B-}** state and complete evaluation in the **{\$B+}** state. It is easy to see from the table above that the result of the **and** operation is *False* if at least one of the operands is *False*. The result of the **or** operation is *True* if at least one of the operands is *True*. The short-circuit evaluation model takes these facts into consideration and evaluates the operands from left to right. The

evaluation stops when the result of the entire expression becomes evident. Short-circuit evaluation is convenient to use and results in minimum execution time. Time consuming operands (e.g. including function calls) should be used in the end of an expression since this reduces the chance for them to be evaluated. Short-circuit evaluation makes it possible to evaluate constructs that could not otherwise be used. For example:

```
while (P <> nil) and (P^.Count <> 10) do
  P: = P^.Next;
```

In the **{SB-}** state the second test is not evaluated if the first test is *False*. In the **{SB+}** state, the second test would cause a run-time error if P was **nil**.

Complete evaluation evaluates all operands of an expression built from **and** and **or** operators without stopping when the result of the entire expression is known. Complete evaluation can be used if a boolean expression involves function calls that can affect the meaning of the program.

String operator

The table below describes the string operator, the possible types of operands and the result type of the operation.

Operator	Operator type	Operation	Operand types	Result type
+	Binary	Concatenation	string type, <i>Char</i> type, or packed string type	string type

The + operator can be used to concatenate two string operands. The result of $op1+op2$, where $op1$ and $op2$ are of a string type, a *Char* type, or a packed string type, is the combined string of $op1$ and $op2$.

In the **{SH+}** state, the result of a string concatenation is always a long string. In the **{SH-}** state, the resulting string is a short string, truncated to 255 characters. The result value is compatible with all string types, but not compatible with *Char* and packed string types.

PChar operators

In the **{SX+}** state (when extended syntax is enabled), a number of *PChar* operations are allowed. In the following table, where P and Q are values of type *PChar* and I is a value of type *Longint*, the permitted operation with *PChar* values are shown:

Operation	Result
$P + I$	Add I to the flat address of P
$I + P$	Add I to the flat address of P
$P - I$	Subtract I from the flat address of P
$P - Q$	Subtract the flat address of Q from the flat address of P

The plus (+) and minus (-) operators can be used to increment and decrement the flat address of a pointer value. The result of the operations $P+I$ and $I+P$ is a pointer value that points I characters after P . The result of the operation $P-I$ is a pointer that points I characters before P .

The (-) operator can be used to calculate the difference between the flat addresses of two *PChar* values. The result of the operation $P-Q$ (where P is the higher address and Q is the lower address) is a value of type *Longint* that is the number of characters between Q and P . P and Q should point within the same character array, or the result value is undefined.

Example

```

uses
  Strings;
var
  P, Q: PChar;
begin
  P := 'This is a text string';
  Q := StrPos(P, 'text');
  if Assigned(Q) then
    Writeln('Q points ', Q-P, ' characters into P')
  else
    Writeln('Text not found; Q is nil');
end;

```

Set operators

The table below describes the set operators, the possible types of the operands and the result type of the operations.

Operator	Operator type	Operation	Operand types
+	Binary	union	Compatible set types
-	Binary	difference	Compatible set types
*	Binary	intersection	Compatible set types

The result value of the operation $op1 + op2$ is a set containing the members of both sets $op1$ and $op2$.

The result value of the operation $op1 - op2$ is a set, containing all the members of the set -type value $op1$, which are not members of the set-type value $op2$.

The result value of the operation $op1 * op2$ is a set, containing only the values which are members of both sets $op1$ and $op2$.

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 a **set of $A..B$** .

Example

```

var
  A: set of 0..10;
  B: set of 5..100;
begin
  A := [3, 5, 7..9];
  B := [6, 8, 31];
  // Ask three questions, the answer to all three is TRUE
  Writeln('A and B both contain 8? ', 8 in A*B);
  Writeln('Either A or B contain 3? ', 3 in A+B);
  Writeln('Only A contains 7? ', 7 in A-B);
end.

```

Relational operators

The table below shows the possible types of operands and the result types for the relational operations:

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

- Comparing simple types

The relational operators =, <>, <, >, >=, <= can be applied to operands of a simple type. In order for two simple operands to be compared, they must be of compatible types, although one operand can be of a real type and the other can be of an integer type.

- Comparing strings

The relational operators =, <>, <, >, >=, <= can be applied to operands of a string type. Relations between two strings are established by relation between characters on corresponding positions. If all characters of the first string are equal to the corresponding characters of the second string, but the first string is shorter, then it is considered to be less. A string-type value can be compared with any other string-type value (because all string-type values are compatible), with a character-type value, or with a packed string-type value. A character-type value is considered to be a string of length 1 when it is compared with a string-type value. A packed string-type value with *N* components is considered to be a string with length *N* when it is compared with a string-type value.

- Comparing packed strings

The relational operators =, <>, <, >, >=, <= can be applied to the two operands of a packed string type if both of them have the same number of components. In this case they can be compared just like two values of a string type with the same length.

- Comparing pointers

The relational operators =, <> can be applied to pointer-type operands. Two pointer values are equal only if they point to the same object. To be compared the two operands must be of compatible pointer types.

- Comparing *PChar* pointers

The relational operators >, <, >=, <= can be applied to *PChar* values in the **{SX+}** state (when extended syntax is enabled). The operators compare the flat addresses of the pointer values, so the two *PChar* values must point to the same character array, otherwise the result of the relational operation will be undefined.

- Comparing sets

The relational operators =, <>, <=, >= may be used to compare sets:

The = operator denotes set equality and the <> operator denotes set inequality. A set-type value X is equal to a set-type value Y (X = Y) only if X and Y contain exactly the same members; otherwise X <> Y.

The <= operator denotes 'is contained in'. A set-type value X is contained in a set type value Y (X <= Y) if every member of X is also a member of Y.

The >= operator denotes 'contains'. A set-type value X contains a set type value Y (X >= Y) if every member of Y is also a member of X.

- Testing set membership

The **in** operator is used to test set membership. If a value of the ordinal-type operand *op1* is a member of the set-type operand *op2*, then the result of the operator *op1 in op2* is *True*, otherwise, it is *False*.

Class operators

There are two class operators: **is** and **as**. Both operators have the same syntax:

ObjectRef operator *ClassRef*

where operator is **as** or **is**, *ObjectRef* is an object reference and *ClassRef* is a class reference. *ObjectRef* must be an instance of a class that is an ancestor of, equal to, or a descendant of *ClassRef*. These operators use run time type information (RTTI) generated for the class type.

The is operator

With the **is** operator it is possible to determine whether an object is of a given type or one of its descendants. The **is** operator returns *True* if *ObjectRef* is an instance of the class or an instance of a class derived from the class denoted by *ClassRef*. Otherwise, the **is** operator returns *False*. If *ObjectRef* is **nil**, the result is also *False*.

The **is** operator can also be used to perform a *protected typecast*:

```
if (Sibling <> Self) and (Sibling is TRadioButton) then
    TRadioButton(Sibling).SetChecked(False);
```

If the value of the test expression is *True*, *Sibling* can be safely typecast to be of class *TRadioButton*.

According to the rules of precedence, the **is** operator has the same precedence as the relational operators and the **in** operator. So (as it is shown in the example above) an expression with an **is** operator must be enclosed in parentheses to be treated as a single operand in a boolean expression containing **and** or **or** operands.

The as operator

The **as** operator is used to assure safe typecasting of objects. The result of the expression is a reference to the same object as *ObjectRef* with the type denoted by *ClassRef*. It is equivalent to the typecast construct:

```
if ObjectRef is ClassRef then ClassRef(ObjectRef)...
```

but unlike it, using the **as** operator raises an *EInvalidCast* exception if *ObjectRef* is not an instance of the class denoted by *ClassRef*, an instance of a class derived from the class denoted by *ClassRef*, or **nil**.

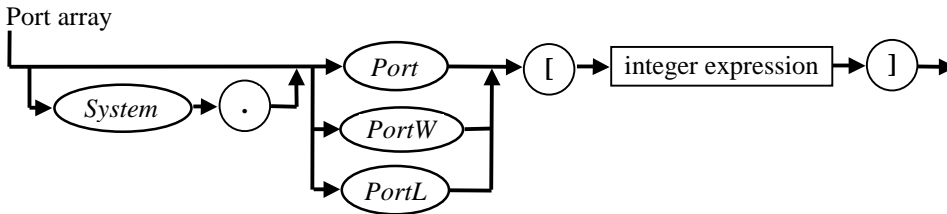
The **as** typecast is most useful as a shorthand when creating a **with..do** block that includes a typecast:

with *GetComponent(0)* **as** *TReport* **do** ...

According to the rules of precedence the **as** operator has the same precedence as the multiplying operators. So if an **as** operator is used in a variable reference and the field designator follows the variable reference, an **as** type cast must be enclosed in parentheses.

Port arrays

Port arrays are used to access CPU data ports directly. Here is the syntax diagram for the *Port* arrays:



Virtual Pascal has three built-in port arrays: *Port*, *PortW* and *PortL*. The arrays are one-dimensional and an *i*-th component of a port array specifies a data port with the address corresponding to the index *i*. The components of *Port* have type *Byte*, the components of *PortW* have type *SmallWord* and the components of a *PortL* have type *Longint*. The index type is the 16-bit type *SmallWord*.

If a component of *Port*, *PortW*, *PortL* is referenced in an expression, its value is input from the selected port. If a component of *Port*, *PortW*, *PortL* is assigned a value, the value is output from the selected port. Components of *Port*, *PortW*, *PortL* can not be used as variable parameters. It is also not allowed to reference the entire *Port*, *PortW* or *PortL* array.

The actual reading from port and writing to port is performed by special run-time library routines which are only linked into the program if a *Port* array is used. Special privileges and other system-dependent setup is handled seamlessly by the compiler.

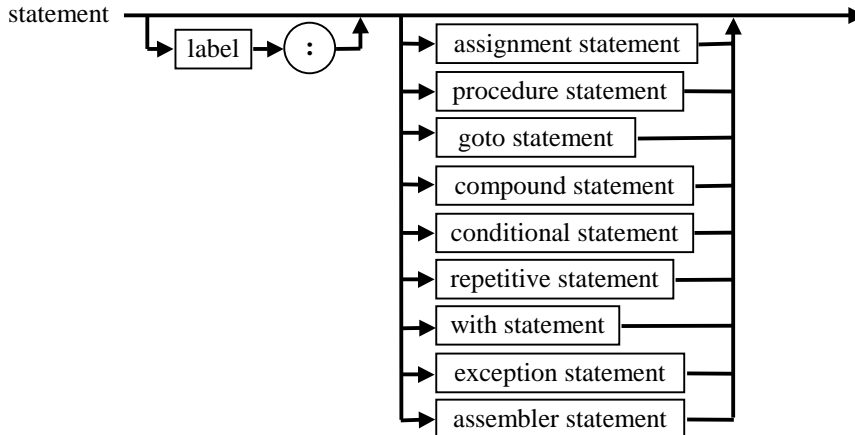
In OS/2, a program that uses direct port access can only run if a system wide option allows direct port access. To make sure this is the case, the following line must be specified in the CONFIG.SYS file of OS/2:

IOPL=YES

CHAPTER 8

Statements

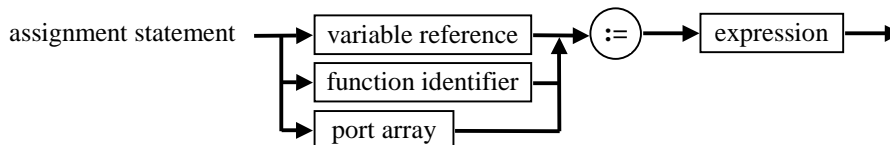
Statements specify the flow of control as a program executes. If there are no explicit jumps or conditional statements, statements are executed sequentially in the order of appearance in the source code.



A statement can be prefixed by a label. The label can be a digit sequence in the range 0..9999 or an identifier. A label can be referenced by a **goto** statement within the routine in which it is declared.

Assignment statements

Assignment statements are used to set the value of a variable or the return value of a function. An assignment statement has the following syntax:



An expression specifies the value that will be given to the variable or the function result. The type of the expression must be assignment-compatible with the type of the variable or the type of the function result.

In the **{&Delphi+}** state, functions also implicitly defines a local variable named *Result*, of the same type as the function return value. Assigning a value to *Result* has the same effect as assigning a value to the function identifier. In addition, *Result* can be referred to in expressions referring to the current function result value, without causing a recursive call.

Examples

```
Number := 1;
Square := A*B;
Z := Exp(2*X+Y);
ColourSet := [Yellow, Succ(Blue)];
```

```
function Sum(_X: Array of Longint): Longint;
```

```

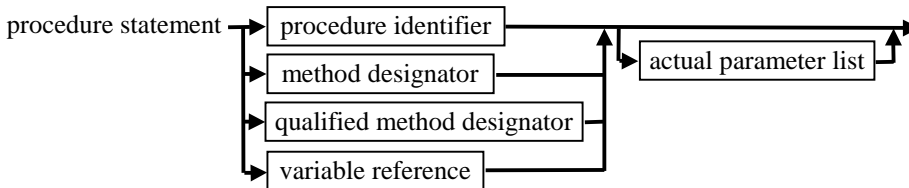
var
  I: Integer;
begin
  Result := 0;
  for i := Low(X) to High(X) do
    inc(Result, _X[i]);
end;

```

For old-style **object** types, an instance of an object type can be assigned an instance of any of its descendant types. This ensures that all fields of the parent object will receive valid values from its descendant.

Procedure statements

A procedure statement invokes a procedure, specified by a procedure identifier, a method designator, a qualified method designator or a variable reference of a procedural type. The syntax for a procedural statement follows:



The actual parameters are written in a list following the procedure identifier. For each formal parameter in the procedure definition, there must be an actual parameter of a corresponding type in the procedure statement. Refer to page 120 for a description of parameter passing rules.

Examples:

```

Writeln('Examples:');
DrawTable(M, N);
SelectTable;
MyObject.Test(17);

```

Goto statements

A **goto** statement directly transfers control to a labelled statement, which must be within the same statement part of the procedure, function or a module initialisation. The following diagram shows the syntax of a **goto** statement:

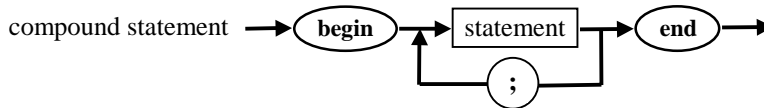


There are some important restrictions on the use of the **goto** statement. A **goto** statement must only be used to jump within a block. If the current block is not an exception block, it is possible to jump from an inner level to an outer level. A compiler error is reported if a **goto** attempts to jump into or out of an exception block. In all other cases an error is not reported, although a **goto** to a statement located in a deeper level of nesting can result in unpredictable effects.

The use of the **goto** statements makes a program less readable and may introduce bugs that are hard to track and it is recommended to avoid the use of **goto** statements wherever possible. As a good alternative, the safe *Break*, *Continue* and *Exit* standard procedures can be used to replace **goto** statements.

Compound statements

A compound statement is a sequence of statements introduced by **begin** and terminated by **end**. There must be a semicolon between each pair of statements. The statements are to be executed in the same sequence as they are written. Wherever a statement may be written in a program, a compound statement may be used instead.

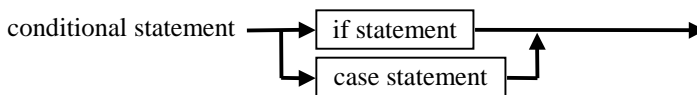


This is an example of a compound statement:

```
begin
  Z: = Exp(2*X+Y);
  Writeln('Result:', Z);
end;
```

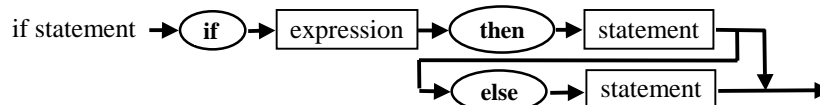
Conditional statements

Conditional statements select from alternative statements that could be executed. The selection is based on the evaluation of a conditional expression.



if statements

An **if** statement enables the process to select one of two actions. The selection is made by evaluating a conditional expression of type *Boolean*. The following diagram shows the syntax for an **if** statement:



If the value of the expression is *True*, the statement following **then** is executed. If the value of the expression is *False*, the statement following **else** is executed. The **else** clause may be omitted, in which case no action will be taken if the condition yields *False* when it is evaluated. There are no semicolons in an **if** statement and it is syntactically incorrect to put a semicolon before **then** or **else**. Each statement may be a compound statement. The statements after **then** and **else** may themselves be **if** statements:

```
if expression1 then if expression2 then statement1 else statement2 else statement3;
```

In this case the **else** part belongs to the nearest **if** for which there is no **else** clause and is equivalent to the following:

```
if expression1 then
  begin
    if expression2 then
      statement1
    else
      statement2
  end
else
  statement3;
```

Example:

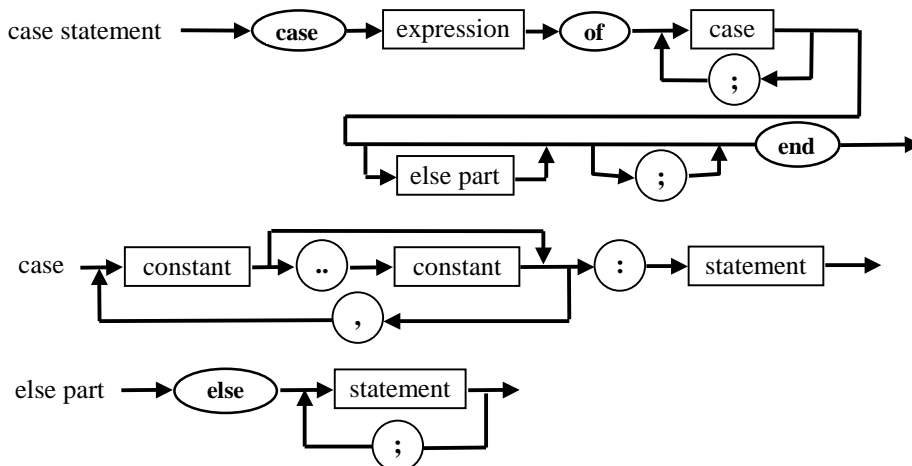
```

if Number <= Max then
    Inc(Number)
else
    Number := Max;

```

Case statements

A **case** statement enables one of several statements to be executed depending on the value of a selector expression. The selector expression can be of any ordinal type. The syntax for a **case** statement follows:



Any statement in the **case** statement is prefixed by one or more *case constants*, *case ranges* or with the reserved word **else**. All *case constants* must be the unique constants of an ordinal type compatible with the type of the selector. *Case constants* and *case ranges* may not overlap each other.

After evaluating the selector, control is transferred to the statement with an equal *case constant* or with a *case range* which includes the selector value. If no match is found and an **else** part is present, the statements following **else** are executed. If no match is found and there is no **else** part, none of the statements in the **case** statement are executed and control is passed to the statement following the **case** statement.

Examples:

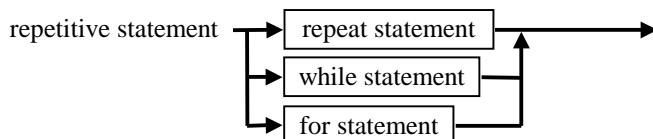
```

case Ch of
    'A'..'Z', 'a'..'z': WriteLn ('The letter ', Ch ' is found');
    '0'..'9': WriteLn ('The digit ', Ch ' is found');
else
    WriteLn ('No letters or digits are found');
end

```

Repetitive statements

Repetitive statements allow a set of statements to be executed several times. There are three forms of repetitive statements:

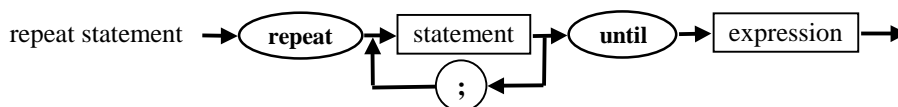


The **repeat** statement is a loop with a termination condition that is evaluated at the end of the loop body. In the **while** statement, the condition is evaluated at the beginning of the loop. The **for** statement is used when the number of repetitions is known when the program is written.

The *Break* and *Continue* standard procedures can be used only inside repetitive statements. *Break* terminates the repetition, *Continue* transfers control to the test expression for **while** and **repeat** statements and to the next iteration in **for** loops.

Repeat statement

A repeat statement has the following syntax:



The expression must be of type *Boolean*. The reserved words **repeat** and **until** act as statement brackets and several statements separated by semicolons may appear between them. The statements within the loop execute repeatedly. Usually one of them affects the value of the terminating expression and when it is evaluated to *True*, the loop terminates. The expression is evaluated after each execution of the loop, so at least one execution of the **repeat** statement body is assured.

Examples:

```

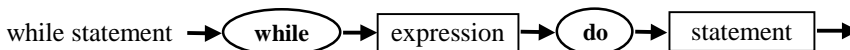
repeat
  if A[I] < Min then
    Min := A[I];
  I := I + 1;
until I > 31;
  
```

```

repeat
  DoForever;
until False;
  
```

While statement

A **while** statement has the following syntax:



The expression must be of type *Boolean* and is evaluated at the beginning of the loop. The expression must have a well-defined value on entry to the statement. The body of the loop consists of a single statement, which can be a compound statement. The statement is executed repeatedly while the expression is *True* and the loop terminates when the expression becomes *False*. If it is *False* at the beginning, the statement is not executed at all.

Example:

```

P := CmdLine;
Q := Param;
while P^ in [#1..' ' ]do
  
```

```

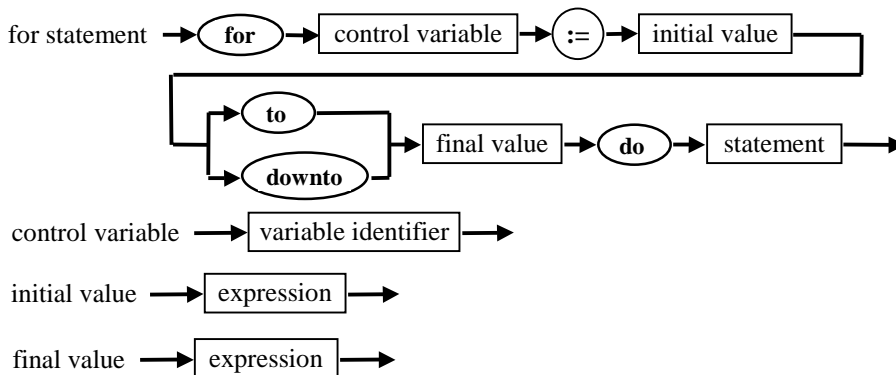
    Inc(P);
while not ( P^ in [#0.. ' ']) do
    begin
        Q^ := P^;
        Inc(P);
        Inc(Q);
    end;
    Q^ := #0;

while True do
    DoForever;

```

For statements

If the number of repetitions does not depend on the statements within the loop, the appropriate construction is a **for** loop. It has the following syntax:



The *control variable* is used as a counter of iterations in the **for** statement. The control variable must be a variable identifier of an ordinal type. The control variable's values range from *initial value* to *final value* and always start at *initial value*. The *initial* and *final* value must be of a type, assignment-compatible with the control variable type. The body of the loop consists of one statement that can be a compound statement. It is executed once for each value of the control variable. The value of the control variable is incremented by one in each iteration for the **to** variant of the loop and is decremented by one for the **downto** variant.

The general form of the **for** statement is:

```

for cv := expression1 to expression2 do statement;
for cv := expression1 downto expression2 do statement;
and is defined as to being equivalent to the following compound statement:

```

```

begin
    temp1 := expression1;
    temp2 := expression2;
    if temp1 <= temp2 then    { temp1 >= temp2 for the downto variant }
    begin
        cv := temp1;
        statement;
        while cv <> temp2 do
        begin
            cv := Succ(cv);          { cv := Pred(cv); for the downto variant }
            statement
        end
    end

```

```

end
end
end;
```

where *cv* is the control variable of the **for** statement. The variables *temp1* and *temp2* have the same type as *cv*. They are created by the compiler and can not be accessed by the program. The point of introducing them is to ensure that *expression1* and *expression2* are evaluated once only before the loop is entered and to prevent the limits of the **for** loop from being changed within the loop.

When the **for** statement terminates, the value of the control variable is undefined, unless the execution of the loop was interrupted by a **goto** or *Break* from the **for** loop body. The value of the **for** control variable must not be changed in the body of the loop.

The value of the **for** loop control variable is undefined when the **for** loop terminates and must not be relied upon.

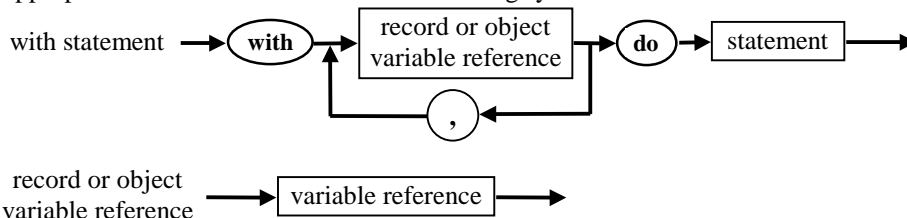
Examples:

```

for I := 1 to 10 do
  for J := 1 to 10 do
    if I=J then
      E[I,J] := 1
    else
      E[I,J] := 0;
```

With statements

If it is necessary to access the same component of a record or object, or different components of the same record or object, several times in a small section of the program, a **with** statement is the appropriate construct to use. It has the following syntax:



Within the *statement*, components of the record or object type may be referred to by field name only: the compiler supplies the variable reference to the record or object. In addition to saving some writing, the **with** statement may result in more efficient code because the record or object needs only be located once, instead of several times. This is particularly useful if a reference to a record or object variable involves function calls, indexing or dereferencing a pointer, as they will be evaluated once, before the statement is executed.

Examples:

```

type
  PTreeNode = ^TTreeNode;
  TTreeNode = record
    Left: PTreeNode;
    Right: PTreeNode;
    Info: Integer;
  end;
var
  P: PTreeNode;
```

```
begin
  ...
  New(P);
  with  $P^{\wedge}$  do
  begin
    Left := nil;
    Right := nil;
    Info := 1;
  end;
  ...
end;
```

A **with** statement introduces a new record or object scope which overrides all enclosing scopes. This means that if a variable reference can be interpreted as a field of the record within a **with** statement, it will be interpreted so, even if a variable with the same name could also be accessed.

The form of the **with** statement:

```
with  $V_1, V_2, \dots V_N$  do  $S$ ;
```

is defined as equivalent to

```
with  $V_1$  do
  with  $V_2$  do
  ...
  with  $V_N$  do
     $S$ ;
```

Exception statements

An exception is an abnormal situation that arises at run time and interrupts the execution of a program. In such a case, control is passed to the *exception handler*, which specifies the actions to be taken to deal with the situation. The *exception - handling mechanism* supported by Virtual Pascal provides a structured means of processing error conditions.

The exception - handling mechanism is only available in the **{&Delphi+}** state and is implemented by the *SysUtils* unit, which must be included in programs relying on exception statements.

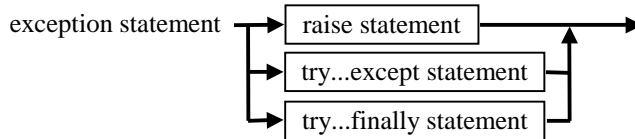
Exceptions are represented by class types descending from *Exception*. This allows exceptions to be grouped in a hierarchy, so new exceptions can be easily implemented without changing the existing code. An exception class instance carries information from the point where the exception is raised to the point where the exception is caught. Information about what error occurred and where it happened is stored in the exception and can be useful when the program encounters an anomaly at run time.

Exceptions are declared in the *SysUtils* unit. If an application that uses the *SysUtils* unit encounters an abnormal situation during execution, control is transferred to the appropriate exception handler. The exception handler usually lets the application recover from the error and continue running. Error conditions that would otherwise have terminated the program, (like writing to full disks or attempting to open non-existing files) can be caught and handled.

The declaration of an exception is the same as the declaration of an ordinary class. Although Virtual Pascal allows an exception to be an instance of any class, it is advisable for an exception object to be derived from the *Exception* class declared in the *SysUtils* unit, since the standard exception handlers handle only exceptions derived from the *Exception* class. This means, that if a new exception is

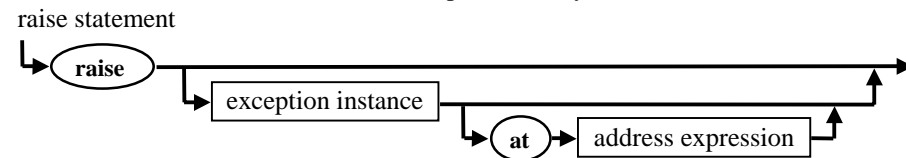
raised in a block of code that is not protected by a specific exception handler for that exception, one of the standard handlers will be used.

Exception statements allows the use of the exception-handling mechanism. There are three forms of exception statements:



The raise statement

A **raise** statement is used to raise an exception. The syntax of the **raise** statement is shown below:



The reserved word **raise** must be followed by an expression of a class instance type. An instance of an exception is usually constructed in the **raise** statement through a call to the *Create* constructor of the corresponding exception class. When an exception is raised, it causes control to be transferred to the innermost exception handler: control *does not* return to the statement following **raise**. After handling the exception, the exception object is automatically destroyed by the default object's *Destroy* destructor. Destroying a raised exception object manually is not allowed. The innermost exception handler is the handler whose **try...except** block was most recently entered and not yet exited.

If the optional **at** part is not specified, an exception is raised on the address of the **raise** statement. To specify a particular address for the exception, use the reserved word **at** followed by an address expression.

Example:

```

if FontSize <= 0 then
  raise EFontError.CreateFmt('Invalid font size: %d', [FontSize]);

```

As can be seen from the example, the default *Exception* class defines a number of constructors that allow meaningful descriptions to be created when exceptions are raised. The *CreateFmt* constructor used here allows the same parameters to be passed as is the case for the *Format* function defined in *SysUtils*. Please refer to the Virtual Pascal run-time library reference for more information.

A **raise** statement can omit the argument, which causes it to *re-raise* the current exception. This form of a **raise** statement can be used in an *exception block* only. Usually an exception handler handles all local exceptions that are known to it. If an exception handler does not know what to do with an exception of a particular type, the best action is to transfer control to an outer exception handler. The *re-raise* form of the **raise** statement transfers control to the exception handler of the enclosing block. Without it, after exiting the handler, the exception would be automatically destroyed and the enclosing block would not be given the option of handling it.

Example:

```

try
  DoCalculation;
except

```

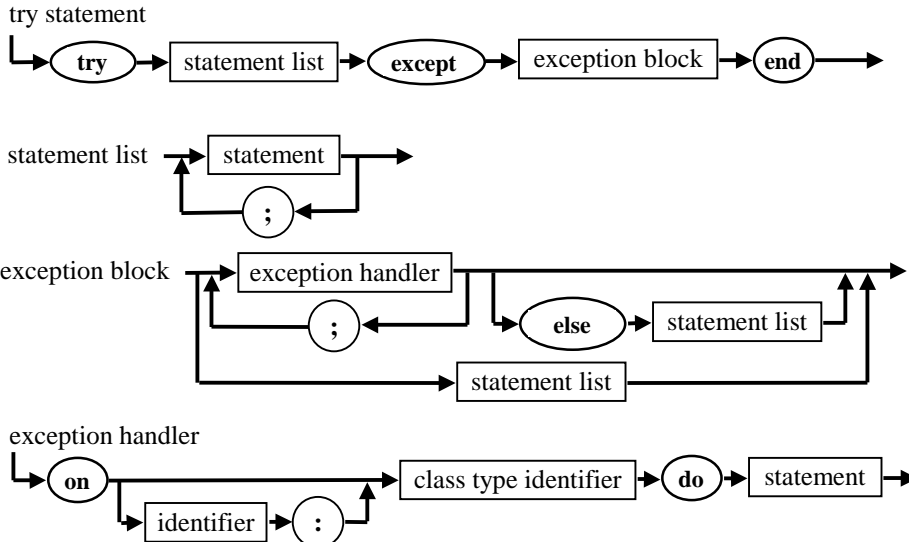
```

    on EZeroDivide do HandleZeroDivide;
else { Exception that is unknown to this handler }
    raise;
end;

```

The try...except statement

Try...except statements are used to handle exceptions.



The *try statement list* contains the statements that are executed normally. If no exception occurs, the **try...except** statement is exited after the last statement of the *statement list*. The **except** part of the statement contains exception handlers that are used to catch exceptions. If an exception is raised, control is transferred to the exception handler which can then handle the exception or re-raise it if it does not know how to handle it. Exceptions can be raised only by a **raise** statement in the *try statement list* or by a procedure or function called in the *try statement list*. The *SysUtils* unit is used to convert run time error codes and CPU exceptions into corresponding language exceptions. All CPU and system exceptions are caught by the system exception handler and then translated to the corresponding exception object which is then raised by a **raise** statement.

Example:

```

try
    DoSomething;
except
    on EInOutError do HandleIOError;
end;

```

An exception handler listed in the **except** part will receive exceptions of the exception class specified after the **on** reserved word or any descendants of that exception class. The first matching handler is executed, which is why exception handlers for more general exception classes should be listed *after* descendant exception handlers to give them a chance to catch their own exceptions. If none of the **on...do** exception handlers can handle the exception and the *exception block* contains an **else** part, the statements in the **else** part are supposed to handle the exception.

Examples:

```

try
...
except

```

```

    on X1 do HandleX1;
    on X2 do HandleX2;
else
    HandleOtherExceptions;
end;

```

If an *exception block* consists only of a statement list, it is supposed to handle all exceptions.

```

try
...
except
    HandleAnyException;
end;

```

When an exception is raised, the innermost exception handler that can handle exceptions of the given class is sought. First the current **try...except** statement is checked. If it can not handle exceptions of the given class, the previously current **try...except** statement is examined. This process continues until an appropriate handler is found and control is passed to it. If a handler is not found and there are no more active **try...except** statements, a run-time error occurs and the application is terminated.

When an appropriate exception handler is found, the stack is unwound to the procedure or function that contains the handler and control is passed to the handler.

After the exception handler has executed, the exception object is automatically destroyed through a call to the object's *Destroy* destructor and control is transferred to the statement following the **try...except** statement. Control *does not* return to the *statement list*.

If an **on...do** exception handler specifies an identifier and a colon before the exception class identifier, the identifier will represent the exception object during execution of the statement that follows **on...do**. For example:

```

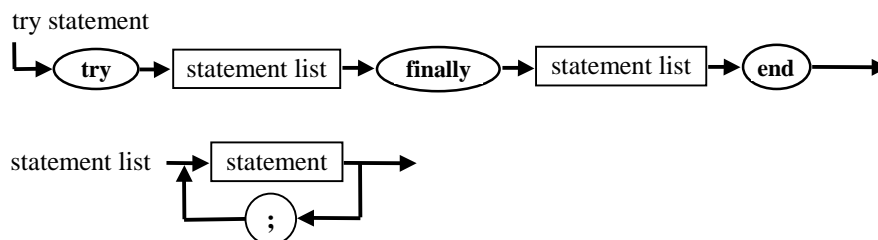
try
...
except
    on X: Exception do ApplicationError(X.Message);
end;

```

The scope of an identifier declared in an exception handler is the statement that follows **on...do**. It hides any other identifiers with the same name in the outer scopes.

The try...finally statement

The **try...finally** statement is used in situations where it is necessary to ensure that a resource allocated by the code section will be released again, even if code execution is interrupted by an exception.



The statements in a **try** *statement list* are executed in the order of appearance until the last statement in the list is executed or until an exception is raised. In either case, the **finally** *statement list* will be executed. If an exception is raised in the **try** *statement list*, control is passed to the **finally** *statement list* and the exception is re-raised.

Note that **try...finally** statements do not handle exceptions. In fact, the **finally** part of the statement does not even know whether an exception has occurred or not.

If an exception occurs in the **finally** statement list itself and it is not handled, it will be re-raised to the outer exception handlers and the original exception will be lost.

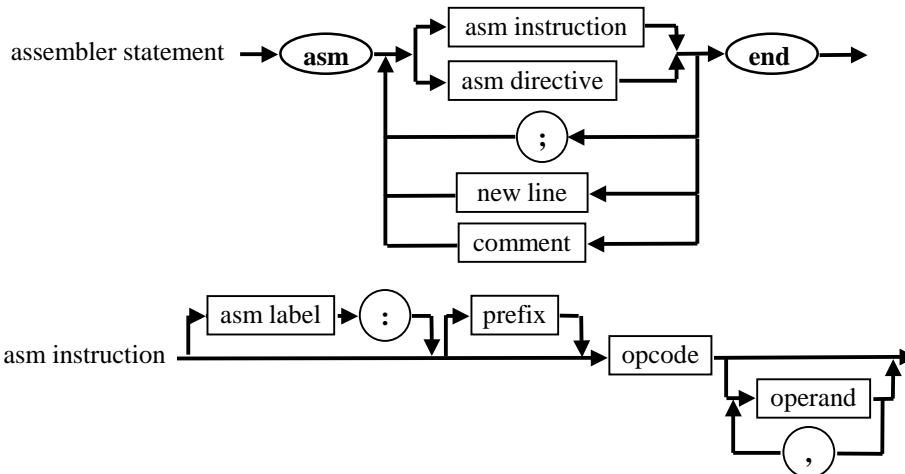
Example:

```
New(P);
try
  DoSomething(P);
finally
  Dispose(P);
end;
```

Assembler statements

The built-in assembler (BASM) allows the writing of 386, 486 and Pentium assembler code inside a Pascal program. Unlike macro assemblers, only a small subset of assembler directives are supported and no macro processing is available.

The built-in assembler is accessed through assembler statements. The syntax of an assembler statement is:



Several **asm** instructions can be written in one line, separated by semicolons. Unlike a macro assembler, only Pascal style comments can be used. It is not possible to insert a comment inside an **asm** instruction.

Assembler statements and assembler procedures and functions should conform to the current register convention. By default, the *ebx*, *esi* and *edi* registers must be preserved by **asm** statements, although the **{&Saves RegList}** or the **{&Alters RegList}** compiler directives can be used to modify the register saving convention used by an **asm** statement. This is the preferred method for embedded **asm** statements. In this case, the directives must be specified inside the procedure or function body just before the reserved word **asm** and will affect all subsequent **asm** statements within the current

statement part, but will not affect the default register saving convention for procedures and functions.

For **asm** procedures and functions, it is better to use the **{&Uses RegList}** compiler directive. It lists those of the registers that must be preserved (by default *ebx*, *esi*, *edi*) but are changed by the **asm** procedure or function. The **{&Uses}** directive causes the listed registers to be pushed on entry and popped on exit. It is not recommended to use the **{&Saves}** or **{&Alters}** directives, because the effect is global when applied to procedures and functions and it will affect all procedures and functions located further down in the source file as well.

Example:

```
function StrLen(Str: PChar): Longint; assembler; {&Uses edi} {&Frame-}
asm
  cld                // Set forward scan direction
  mov     edi,Str    // Load address of Str into edi
  or      ecx,-1     // Set ecx to -1
  xor     eax,eax    // Set eax to 0
  repne  scasb      // Scan forward, until 0 byte found
  sub     eax,ecx    // Get 1-Count value into eax
  sub     eax,2     // Subtract 2 to get function result value
end;
```

Because of the **{&Uses edi}** directive in the example, the compiler automatically saves the value of the *edi* register on entry, and restores it on exit. If the directive had not been present, no registers would all have been saved and restored, with possible negative consequences for the program. VP always assumes that the *ebx*, *esi* and *edi* registers are preserved, so these must either be saved manually or saved by making the compiler aware that they may change by using a **{&Uses}** directive.

Labels

Assembler statements are treated by the compiler as a block with one entry and one exit. As a result, Pascal labels cannot be referenced in BASM and it is not possible to declare Pascal labels in the assembler statement. Only local assembler labels can be used. A local assembler label must start with an @ sign followed by one or more characters; each of them can be either a letter **A...Z**, **a...z**, a digit **0...9**, an underscore **_** or an @ sign. Only the 31 first characters are significant.

Prefixes

A prefix is one of the following:

lock	repz	seges	segds
rep	repne	segcs	segfs
repe	repnz	segss	seggs

Instructions

An instruction is one of the following instruction mnemonics:

aaa	adc	bound	bt	call	cld
aad	add	bsf	btc	cbw	cli
aam	and	bsr	btr	cdq	clts
aas	arpl	bswap	btc	clc	cmc

cmp	fisub	fucompp	jnz	movzx	setna
cmpsb	fisubr	fwait	jo	mul	setnae
cmpsd	fld	fxam	jp	neg	setnb
cmpsw	fld1	fxch	jpe	nop	setnbe
cmpxchg	fldcw	fxtract	jpo	not	setnc
cmpxchg8b	fldenv	fy12x	js	or	setne
cpuid	fldl2e	fy12xp1	jz	out	setng
cwd	fldl2t	hlt	lahf	outsb	setngc
cwde	fldlg2	idiv	lar	outsd	setnl
daa	fldln2	imul	lds	outsw	setnle
das	fldpi	in	lea	pop	setno
dec	fldz	inc	leave	popa	setnp
div	fmul	insb	leaved	popad	setns
enter	fmulp	insd	leavew	popf	setnz
enterd	fnclex	insw	les	popfd	seto
enterw	fndisi	int	lfs	push	setp
f2xm1	fneni	into	lgdt	pusha	setpe
fabs	fninit	invd	lgs	pushad	setpo
fadd	fnop	invlpg	lidt	pushf	sets
faddp	fnsave	iret	lldt	pushfd	setz
fbsd	fnstcw	iretd	lmsw	rcl	sgdt
fbstp	fnstenv	ja	lodsrb	rcr	shl
fchs	fnstsw	jae	lodsd	rdmsr	shld
felex	fpatan	jb	lodsw	rdtsc	shr
fcom	fprem	jbe	loop	ret	shrd
fcomp	fprem1	jc	loopd	retf	sidt
fcompp	fptan	jcxz	loopde	retn	sldt
fcos	frndint	je	loopdne	rol	smsw
fdecstp	frstor	jecxz	loopdnz	ror	stc
fdisi	fsave	jg	loopdz	rsm	std
fdiv	fscale	jge	loope	sahf	sti
fdivp	fsetpm	jl	loopne	sal	stosb
fdivr	fsin	jle	loopnz	sar	stosd
fdivrp	fsincos	jmp	loopw	sbb	stosw
feni	fsqrt	jna	loopwe	scasb	str
ffree	fst	jnae	loopwne	scasd	sub
fiadd	fsrcw	jnb	loopwnz	scasw	test
ficom	fstenv	jnbe	loopwz	seta	verr
ficompp	fstp	jnc	loopz	setae	verw
fidiv	fstsw	jne	lsl	setb	wait
fidivr	fsub	jnge	lss	setbe	wbinvd
file	fsubp	jng	ltr	setc	wrmsr
fimul	fsubr	jnl	mov	sete	xadd
fincstp	fsubrp	jnlc	movsb	setg	xchg
finit	ftst	jno	movsd	setge	xlat
fist	fucom	jnp	movsw	setl	xlatb
fistp	fucomp	jns	movsx	setle	xor

Pseudo instructions

Two pseudo-instructions are supported: *Align* and *PopArgs*.

The *Align* pseudo instruction accepts one constant integer operand that can have the following values:

Value	Description
1	No code is generated
2	The compiler generates code to align the next instruction to a word (2.byte) boundary
4	The compiler generates code to align the next instruction to a double word (4.byte) boundary.

Code generated by the *Align* instruction does not change any CPU register, except *eip* (The instruction pointer).

The *PopArgs* pseudo instruction accepts one constant integer operand that defines the size of the arguments that should be popped out after the current subprogram returns. It has meaning only for assembler procedures and functions. This value is used for RET instruction, generated by the **end** statement of the assembler procedure or function.

Asm directives

Virtual Pascal supports the following directives:

Directive	Meaning	Operand range	Size bytes
db	Define byte	-128...255	1
dw	Define word	-32,768...65,535	2
dd	Define double word	-2,147,483,648...4,294,967,295	4
df	Define far 48 bit pointer	-2,147,483,648...4,294,967,295	6
dp	Define far 48 bit pointer	-2,147,483,648...4,294,967,295	6

These directives are used to define data of the corresponding size. Each operand of the directives must be a value within the specified range; multiple operands must be separated by commas. The operand can be either an integer constant or a literal string constant. For **db** a literal string can be of any length, for **dw** it should be not longer than 2 characters, for **dd** not longer than 4 characters and as for **df** and **dp**, its length may not exceed 6 characters.

Definitions of floating point values using **dd**, **dq** and **dt** are not supported. These values must be declared as typed constants of *Single*, *Double*, *Comp* or *Extended* type.

The data generated by these directives is stored in the code segment. To define data in the initialised or uninitialised data segment, Pascal variables or typed constants should be used instead.

It is not possible to specify the name of a data value before a directive name. To mark the generated data, a local label can be used:

```
@@PowerOf10Table: dd 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000
```

Operands

Assembler instruction operands are expressions constructed from constants, registers, symbols and operators. BASM supports MASM style syntax for the operands. The exact syntax definition is beyond the scope of this chapter and can be found in macro assembler manuals. Assembler operands must be separated by commas. See the *Intel manual* for a detailed description of the operands for a particular instruction.

The following words are reserved in the built-in assembler:

ah	and	bh	bp
al	ax	bl	bx

byte	dr6	fword	shr
ch	dr7	gs	si
cl	ds	high	small
cr0	dword	large	sp
cr2	dx	low	ss
cr3	eax	mod	st
cr4	ebp	near	tbyte
cs	ebx	not	tr3
cx	ecx	offset	tr4
dh	edi	or	tr5
di	edx	ptr	tr6
dl	es	pword	tr7
dr0	esi	qword	type
dr1	esp	seg	word
dr2	far	shl	xor
dr3	fs	short	

If a Pascal symbol name clashes with one of these reserved words, the ampersand (&) identifier override operator can be used to tell the built-in assembler to look for the Pascal identifier with the same name instead of the reserved word. For example:

```
var
  Ch: Char;
...
asm
  ...
  inc    &Ch          { Increases the Pascal variable Ch }
  inc    ch           { Increases ch register }
  ...
end;
```

The built-in assembler can be considered as an assembler using Pascal syntax for data definitions. BASM allows referencing of the following symbols:

Symbol	Value	Class	Type
Local label	Address of label	Memory	SHORT
Untyped constant	Value of constant	Immediate	0
Type	0	Immediate	Size of type
Field	Offset of the field	Memory	Size of field
Variable	Address	Memory	Size of variable
Procedure, function or method	Address of entry point	Memory	NEAR
Unit	0	Immediate	0
@Code	Flat CS selector of the code segment	Memory	0FFFFFFF0h
@Data	Flat DS selector of the code segment	Memory	0FFFFFFF0h
@Result	Address of the function result	Memory	Size of variable
@Locals	Total size of local variables	Immediate	0
@Params	Total size of parameters	Immediate	0
@Uses	Total size of the registers specified in the {&Uses RegList} directive	Immediate	0

@Code and @Data symbols must be used only with the SEG operator like this:

asm

```
...
    mov    eax, SEG @Data
```

end;

@Result, @Locals, @Params and @Uses can be used only inside assembler procedures or functions.

The following symbols can not be accessed from the built-in assembler:

- Standard procedures and functions declared in the *System* unit, like *ReadLn*.
- *Mem* and *Port* arrays.
- String, floating point and set untyped constants.
- **Inline** procedures and functions.
- Pascal labels
- @Result, @Locals, @Params and @Uses symbols inside a Pascal procedure or inside an embedded **asm** statement.

In **asm** statements within a Pascal statement part, local variables are always accessed via the *ebp* register. If a local variable is accessed using the built-in assembler, it will not be allocated in a CPU register, even in the **{&Optimize+}** state.

Inside assembler procedures or functions local variables are accessed via *ebp* in the **{&Frame+}** state or via *esp* in the **{&Frame-}** state.

The assembler automatically adds [*ebp*] or [*esp*] in all references to local variables. However in the **{&Frame-}** state references to local variables are valid only if the value of the *esp* register has not changed within the body of the **asm** procedure or function (there are no pushes or pops). Disabling the

frames is usually used for small and simple procedures and functions in order to generate more efficient code.

The type of an assembler expression is returned by the TYPE operator. The following types are predefined in BASM and can be used in addition to any Pascal types:

Symbol	Type
byte	1
word	2
dword	4
pword	6
fword	6
qword	8
tbyte	10
near	0FFFFFFEh
far	0FFFFFFFh
short	0FFFFFFDh

Example

{ Fill Count number of double words (4 bytes) with Value }

procedure FillDWord(**var** Dest; Count, Value: Longint); *{&Uses edi} {&Frame-}*

asm

```
cld  
mov edi, Dest  
mov eax, Value  
mov ecx, Count  
rep stosd
```

end;

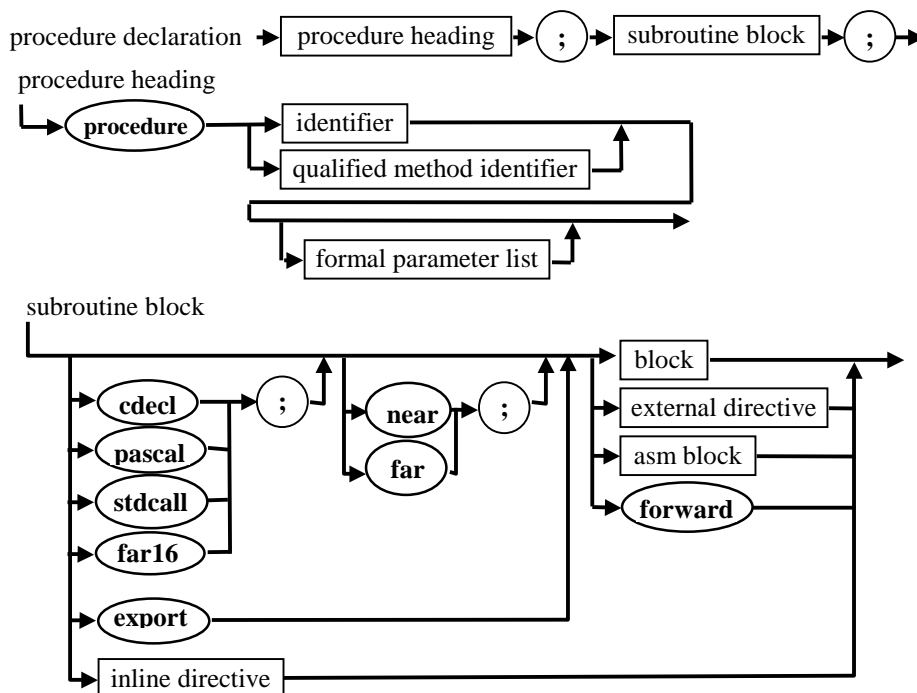
CHAPTER 9

Procedures and functions

Procedures and functions allow the logic of a program to be divided into separate tasks, making it much easier to develop and understand the program. Each procedure or function is made up of a heading, which names the procedure or function and a procedure or function block, which contains all the actions of the routine. To invoke a procedure or function, a procedure or function call is used as explained later in this chapter.

Procedure declarations

A procedure or function must be declared before being used in a program. The syntax for a procedure declaration is shown on the diagrams below:



The procedure heading describes the procedure identifier and specifies any optional formal parameters. The subroutine block immediately follows the heading and contains all statements to be executed on activation. In case of a forward declaration by means of a **forward** procedural directive or a declaration in the interface part of a unit, the block containing the statement part of the procedure should be omitted. A full declaration with statement part must be declared later on. A procedure is called by a procedure statement, which consists of the procedure's identifier and actual parameters in place of the corresponding formal parameters. Within the statement part of a subroutine block, a procedure can call itself while executing, i. e. it can be a recursive call.

Examples:

```

procedure UpStr(var S: String);
var
  I: Integer;
begin
  
```

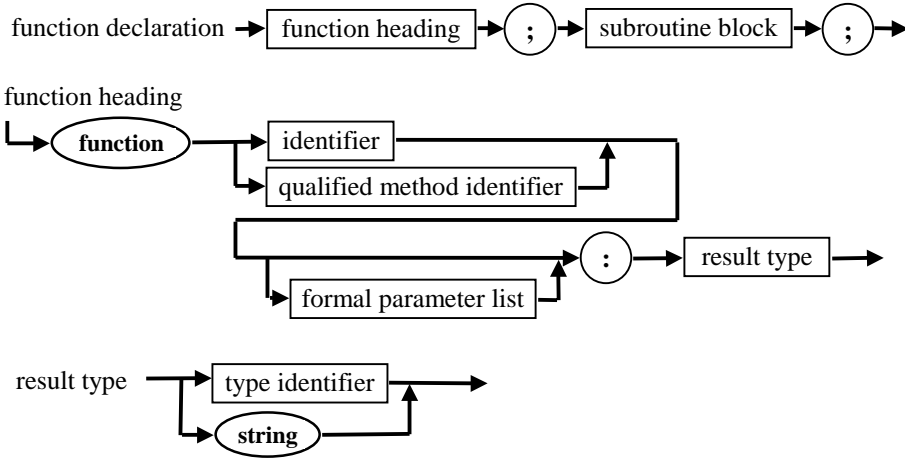
```

for I := 1 to Length(S) do
    S[I] := UpCase(S[I]);
end;

```

Function declarations

A function declaration is similar to a procedure declaration, but the reserved word **function** replaces the keyword **procedure** and the type of the function follows the parameter list.



The function heading introduces the function identifier and optionally specifies any parameters and the type of the function result. The subroutine block has the same syntax and meaning as the subroutine block of a procedure. When a function is used as a factor in an expression, the actual parameters substitute the formal parameters.

After the function has executed, the value of the factor is the return value of the function. Inside the statement part of the function, the function identifier can be used in assignment statements to assign a value to the function result. If the function identifier is used as a factor of an expression within the function statement part, it is interpreted as a recursive call. In the **{SX+, Delphi+}** state, an implicit local variable *Result* is also defined to reference the function result. Unlike the function identifier, it is possible not just to assign a value to the *Result* variable, but also to read the current value of it, pass it as a variable parameter to other procedures or functions, etc. The function block should contain at least one statement that assigns a value to the function result. If there is more than one statement of this kind, the last one will assign the return value to the function identifier. If there are no such statements, the function result is undefined. If extended syntax is enabled (in the **{SX+}** state), a function can be used as a procedure in a procedure call statement. In this case the result value of the function will be discarded.

Examples:

```

function UpStr(const S: String): String;
var
    I: Integer;
begin
    SetLength(Result, Length(S));
    for I := 1 to Length(S) do
        Result[I] := UpCase(S[I]);
end;

```

```

function Faculty(X: Cardinal): Cardinal;

```



```

begin
  if X <= 0 then
    Result := 1
  else
    Result := X * Faculty(X-1);    // Recursive call
end;

```

When writing assembler procedures and functions, the following conventions used for returning function result values must be taken into consideration:

- Ordinal type function results are returned in the *eax* CPU register or its part depending on the result value size as follows: byte sized values are returned in AL, word sized (2-bytes) in *ax* and double word sized (4-bytes) values are returned in *eax*.
- Pointer, class, class reference and global procedure pointer type function results are returned in the *eax* register.
- Real type function result values (*Single*, *Double*, *Extended*, *Comp*, *Real* and *Currency*) are returned in the topmost coprocessor register ST(0).
- For a string or method pointer result, the *caller* allocates local memory for the function result and pushes its address as an implicit additional parameter just before any other explicit parameters. The function must not pop it from the stack.
- For array, records, object and set type function results, byte sized values are returned in *al*, word sized are returned in *ax*, double word sized are returned in *eax*. For all other values, the caller allocates the local memory for the function result and pushes its address as an implicit additional parameter just before any other explicit parameters. The function must not pop it from the stack.

Near and far declarations

All procedure and function calls in a flat memory model are **near**. Virtual Pascal supports **near** and **far** procedural directives for compatibility with the 16-bit Borland Pascal compiler. The standard procedural directives **near** and **far** are ignored.

Export declarations

There are several ways of exporting symbols from DLLs (refer to page 130 for details). An export declaration is one of them. The **export** directive exports a procedure or function by its name and makes it available to other run-time modules – i.e. applications and dynamic link libraries.

The **export** directive can only be applied to procedures or functions declared either in the interface part of a unit or in the source of the program or library.

Forward declarations

A **forward** declaration is a procedure or function declaration that contains the **forward** directive instead of a block with a statement part. Later in the program there must be a defining declaration of the procedure containing the statement part. The parameter list and the function result can be omitted in the defining declaration. If they are repeated in the defining declaration, there must be no difference in the order, types, names of parameters or the type of the function result, from the **forward** declaration.

The declaration of a procedure or function in the **interface** part of a unit is considered to be forward by default and the **forward** directive may not be used. The defining declaration must be present in the **implementation** part of the unit.

The **forward** declaration and the defining declaration together make up a complete procedure or function declaration.

Example:

```
procedure RecursiveProc1(X: Integer); forward;
```

```
procedure RecursiveProc2(I,J :Integer);
```

```
begin
```

```
...
```

```
    RecursiveProc1(J);
```

```
...
```

```
end;
```

```
procedure RecursiveProc1;
```

```
begin
```

```
...
```

```
    RecursiveProc2(X, X*2);
```

```
...
```

```
end;
```

In the example, *RecursiveProc2* needs to call *RecursiveProc1*, and vice versa. This can be achieved by using a **forward** declaration as shown.

Calling conventions

The calling convention specifies how parameters are passed to a procedure or function and who is responsible for cleaning up the stack after the call. The following standard procedural directives set up the default language calling conventions.

State	Convention	Parameter push order	Stack cleanup
Cdecl	C language	Last to first	Caller
Far16	16-bit Pascal	First to last	Called routine
StdCall	STDCALL	Last to first	Called routine
Pascal	32-bit Pascal	First to last	Called routine

If none of the directives are specified, the calling convention is taken from the settings of the **&Cdecl**, **&Far16** or **\$StdCall** compiler directives. The enabled state chooses the corresponding calling convention, specifying any of them in the disabled state enables the **Pascal** calling convention (for details see the description of the **&Cdecl** directive on page 151).

Virtual Pascal supports **far16** calling convention without thunking, which means that it is not required to write a thunk wrapper routine for calling 16-bit code from 32-bit code; this is performed seamlessly via a special run-time library helper routine. The **far16** calling convention is provided for external routines only as the compiler itself always generates 32-bit code. A procedure or function that uses the **far16** calling convention must have no more than 16 parameters. By means of the **far16** calling convention, it is possible to call procedures and functions located in 16-bit DLLs, which is very useful in OS/2 where the text mode functions (*Vio, Kbd, Mou and MonCalls*) still use a 16-bit interface.

Note

Since **far16** procedures and functions are called via a special helper routine, it is not possible to call them directly from built-in assembler. In this case a normal pascal wrapper routine which will call the **far16** routine must be written.

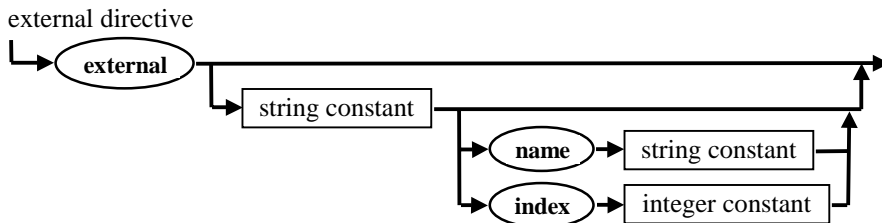
Example:

```
{&Cdecl+}
function CFun(P1: Integer): LongBool;
{&Cdecl-}
function PascalFun(P1: Integer): Boolean;
procedure CProc(P1: Integer; P2: Byte); cdecl;
```

In the example, both *CFun* and *CProc* use the C calling convention, whereas *PascalFun* uses the default Pascal calling convention.

External declarations

External declarations make it possible to interface with separately compiled procedures and functions written in assembly or other programming languages. They allow procedures and functions to be imported from DLLs. There are several other ways of importing procedures and functions (for more information on import refer to page 128).



External directives can consist of the reserved word **external** by itself. The actual implementation of the procedure or function may be either in an external object file, an object library or in a dynamic link library. The **{\$L filename}** directive should be used to inform the compiler about the name of the object file, object library or import library.

If an external directive specifies a dynamic link library name and (optionally) an import name or an import ordinal number, it can be used to import a procedure or a function from a dynamic link library, in which case an import library is not required.

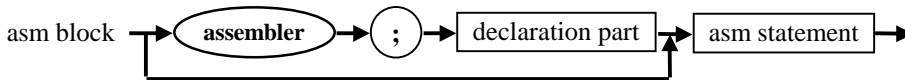
Example:

```
function inet_makeaddr(net: ULong; ina: ULong): ULong;
external 'TCP32DLL' index 8;
```

The **external** directive takes the place of the statement part that should otherwise be present. It must be used only in the defining declaration of a procedure or a function and may not be used as a **forward** declaration.

Assembler declarations

Using **assembler** declarations, it is possible to write a procedure or function in the built-in assembly language. Refer to page 104 for more information on BASM.



Inline declarations

The **inline** directive makes it possible to write macro-style procedures and functions that are expanded rather than called when they are invoked.



When an **inline** procedure or function is invoked, the body of the procedure or function is expanded and takes the place of the call; the actual parameters replace the formal parameters. Since the body of the procedure or function does not contain the code as it is, it is not possible to apply the @ operator to an **inline** procedure or function, or assign the **inline** procedure or function to a variable of a procedural type. The following restrictions apply to inline procedure definitions:

- **inline** procedures and functions may not be nested.
- An **inline** procedure or function may not contain typed constant declarations.
- An **inline** procedure or function may not contain parameters that need to be copied to the local stack of the routine.
- An **inline** procedure or function may not be called from another **inline** routine.
- An **inline** procedure or function can not contain **asm** statements.

Since the compiler generates code for the entire body of the procedure or function each time it is called, it makes sense to use the **inline** directive only for small procedures and functions, or the executable file may grow significantly in size.

Example:

```

function MaxL(A, B: Longint): Longint; inline;
begin
  if A > B then
    Result := A
  else
    Result := B;
end;
  
```

Method declarations

The declaration of a method within a class type corresponds to a **forward** declaration of that method. Later in the same module, the method must be implemented by a defining declaration.

The defining declaration of procedure or function methods is the same as that of regular procedure or function declarations, with the exception that the method heading should contain a *qualified method identifier*, consisting of an object or a class-type identifier followed by a period (.) and a method identifier.

The defining declaration of constructors and destructors is similar to a procedure declaration, except that the reserved word **constructor** or **destructor** replaces **procedure**.

The parameter list can be omitted in the defining declaration (but if it is repeated there must be no difference in the order, types, names of parameters and function result type from the method declaration in the object or class type).

The defining declaration of a method always contains an implicit parameter with the identifier *Self*, which corresponds to a formal parameter of the class type. Within the method block, *Self* represents the instance whose method component was designated to activate the method.

The scope of a component identifier in an object or a class type includes all procedure, function, constructor, or destructor blocks implementing all methods of the type. The method block behaves as if it was enclosed by a **with** statement:

with *Self* **do**

MethodStatementPart;

To access a redeclared or an overridden component identifier within a method block, use the reserved word **inherited**. If an identifier is used in conjunction with **inherited**, the search for the identifier starts with the immediate ancestor of the object or class type.

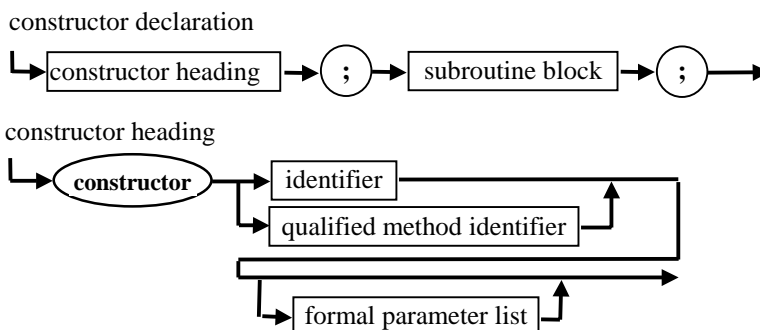
Constructors and destructors

Constructors and destructors are methods that perform construction and destruction of objects. There can be zero or more constructors and destructors for the object or class type. The declaration of constructors and destructors is the same as the declaration of a procedure or function method, but the method heading contains the reserved word **constructor** or **destructor** instead of **procedure** or **function**.

Constructors

A constructor defines the actions associated with creating and initialising of a new object or class. A constructor usually initialises an object or a class by using values passed as parameters to the constructor.

The syntax of a constructor declaration is shown below:



Only constructors in the class model can be virtual; a constructor of an object type can not be virtual. This is because the mechanism of virtual method calls requires the constructor to be called to initialise the pointer to the VMT inside the object instance. Virtual class constructors must be called using a class reference rather than a class instance. Static constructors can be invoked on both a class instance and a class reference.

Dynamic object variables are created by specifying the object's constructors as the second operand to the standard procedure *New* call. The first parameter must be a pointer type or a pointer variable to the object type. If the returned value is **nil**, the constructor call failed. Since no pointer value is returned

for a static object constructor, it instead returns a *Boolean* value indicating whether construction of the object was successful. A return value of *True* indicates success and a return value of *False* indicates failure caused by a call to *Fail* within the constructor.

Note: Constructing an object variable is does *not* automatically clear the allocated memory; the *Init* constructor of *TObject* must be called to do this. This step is important, particularly if the object in question contains fields of *AnsiString* type.

Example:

```
var
  MyObject: TCollection;
begin
  if MyObject.Init then
    begin
      ....
    end
  else
    Writeln('Object could not be initialised; fail was called in the constructor');
end;
```

Construction of class variables is more complex. When a constructor is used to create a new class variable, it is invoked on a class reference and the following sequence of actions takes place:

- A call to the *NewInstance* virtual method is made. The standard behaviour of *TObject.NewInstance* is to allocate storage for a new object on the heap and clear it. This means that it sets the ordinal values of all ordinal type fields to zero, the values of all pointer and class type fields to **nil** and the values of all string fields, including long strings, to the empty string.
- The actions specified in the constructor's statement part are executed.
- The constructor returns a reference to the newly allocated and initialised class instance. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor invoked on a class reference, the *Destroy* destructor is automatically called to destroy the unfinished object.

If a constructor is called on a class instance, it only executes the actions specified by the user in the constructor's statement part. In this case a new object is not allocated and cleared and the constructor does not return a class instance. Usually, a constructor is invoked on a class instance to execute an inherited constructor. This is done by prefixing the constructor name with the reserved word **inherited**.

When a constructor is activated, it usually first calls an inherited constructor to initialise the inherited fields of the object. Following this, the constructor initialises the fields that were introduced in the class. Unless a field's default value is non-zero, there is no need to initialise the field in a constructor.

A virtual class constructor is equivalent to a static constructor invoked on a class type identifier. In addition, a virtual constructor can be invoked on a variable reference of a class reference type, which allows polymorphic construction of classes whose types are not known at compile time.

Example:

```
type
  TMyClass = class(TObject)
    fData: Integer;
    constructor Create;
  end;
```

```

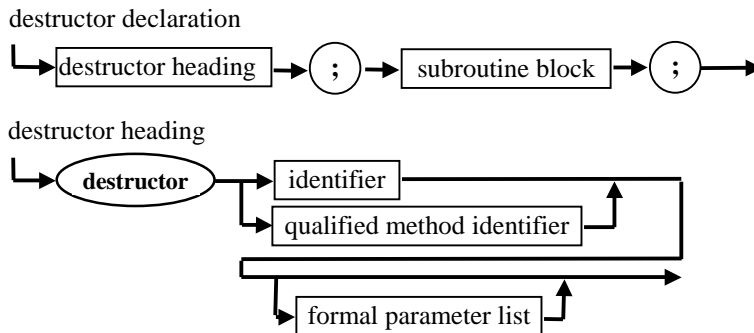
constructor TMyClass.Create;
begin
  inherited Create;           // No memory allocated here
  fData := 8;
end;

var
  MyClass: TMyClass;
begin
  try
    MyClass := TMyClass.Create;   // Allocate new object
    try
      Writeln(MyClass.fData);
    finally
      MyClass.Free;             // Free memory used by class instance
    end;
  except
    on E:Exception do
      Writeln('Error: ',E.Message);
  end;
end;

```

Destructors

Destructors define the actions associated with destroying object and class instances. The syntax of a destructor declaration is the following:



Dynamic object variables are destroyed by specifying the object's constructors as a second operand to a call to the standard procedure *Dispose*. The first parameter must be a pointer variable to the object type.

Class variables are always dynamically allocated. When a destructor is called, the virtual method *FreeInstance* is automatically invoked after the statement part of the destructor has executed. The default *TObject.FreeInstance* method deallocates the memory used by the class instance. Normally, a destructor calls a destructor of its ancestor to let it perform relevant clean-up actions required. To avoid situations where a class instance would be attempted to be freed multiple times, *FreeInstance* is not invoked when a destructor is called using the reserved word **inherited**.

Although it is possible to declare multiple destructors for a class, it is advisable only to use overrides of the inherited *Destroy* destructor. *Destroy* is declared in the *TObject* class and is used for a special purpose. If an exception is raised during the execution of a constructor, the *Destroy* destructor is called to destroy the unfinished object. Even if more destructors are defined, they will not be called in case of an exception. For this reason, descendants of the *Destroy* destructor must be prepared to

handle destruction of *partially constructed* objects. Since all fields of a new object are set to zero before the statement part is executed, any class-type or pointer-type fields in a partially constructed object will be **nil** and destructors should check for **nil** values before performing operations on class-type or pointer-type fields. Since the *Destroy* destructor does not make such a check, direct calls to *Destroy* are not recommended. It is better to call the *Free* method declared in *TObject* which checks whether *Self* is **nil** and only invokes *Destroy* if *Self* is not **nil**.

Class methods

Class methods are methods of a type, rather than of a particular instance of a type and can be called without first constructing a class instance. The implementation of a class method must not depend on run-time values of any fields of instances of the class.

A class method is declared by specifying the reserved word **class** in front of the method declaration. The defining declaration must also start with the reserved word **class**. For example, the *InstanceSize* class function is declared in the *System* unit as follows:

```
class function TObject.InstanceSize: Longint;
asm
...
end;
```

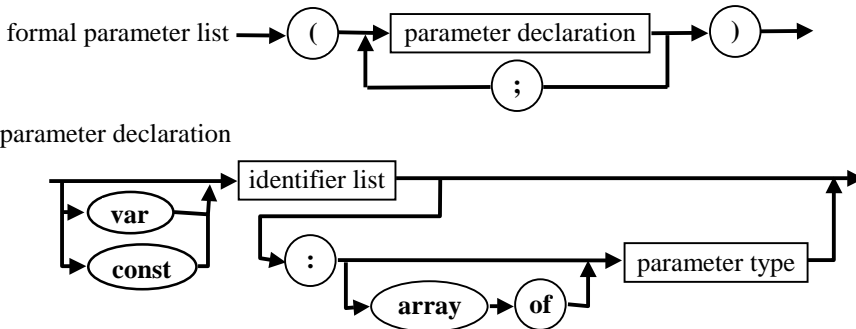
InstanceSize returns the number of bytes used by an instance of a given class. Other class methods defined include *ClassName*, which returns the name of a class as a string value, and *ClassNameIs*, which returns *True* if the string passed as parameter is identical to the name of the class.

In the implementation of a class method, the identifier *Self* represents the class for which the method was activated. The type of *Self* is **class of ClassType**, where *ClassType* is the class type for which the method is implemented.

It is possible to call a class method by using either a class reference or a class instance. When a class method is called as a normal method using an instance of a class type, the class reference of the class instance is passed as the *Self* parameter.

Parameters

A *formal parameter list* follows a procedure or function heading in procedure and function declarations. The parameters declared in the formal parameter list all have local scope and are valid for the duration of the procedure or function. The diagrams below define the syntax of a formal parameter list:



The formal parameter list can contain value, constant, variable, untyped parameters and open parameters.

Value parameter declarations include a list of identifiers, followed by a type. Constant parameter declarations contain an identifier list preceded by the reserved word **const** and followed by a type. A variable parameter declaration contains the identifier list preceded by the reserved word **var** followed by a type, and untyped parameter declarations has an identifier list, preceded by **var** or **const**; the list is not followed by a type.

String and array type parameters can be *open parameters*, which means that they can be of variable size.

Parameters can be passed to procedures or functions in two ways: by *reference* or by *value*. If a parameter is passed by *reference*, a pointer, pointing to the actual storage location is passed. If a parameter is passed by *value*, the actual value is passed.

Value parameters

When a formal parameter is a value parameter, it can be used like a local variable to the procedure or function. The corresponding actual parameter initialises the value parameter when the function is called. The actual parameter must be an expression and the result of the expression cannot be of file type or any structured type containing a file type. The actual parameter must be assignment-compatible with the type of the formal parameter. Changes made to a formal value parameter have no effect on the actual parameter.

Value parameters can be passed by value or by reference depending on their type and size. In general, if the value parameter has a size of 1, 2, 4 or 8 bytes, the actual value is pushed onto the stack - the parameter is passed by value. Otherwise, a pointer to the value is pushed onto the stack and the routine then copies the value into a local storage location - the parameter is passed by reference. Parameters of type *Extended* are the exception to this rule - they are passed by directly pushing the value on the stack.

To maintain double word alignment of the stack (which is desirable for reasons of performance), values pushed on the stack are always double word aligned. This means that byte and word-sized parameters always are passed as double words with the low byte or word containing the parameter.

A string-type parameter is always passed as a pointer to the value.

Constant parameters

When a formal parameter is a constant parameter, it can be used like a local read-only variable in the procedure or function. The corresponding actual parameter substitutes the constant parameter when the function is called. The actual parameter must be an expression and the result of the expression cannot be of file type or any structured type containing a file type. The actual parameter must be assignment-compatible with the type of the formal parameter. A constant formal parameter can not be assigned values, or be passed as an actual variable parameter to another procedure or function.

Constant parameters are passed in the same way as value parameters. However, when a value needs to be copied to the local stack of the routine, constant parameters are not copied as the compiler makes sure the variable is not written to. For this reason, it is better to use a constant formal parameter instead of a value formal parameter, if a formal parameter has the same value during the execution of a procedure or function. This protects it against accidental assignments and also lets the compiler generate more efficient code.

Example:

```
procedure OutputText(const S: String);
begin
  WriteLn( S );
```

end;

Variable parameters

When a formal parameter is a variable parameter, the corresponding actual parameter must be a variable reference: it can not be an expression. The formal parameter represents the actual variable during the activation of the procedure or function and any reference to the formal parameter operates on the actual parameter. Any changes made to the formal variable parameter will also affect the actual parameter.

The type of the actual parameter must be the same as the type of the formal variable parameter. The exception is untyped variable parameters, which accept a variable reference of any type as the actual parameter. For object types, it is possible to pass an instance of the formal parameter type or any of its descendant types as the actual parameter.

The **\$P** compiler directive controls the meaning of variable parameters declared using the **string** reserved word. In the **{\$P+}** state, variable parameters declared using the **string** reserved word are open string parameters. The actual parameter of an open string parameter can be a variable of any string type and within the procedure or function, the size attribute (maximum length) of the formal parameter will be the same as that of the actual parameter. In the **{\$P-}** state (default), variable parameters declared using the **string** reserved word are normal variable parameters. Regardless of the setting of the **\$P** directive, the *OpenString* identifier can always be used to declare open string parameters.

Variable parameters are always passed by reference.

Untyped parameters

If no type is specified in the declaration of a **var** or **const** parameter, it is an untyped parameter. A variable or constant reference of any type can be used as an actual parameter. The formal untyped parameter is incompatible with variables of all other types, but can be given a specific type through a variable typecast.

Example:

```
function MemCmp(const Src1, Src2; Count: Cardinal): Integer;
var
  P,P1,P2: PChar;
begin
  if Count = 0 then
    Result := 0
  else
    begin
      P1 := @Src1;
      P2 := @Src2;
      P := P1 + Count;
      while (P1 < P) and (P1^ = P2^) do
        begin
          Inc(P1);
          Inc(P2);
        end;
      if (P1-1)^ = (P2-1)^ then Result := 0
      else if (P1-1)^ < (P2-1)^ then Result := -1
      else Result := 1;
    end;
  end;
```

end;

This function can be used to compare two memory buffers of any size. It accepts variables of any size in place of the *Src1* and *Src2*.

Open parameters

Open parameters are used to pass strings and arrays of any size as actual parameters to the same procedure or function.

- Open string parameters

Open string parameters are used to ensure the safe passing of variable parameters of a short string type. It is possible to use the **{*\$V-*}** state to pass a string parameters of any size as an actual parameter, but it is not safe to do so in those cases where the maximum declared size of the actual parameter is less than the maximum declared size of the formal one. Even in the **{*\$R+*}** state, the compiler can not check the indexing of these string parameters.

For an open string parameter, the actual parameter can be a variable of any string type. Unlike normal variable string parameters, information about the maximum declared size of the actual parameter is made available, making it possible to check the indexing of the formal parameter in the **{*\$R+*}** state. It is also possible to query the maximum size of the actual parameter at run-time. The maximum declared length is passed automatically as an additional parameter to the procedure or function just after the open string parameter address. The *High* standard function returns this value and the *SizeOf* standard function returns this value plus one (the *Low* standard function returns zero).

When a long string is passed to a routine expecting an open string parameter, the string is truncated to 255 characters inside the routine.

There are two ways to declare an open string parameter. The built-in *OpenString* identifier, declared in the *System* unit, can be used to denote an open string parameter. It can only be used in parameter declarations and may not be used in any other context. In the **{*\$P+*}** state, a variable parameter declared using the reserved word **string** is treated to be an open-string parameter. Value and constant formal parameters declared using the *OpenString* identifier or the reserved word **string** in the **{*\$P+*}** state are not considered to be open string parameters.

Open string parameters cannot be passed as regular variable parameters to other procedures or functions, but they can be passed as open string parameters. Open string parameters are passed by first pushing a pointer to the string and then pushing a double word containing the maximum length of the string.

- Open array parameters

Parameters declared with the **array of *T*** syntax, where *T* is a type identifier, is an *open array parameter*. Unlike ordinary array declarations, the dimension of the array is omitted and is assumed to be in the range $0..N-1$, where *N* is the number of elements in the corresponding actual parameter.

Open array parameters allow the passing of arrays of varying dimensions as the actual parameter. It is also possible to pass a simple variable of type *T*. This special case is considered to be an array with one element of type *T*.

The *Low*, *High* and *SizeOf* standard functions work for an open array parameter in the same way as for normal arrays, but the result is based on the actual array parameter. *Low* returns zero, *High* returns $N-1$ and *SizeOf* returns $N*SizeOf(T)$.

If type *T* is *Char*, the actual parameter may be a string constant. In this case a string is converted to a packed string and is passed as a parameter. The empty string is passed as a string with the dimension one, containing only the #0 character.

A formal array parameter can be accessed by elements only: it is not possible to make assignments to an entire array.

As other parameters, open array parameters can be value, constant or variable parameters with the same meaning as for ordinary parameters. Value parameters are copied to the local stack, constant and variable parameters are passed by reference. The difference between them is that elements of a constant array parameter can not be modified.

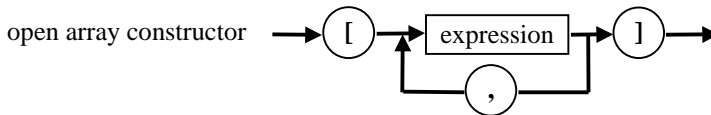
An open array parameter can be passed to other procedures and functions either as an open array parameter or as an untyped variable parameter. Open array parameters are passed by first pushing a pointer to the array and then pushing a double word containing the number of elements in the array less one.

Example:

```
procedure WriteLines(const A: array of PChar);  
var  
    I: Cardinal;  
begin  
    for I := Low(A) to High(A) do  
        WriteLn(A[I]);  
end;  
  
begin  
    WriteLines( ['This is the first line', 'This is the second line' ] );  
end.
```

Open array constructors

Open array constructors are used to construct open array parameters directly within procedure and function calls. An open array constructor has the following syntax:



An open array constructor can be used as an actual parameter in a procedure or function call, if the formal parameter in a procedure or function is an open array value or open-array constant parameter. All expressions of the open array constructor must be assignment compatible with the element type of the open array parameter. When used, the open array constructor creates a temporary array variable, the elements of which are initialised with the values of the expressions.

Example:

It is possible to make the following call to the *WriteLines* procedure declared above:

```
var  
    Row1, Row2, Row3: PChar;  
begin  
    ...  
    WriteLines(['---Table header ---', Row1, Row2, Row3]);  
end;
```

Type variant open array parameters

Parameters declared with the **array of const** syntax are *type variant open array parameters*. They are used to pass open arrays of expressions of different types to a procedure or function.

Type variant open array parameters can also be declared with the syntax **array of TVarRec**, where *TVarRec* is a variant record type declared in the *System* unit. It can represent values of integer, boolean, character, real, string, pointer, class and class reference types. The declaration of *TVarRec* is:

type

```
TVarRec = record
case Vtype: Byte of
  vtInteger: (VInteger: Longint);
  vtBoolean: (VBoolean: Boolean);
  vtChar: (VChar: Char);
  vtExtended: (VExtended: PExtended);
  vtString: (VString: PShortString);
  vtPointer: (VPointer: Pointer);
  vtPChar: (VPChar: PChar);
  vtObject: (VObject: TObject);
  vtClass: (VClass: TClass);
  vtCurrency: (VCurrency: PCurrency);
  vtAnsiString: (VAnsiString: Pointer);
end;
```

The *vtXXX* value type constants are declared in the *System* unit as follows:

const

```
vtInteger      = 0;
vtBoolean      = 1;
vtChar         = 2;
vtExtended     = 3;
vtString       = 4;
vtPointer      = 5;
vtPChar        = 6;
vtObject       = 7;
vtClass        = 8;
vtCurrency     = 9;
vtAnsiString   = 10;
```

If a type variant open array value parameter or type variant open array constant parameter is used as a formal parameter of a procedure or function, an open array constructor can be used as the corresponding actual parameter. In this case the open array constructor creates a temporary type variant open array parameter and initialises its elements with the values of the expressions of the open array constructor.

The table below represents the possible types of the expressions in a type variant open array constructor and the corresponding value type codes.

Type code	Expression type
<i>vtInteger</i>	Any integer type
<i>vtBoolean</i>	Any boolean type
<i>vtChar</i>	Any character type
<i>vtExtended</i>	Any real type
<i>vtString</i>	Any string type
<i>vtPointer</i>	Any pointer type except <i>PChar</i>
<i>vtPChar</i>	<i>PChar</i> or array [0..X] of <i>Char</i>
<i>vtObject</i>	Any class type
<i>vtClass</i>	Any class reference type
<i>vtCurrency</i>	A pointer to a <i>TCurrency</i> type
<i>vtAnsiString</i>	A pointer to a long string

Type variant open array parameters are very useful for string formatting routines and are also useful when it is necessary to pass a variable number of parameters of any type together with the appropriate type information.

Example:

```
uses
  SysUtils;

function IntSum(const A: array of const): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    with A[I] do
      case vType of
        vtInteger: Inc(Result, VInteger);
        vtString: Inc(Result, StrToInt(VString^));
        vtAnsiString: Inc(Result, StrToInt(AnsiString(VAnsiString)));
        vtPChar: Inc(Result, StrToInt(StrPas(VPChar)));
        vtExtended: Inc(Result, Round(VExtended^));
      else
        raise EConvertError.Create('Invalid argument to IntSum');
      end;
    end;
  end;

var
  S: String;

begin
  S := '48';
  WriteLn( IntSum( [ 32, '71', s ] );
end.
```

Dynamic link libraries

What is a DLL?

DLLs are libraries which are linked to a program at load time or run time. When a DLL is loaded, its code, data or resources can be shared among several applications. DLL's code, data and resources are stored in a separate executable file with the extension .DLL.

There are a number of benefits of using DLLs:

- The size of the .EXE file is smaller, because the code in the DLL is not linked into the program that uses the DLL. This is particularly useful for large general-purpose libraries.
- Applications can be changed, extended or upgraded without recompiling and relinking.
- System memory is conserved, if multiple applications are using the same DLL.
- A DLL provides an easy way to interface with applications written in different languages.

The syntax of a DLL source file is very similar to that of a program and uses the reserved word **library** instead of **program** (refer to the page 20 for the exact syntax).

The process of making the symbols (procedures, functions, variables and typed constants) available for other run time modules is called *export*. The process of using symbols (procedures, functions, variables and typed constants), located in a DLL, is called *import*. Internally, the system can reference symbols in the DLL either by its name or by an ordinal position in the entry table. Importing/exporting by means of the symbol name is more flexible, since it avoids the problems associated with keeping track of the exact order of the symbols in the DLL. On the other hand, importing/exporting symbols by their ordinal numbers is faster, since the system does not need to lookup the name of the symbol to resolve the external reference. The size of the application and DLL executables when using this method are also smaller because they do not have to contain the imported/exported symbol names. The disadvantage is, that all subsequent versions of the same DLL must keep the same ordinal numbers, in order for earlier versions of the program using the DLL to work properly with the new version.

There are two ways of creating a DLL:

- Exporting procedures and functions by means of **export** directives. These functions in turn make up the interface of the DLL.
- Creating libraries on per unit basis, using the **{&Export}** directive (Currently available for OS/2 targets only).

The traditional method using export

Procedures and functions are exported by an **exports** clause and are imported by specifying an **external** directives with a **name** or **index** clause. Borland Pascal only supports this method of creating a DLL.

Example:

```
library SmallDLL;  
  
function MaxL(A, B: Longint): Longint;  
begin
```

```
    if A > B then Result := A else Result := B;
end;

function MinL(A, B: Longint): Longint;
begin
    if A < B then Result := A else Result := B;
end;

function SumL(A, B: Longint): Longint; export;
begin
    if A < B then Result := A else Result := B;
end;

exports
    MaxL index 1,           // Export by ordinal
    MinL name 'MinL',      // Export by name
    SumL;                  // Export by name

end.
```

Creating DLLs on a per unit basis

For OS/2 targets, Virtual Pascal provides the ability to export the interface part of a unit from a DLL. In this way, it is possible to create a dynamic version of a unit, where the code of the unit is linked into a DLL. In the DLL source code, the **{&Export UnitName}** compiler directive should be used. This directive causes the entire interface part of the unit, including procedures, functions, objects, methods, variables and typed constants to be exported from the DLL (refer to page 154 for details about the **{&Export UnitName}** directive) In a program using the unit, the **{&Dynamic UnitName}** compiler directive should be present to indicate that a dynamic version of the unit is used. If a DLL contains many units, the **{&Dynamic}** directive alternatively allows the name of the .LIB file (import library of the DLL created by Virtual Pascal) to be specified as the parameter instead of a list of unit names.

Importing symbols from a DLL

There are two ways of importing symbols:

- Static import
- Dynamic import

When static import is used, external references are resolved when the DLL is loaded. Statically imported symbols always refer to the same symbols in the same DLL. With dynamic import, the DLL name, the symbol name or ordinal number is specified at run time.

Static import

There are three ways to declare a symbol for use in a program, if it should be statically imported from a DLL.

- The traditional Borland Pascal method is to use the **name** and **index** standard procedural directives, specifying the name of the dynamic link library and the name of the procedure or function or its ordinal number. Since this method does not allow variables or typed constants to be declared, the use of this method is not recommended.

- External procedures or functions can be declared using the **external** standard directive without the **name** and **index** clause, whether they are imported or not. This method has several benefits over the previous one. The DLL names, symbol names or indices do not have to be entered manually; this process can be automated by using an import library. The *Implib* utility can be used to automatically generate an import library for a DLL. As an additional benefit, it is easy to create applications that use either static or dynamic versions of the libraries by replacing the static libraries by import libraries – the program just has to be relinked.
- For DLLs created on per unit basis (OS/2 only), the import is performed seamlessly. To change from using a statically linked library to a dynamically linked one, no source code needs to be changed, and vice versa. Just include the name of the import library in a **{&Dynamic ImportLibrary}** directive – the Virtual Pascal linker does the rest. This method even allows the creation of dynamic versions of object oriented libraries, like Turbo Vision or Object Professional.

When a dynamic link library is compiled, an import library can be created automatically by the IDE. This has the same name as the primary file of the library and has a .LIB extension and is placed in the directory mentioned first in the Options|Directories|Library directories input box.

- For procedures and functions declared with an **external** directive without **name** and **index** clause, an alternative method to specifying the DLL name and symbol name is to use a module definition file (or an equivalent **{&Linker}** directive) . Remember to take the naming convention of Virtual Pascal into account (see the Appendix on page 187 for details). Use an **IMPORTS** module definition file statement to supply this information; more information about the **IMPORTS** statement can be found on page 143.

Example:

```

program Test;

{&Linker
IMPORTS
    MyDll.SomeFunction
    FunctionByOrdinal = MyDll.2
    RenamedFunction = MyDll.OriginalName
}

```

Dynamic import

Dynamic import allows the name of the DLL as well as the name of the symbol to import to be specified at run time.

Dynamic import is be used in situations where an operating system API is useful to the program, but not required – and the API in question is only available on some installations of the operating system. By using dynamic import, the application can run even on systems not supporting the API in question, because it is loaded dynamically. This method is used in the OS/2 version of the *VpSysLow* unit, to load APIs only available on systems with Presentation Manager installed.

To use dynamic import, first load a DLL and use the handle of the DLL to get the address of the desired entry point in the DLL. When the entry point is no longer needed, it should be freed; when the DLL is no longer needed, the DLL handle should be freed.

In OS/2, this is done by first calling the *DosLoadModule* API function, followed by a call to the *DosQueryProcAddress* function to get the address of the required symbol. The *DosFreeModule* function frees the dynamically loaded module.

In Win32, this is done by first calling the *LoadLibrary* API function, followed by a call to the *GetProcAddress* function. The *CloseHandle* API can be used to free the module.

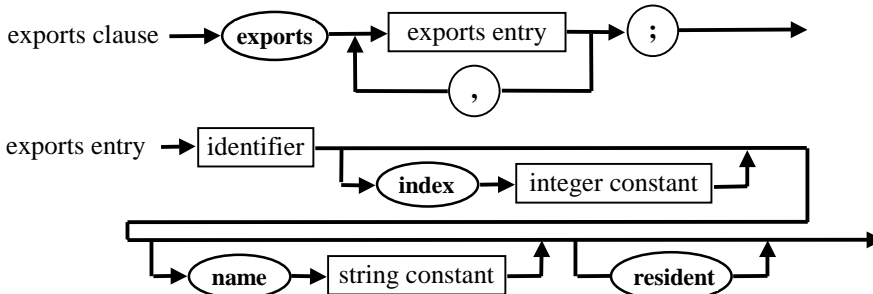
Exporting symbols from a DLL

The following methods can be used to export symbols from a DLL:

- The traditional Borland Pascal method by means of **exports** clause.
- By specifying an **export** directive in the declaration of the procedure or function to be exported.
- By using a module definition file (Or equivalent {&Linker} directive).
- By using the {&Export UnitName} compiler directive for a DLL created on a per unit basis.

The traditional method

The traditional Borland Pascal method of exporting procedures and functions from a DLL is to use an **exports** clause. An **exports** clause may appear anywhere and any number of times in a library's declaration part.



In addition to procedures and functions, Virtual Pascal allows the export of variables and typed constants by means of an **exports** clause. Virtual Pascal does not require procedures and functions to be declared with an **export** directive before being listed in an **exports** clause.

The **index** and **name** directives specify the way the symbol is exported.

If an **index** directive is present, the symbol can be exported both by name and ordinal number. When a **name** directive is specified, it defines the name with which the given procedure or function will be known to other modules. If a **name** is not specified, the original name of the procedure will be used.

If the keyword **resident** is specified, it indicates to the system that the name of the exported procedure or function should be placed in a table of resident names, which stays in memory while the DLL is loaded. This significantly speeds up the search of symbol names at the time of loading a program that uses the DLL.

The export directive

Unlike Borland Pascal, Virtual Pascal does not require exported procedures and functions to be declared using the **export** directive. Rather, all procedures and functions with an **export** directive that are declared in a library source or in the **interface** part of a unit are automatically exported by name. The name used for the export is defined according to the usual Virtual Pascal naming conventions (see Appendix E on page 187).

Using module definition files

As for the import, a module definition file can be used to export procedures, functions, variables and typed constants from the interface part of a unit or from the program or library source file. To do this, an **EXPORTS** module definition file statement should be used (see page 141 for the detailed information).

Exporting the entire interface part (OS/2 only)

It is possible to export the whole interface part of a unit, including procedures, functions, object methods, variables and typed constants. To do this, specify the unit name(s) as parameters to an **&Export** directive in the dynamic library source file (see page 154). See the source file for the DLL version of the Virtual Pascal run-time library for an example (SOURCE\RTL\VPRTL*.*).

Types of DLLs

Depending on whether the static data located in a DLL (typed constants and global variables) is shared or non-shared, DLLs fall into two categories:

- single-data DLLs or subsystems
- multiple-data DLLs or subroutine libraries

Virtual Pascal is capable of creating both types of dynamic link libraries, although subroutine libraries are most commonly used. Note, that source level debugging is not available for procedures and functions located in a library source file, so it usually contains only a library heading, a uses clause and an initialisation part. The debugger is capable of debugging procedures and functions declared in units used by the main library source file.

Subsystems

The term subsystem refers to a DLL that provides a set of services built around a resource and is a term that originates from OS/2. Many of the OS/2 subsystems are implemented in the form of DLLs: The keyboard subsystem (KBDCALLS), which maintains the keyboard, the video subsystem (VIOCALLS), which maintains the video display, etc. The term resource is used in the most general sense of the word and is not to be confused with a Presentation Manager resource.

A subsystem usually has to manage a limited resource for an effectively unlimited number of clients. For this reason, subsystems have common data segment for all applications (clients) that use it. To keep track of its clients, the subsystem needs to know when new clients arrive and when old clients terminate. There are two forms of subsystem initialisation and termination - global and instance. A subsystem can specify either service but not both.

If global initialisation is specified, the initialisation entry point is called only once per activation of the subsystem. When the DLL is first referenced, the operating system allocates the subsystem's static data segments, taking their initial values from the .DLL file. and then calls the subsystem's global initialisation entry point so that the module can do its one-time initialisation.

The second form of initialisation is instance initialisation. The instance initialisation entry point is called in the same way as the global initialisation entry point except that it is called for every new client when that client first attaches to the DLL.

The same applies to the DLL termination.

Ordinarily, it is not necessary to create or know about module definition files to create a DLL. To create a subsystem's dynamic link library, module definition statements are required and typically three statements are required:

```
{&Linker                                ;Avoids using an external file but
                                           stores the definitions in source code
LIBRARY LibName <Init> <Term>          ; specifies the DLL name
DATA SINGLE NONSHARED                  ; only one data segment is created
DESCRIPTION 'Text'                      ; Inserts the specified text into
}                                       ; the library executable file
```

<Init> can be either **INITGLOBAL** or **INITINSTANCE**,

<Term> can be either **TERMGLOBAL** or **TERMINSTANCE**.

The initialisation and termination entry points are the same and correspond to the initialisation statement part of a library. To check whether initialisation or termination is under way, an assembler function similar to this one can be used:

```
{ Returns True for DLL initialisation }
function IsInit: Boolean; {&USES None} {&FRAME-}
asm
    cmp  DWord Ptr [ebp+0Ch],0
    sete al
end;
```

The function checks a special parameter passed to the DLL initialisation / termination by the operating system; this parameter is 0 for initialisation and 1 for termination.

The usual subsystem initialisation part is the following:

```
begin
    if IsInit then
        begin
            ... Initialisation stuff goes here
        end
    else
        begin
            ... Termination stuff is placed here
        end
end.
```

Subroutine Libraries

Subroutine libraries are the most convenient way of creating a dynamic version of a Pascal unit. Procedures, functions and object methods in the subroutine library are written and executed in the same way as statically linked ones. Units included into a dynamic link library, work exactly as they were linked statically, the only difference being that the actual linking takes place at load time instead of at the time of linking.

The operating system creates unique static unit data (global variables and typed constants) for each application that uses a subroutine dynamic link library. Although subroutine libraries do not need initialisation and termination code, the Virtual Pascal run-time library requires the initialisation code of a DLL to be executed each time the DLL is loaded. This means that the **INITINSTANCE** reserved word must be specified in the **LIBRARY** module definition statement. It is not necessary to use a

module definition file to create a subroutine dynamic link library, although it is possible to do so. Typically, it consists of just three statements:

```

{&Linker
LIBRARY LibName           ; specifies the DLL name
INITINSTANCE
DATA MULTIPLE NONSHARED ; forces OS/2 to create new
                           ; instances of data segments for
                           ; each application that uses the DLL
DESCRIPTION 'Text'       ; Inserts the specified text into
}                           ; the library executable file

```

Important notes

Unlike applications, DLLs do not have their own stack but rely on the stack of the calling application. The method used by Virtual Pascal for determining the available stack space may not be compatible with other compilers and it is recommended to turn off Stack Checking (**\$\$-** state) for all code in a Virtual Pascal DLL that is used by programs written in other languages.

Another important aspect is that each unit should be included only once in a program. In other words, it is imperative that a unit included via a DLL is not also linked statically. A common mistake is to include the *System* unit in a statically linked program when *System* is already included via a DLL. This will not happen when using the **{&Dynamic ImportLibrary}** way of using DLLs (but unfortunately, this only works in OS/2). When using the **{&DynamicUnitNames}** method (Also OS/2 only), **{&Dynamic System}** must be specified to avoid program failure because only the initialisation code of the static version of the *System* unit will be called.

Another tricky thing is that a DLL must always contain some static data. This can be a problem for a small DLL containing only code without any static data. TLINK will fail with a '*General error*' message when linking it and LINK386 will work fine, but the produced DLL can not be loaded by OS/2 (This restriction does not apply to a Windows DLL). To work around this problem, declare a dummy global variable in the library source to force the creation of a non-empty data segment.

Quick DLL examples

The first example illustrates how it is possible to write a DLL using the traditional technique. This is mostly suitable for small, simple DLLs.

```

library Arrays;
function ArrMean(const X: array of Longint): Longint;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(X) to High(X) do Inc(Result, X[I]);
  Result := Result div (High(X) + 1);
end;

function ArrMin(const X: array of Longint): Longint;
var
  I: Integer;
begin
  Result := X[0];
  for I := Low(X)+1 to High(X) do if X[I] < Result then Result := X[I];
end;

```

```
function ArrMax(const X: array of Longint): Longint;
var
  I: Integer;
begin
  Result := X[0];
  for I := Low(X)+1 to High(X) do if X[I] > Result then Result := X[I];
end;
exports
  ArrMean index 1,
  ArrMax index 2,
  ArrMin index 3;
begin
end.
```

No module definition file or *{&Linker}* statement is required for this DLL.

It is possible to write a program that uses this DLL.

```
program Test1;
function ArrMean(const X: array of Longint): Longint; external 'Arrays' index 1;
function ArrMin(const X: array of Longint): Longint; external 'Arrays' index 2;
function ArrMax(const X: array of Longint): Longint; external 'Arrays' index 3;

begin
  WriteLn('The mean value of 10, 100, 1000 is ', ArrMean([10, 100, 1000]));
  WriteLn('The min value of 10, 100, 1000 is ', ArrMin([10, 100, 1000]));
  WriteLn('The max value of 10, 100, 1000 is ', ArrMax([10, 100, 1000]));
end.
```

The second example demonstrates an easy way of creating dynamic versions of units. The library source file is the following (OS/2 only):

```
library MyRtl20;
uses SysUtils, Classes;
{&Export System, SysUtils, Classes, TypInfo}
begin
end.
```

The above library uses core Delphi-like units of the run-time library and exports the entire interface part of each unit. Before the compilation, it is necessary to set the Options|Linker|Import library radio button to *Use .DLL*. This will force the import librarian to use the DLL for creating an import library. After compilation of the library, the IDE creates an import library MYRTL20.LIB containing information about all exported symbols. Following this, it is possible to write a program that uses some functions from this DLL:

```
program Test2;
uses SysUtils;

{$IFDEF DYNAMIC_VERSION}
  {&Dynamic MYRTL20.LIB}
{$ENDIF}

begin
```

```
WriteLn('Current date & time is ', DateTimeToStr(Now));  
end.
```

The **&Dynamic** directive is used to specify that all entry points defined in the DLL corresponding to MYRTL20.LIB should be taken from the dynamic link library.

The *DYNAMIC_VERSION* conditional symbol indicates whether the units should be linked statically or dynamically. If this symbol is not defined, the program is linked statically. If it is defined, the dynamic versions of *System* and *SysUtils* units will be taken from MYRTL20.DLL. Refer to page 16 for more information about conditional compilation.

DLLs and unit initialisation code

When a DLL written in Virtual Pascal on a per-unit basis is used by a Virtual Pascal program, the initialisation code of all units located in the DLL is automatically called by the startup code of the program. However if a DLL is written using traditional technique or it is used by a program written in another language, information about unit initialisation is not available. For this reason, the initialisation code of all units must be called from the DLL's initialisation code. To make this happen, the initialisation code of the library should start with the reserved word **initialization** rather than the reserved word **begin**. This forces the compiler to generate code causing the initialisation code of all units to be called before the initialisation statement part of the DLL receives control.

In this case, the *System* unit variables *CmdLine* and *Environment* must be manually initialised for the *ParamStr* function and other *System* unit routines accessing the environment to work in the DLL.

Linker Module Definition File Reference

Virtual Pascal includes a built-in linker, and is also able to use an external linker for linking the executable files. To supply additional information to the linker, a linker module definition file or a **{&Linker}** directive can be used.

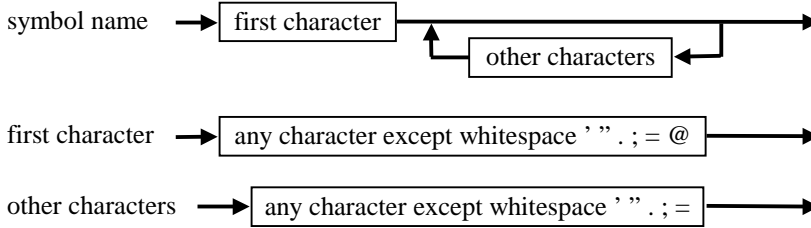
While this is rarely required, it allows information such as segment attributes, description text, etc to be specified. The module definition file should be located in the same directory as the primary file of the program or library and must have the same name, with a .DEF extension, or the statements can be included in a **{&Linker}** directive in the source code of the program.

A module definition file is a plain MS-DOS style text file that describes the names, segment attributes, exports, imports and other characteristics of an application or library. The lexical level of module definition files is quite simple. Module definition files may contain the following:

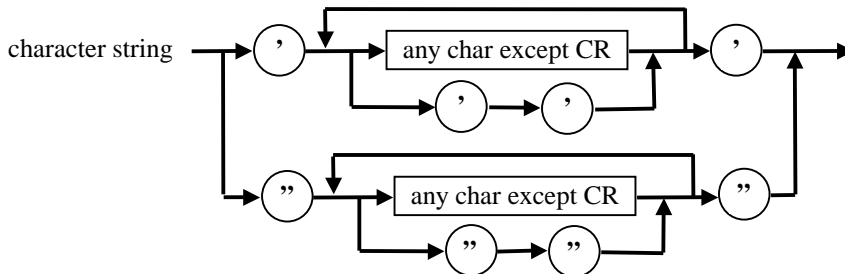
- **Whitespace.** Whitespace includes all control characters and spaces.
- **Comments.** A semicolon character (;) comments the rest of the line.
- **Reserved words.** It is important to note, that unlike Pascal reserved words, module definition file reserved words are case sensitive and must be entered in upper case. The following table lists all the module definition file reserved words. Not all of them are described here because some of them are used for 16-bit executables and become obsolete for 32-bit executables and the others are used for writing device drivers. Also note, that this list is valid for the built-in linker of Virtual Pascal. Other linkers may not accept all of them or may require a different syntax.

ALIAS	INVALID	PHYSICAL
BASE	IOPL	PRELOAD
CLASS	LIBRARY	PRIVATE
CODE	LOADONCALL	PROTECT
CONFORMING	LONGNAMES	PROTMODE
CONTIGUOUS	MAXVAL	PURE
DATA	MIXED1632	READONLY
DESCRIPTION	MOVABLE	READWRITE
DEV386	MOVEABLE	REALMODE
DEVICE	MULTIPLE	RESIDENT
DISCARDABLE	NAME	RESIDENTNAME
DOS4	NEWFILES	ROBASE
DYNAMIC	NODATA	SEGMENTS
EXECUTE-ONLY	NOEXPANDDOWN	SHARED
EXECUTEONLY	NOIOPL	SINGLE
EXECUTEREAD	NONAME	STACKSIZE
EXETYPE	NONCONFORMING	STUB
EXPANDDOWN	NONDISCARDABLE	SWAPPABLE
EXPORTS	NONE	SYSBASE
FIXED	NONPERMANENT	TERMGLOBAL
HEAPSIZE	NONSHARED	TERMINSTANCE
HUGE	NOTWINDOWCOMPAT	UNKNOWN
IMPORTS	OBJECTS	VIRTUAL
IMPURE	OLD	WINDOWAPI
INCLUDE	ORDER	WINDOWCOMPAT
INITGLOBAL	OS2	WINDOWS
INITINSTANCE	PERMANENT	

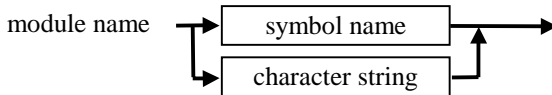
- **Symbol name.** Is used to represent the name of exported and imported variables, functions and procedures. Unlike Pascal identifiers, a wider range of characters are accepted in symbol names:



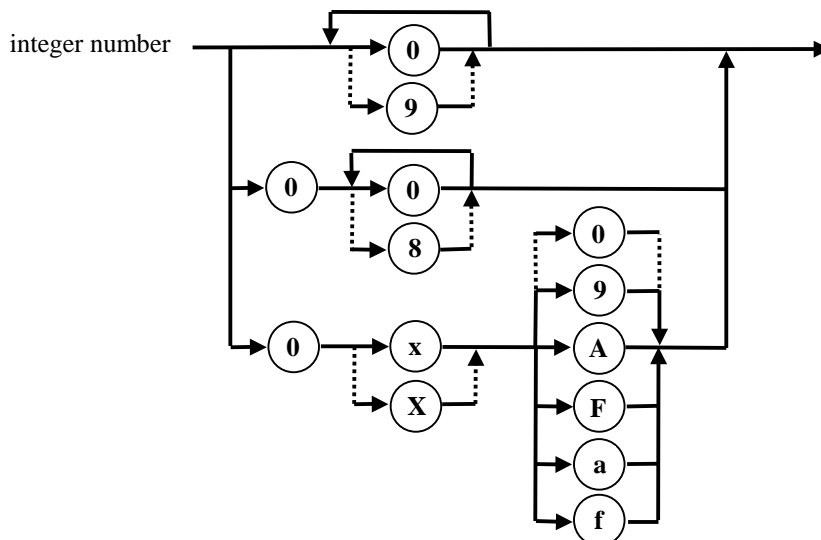
- **Character string.** Is used to specify file names, text description, etc.



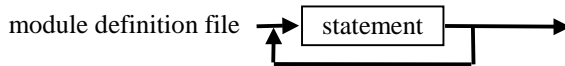
- **Module name.** Is used to specify a program or a library name. A character string must be used for the module name, containing special characters, like spaces or periods (.).



- **Integer number.** Some of the module definition statements have integer parameters. Integer numbers must be entered in C language style, not Pascal style. Decimal numbers must not contain leading insignificant zeros. Octal numbers start with a leading zero, hexadecimal numbers begin with 0x or 0X.



The syntax of the module definition file is quite simple. A module definition file contains one or more module statements. If a **NAME** or **LIBRARY** statement is present, it must precede all other statements in the module definition file.



This appendix lists most of the module definition file statements in alphabetical order with detailed descriptions. Very rarely used, obscure, and ones dealing with writing device drivers are not described here.

Segment attributes

The **CODE**, **DATA** and **SEGMENTS** statements are used to specify attributes of the segments in the executable file. One or more attributes can appear, but only one from each group should be specified. If several are specified, the last specified will take effect since they are mutually exclusive. If neither attribute from a group is specified, the default attribute is assumed.

These attributes determine when a segment is loaded:

- **PRELOAD** The segment is loaded automatically when the program starts.
- **LOADONCALL** The segment is not loaded until accessed (default).

These attributes determine whether a code segment can be read as well as executed:

- **EXECUTEONLY** The segment can only be executed.
- **EXECUTEREAD** The segment can be both executed and read (default).

I/O privilege code attributes determine whether a segment has I/O privilege, that is, whether it can access the hardware directly:

- **IOPL** The code segment has I/O privilege.
- **NOIOPL** The code segment does not have I/O privilege (default).

Note, that the Virtual Pascal compiler automatically assigns I/O privilege to segments requiring it, if the *Port* array is used by a program.

These attributes specify whether a code segment is a 286 conforming segment:

- **CONFORMING** The segment is conforming.
- **NONCONFORMING** The segment is nonconforming (default).

The concept of a conforming segment has to do with privilege level (the range of instructions that the process can execute) and is relevant only when you are writing device drivers and system level code. A conforming segment can be called from either Ring 2 or Ring 3 and the segment executes at the privilege level of the caller.

These attributes determine the access rights to a data segment:

- **READONLY** The segment can only be read.
- **READWRITE** The segment can both be read and written to (default)

These attributes determine whether all instances of the application can share a **READWRITE** data segment:

- **SHARED** One copy of the data segment is loaded and shared among all processes accessing the module (default for dynamic-link libraries). An alternative keyword with the same meaning is **PURE**.
- **NONSHARED** The segment cannot be shared and must be loaded separately for each process (default for applications). The alternative keyword is **IMPURE**. Under OS/2, this field is ignored if **READONLY** is specified, since **READONLY** data segments are always shared.

These attributes determine how an automatic data segment can be shared:

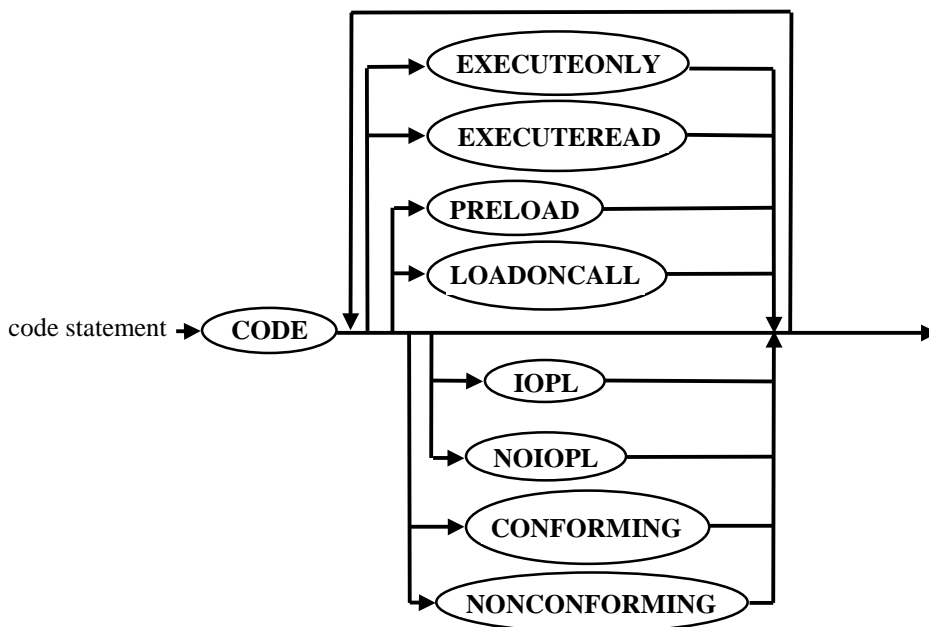
- **NONE** No automatic data segment is created.
- **SINGLE** A single automatic data segment is shared by all instances of the module. In this case, the module is said to have solo data. This reserved word is the default for dynamic-link libraries.
- **MULTIPLE** The automatic data segment is copied for each instance of the module. In this case, the module is said to have instance data. This reserved word is the default for applications.

The automatic data segment is the physical segment represented by the group name DGROUP. This segment group makes up the physical segment that contains the local stack and heap of the application.

It may occasionally be necessary to mix 16-bit code with 32-bit code. To create groups that allow such mixing, the linker requires that the segments in that group are declared as **MIXED1632**.

Segments flagged with the **ALIAS** reserved word can be addressed using the 16-bit segmented method (**far16**), or the 32-bit linear method. The loader must prepare an additional segment selector for each segment designated with the **ALIAS** reserved word. This new segment selector allows for 16-bit addressing.

CODE



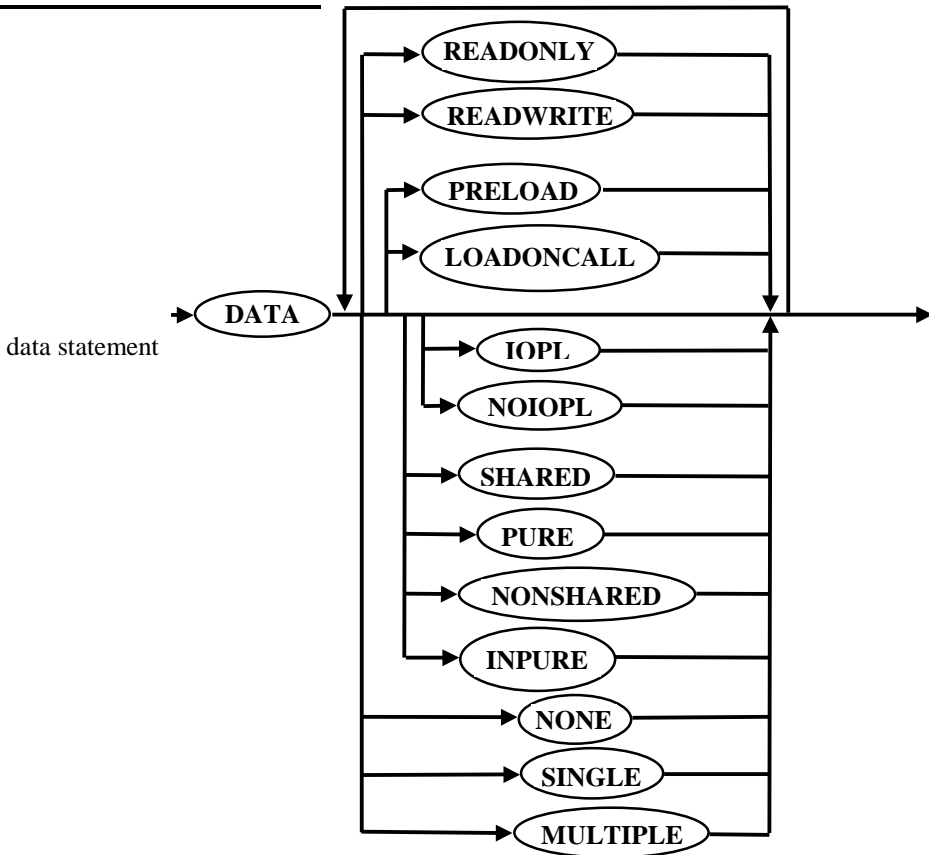
This statement defines the default attributes for code segments within the application or library. See the description of segment attributes on page 138.

Example

The following example sets the defaults for the module code segments so they have I/O hardware privilege and are not loaded until accessed.

```
CODE LOADONCALL IOPL
```

DATA



This statement defines the default attributes for data segments within the application or library. See the description of segment attributes on page 138.

Example

The following example defines the application data segment so that it is loaded only when it is accessed and cannot be shared by more than one copy of the program. By default, the data segment can be read and written, the automatic data segment is copied for each instance of the module and the data segment has no I/O privilege.

```
DATA LOADONCALL NONSHARED
```

DESCRIPTION



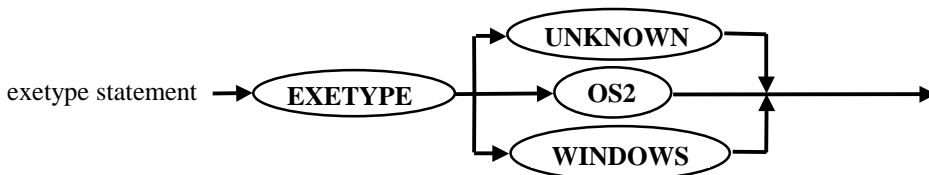
This statement inserts the specified string into the application or library. The **DESCRIPTION** statement is useful for embedding source control or copyright information in an application or library.

Example

The following example inserts the text *Example Program* into the application or library being defined.

DESCRIPTION 'Example Program'

EXETYPE



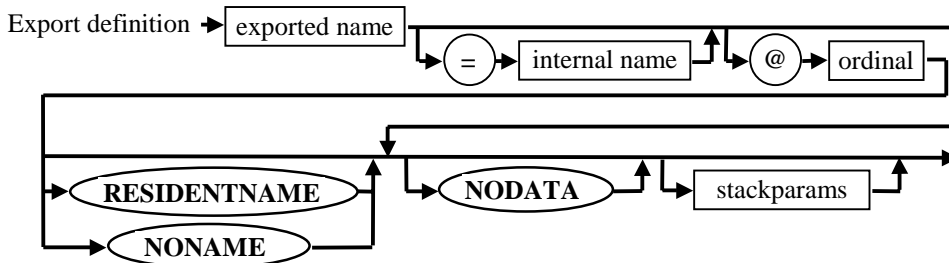
This statement specifies under which operating system the application (or dynamic-link library) is to run. This statement is optional and provides an additional degree of protection against the program being run on an incorrect operating system.

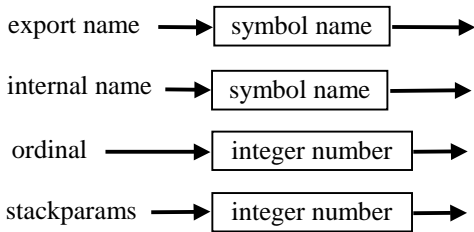
The **EXETYPE** reserved word must be followed by a descriptor of the operating system:

- **OS2** OS/2 applications and dynamic-link libraries (default)
- **WINDOWS** Windows applications
- **UNKNOWN** Other applications

The effect of **EXETYPE** is to set bits in the header that identify operating-system type. Operating-system loaders can check these bits.

EXPORTS





This statement defines the names and attributes of the symbols exported to other modules and of the functions that run with I/O privilege.

The term *export* refers to the process of making a symbol available to other run-time modules. By default, symbols are hidden from other modules at run time.

Normally, the **EXPORTS** statement is meaningful only for symbols within dynamic-link libraries and for functions that execute with I/O privilege.

The **EXPORTS** reserved word marks the beginning of the export definitions. Each definition is entered on a separate line.

- *Exported name* The symbol name as it is known to other modules.
- *Internal name* The actual name of the export symbol as it appears within the module itself; by default, this name is the same as *Exported name*.
- *Ordinal* The symbol's ordinal position within the module definition table. If this field is used, the symbol can be referenced by name or by ordinal. Use of ordinal positions is faster and may save space.
- **RESIDENTNAME** or **NONAME** determines what happens to *Exported name*. The default action with *ordinal* is to place *Exported name* in the nonresident names table. **RESIDENTNAME** places *Exported name* in the resident names table. **NONAME** discards *Exported name* from the DLL and the symbol is exported only by ordinal. These attributes are applicable only if *ordinal* is used. If *ordinal* is not used, OS/2 automatically keeps the names of all exported symbols resident in memory by default.
- **NODATA** Specifies that there is no static data in the function.
- *StackParams* The total size of the function's parameters, as measured in words (bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults *StackParams* to determine how many words to copy from the caller's stack to the I/O-privileged function's stack.

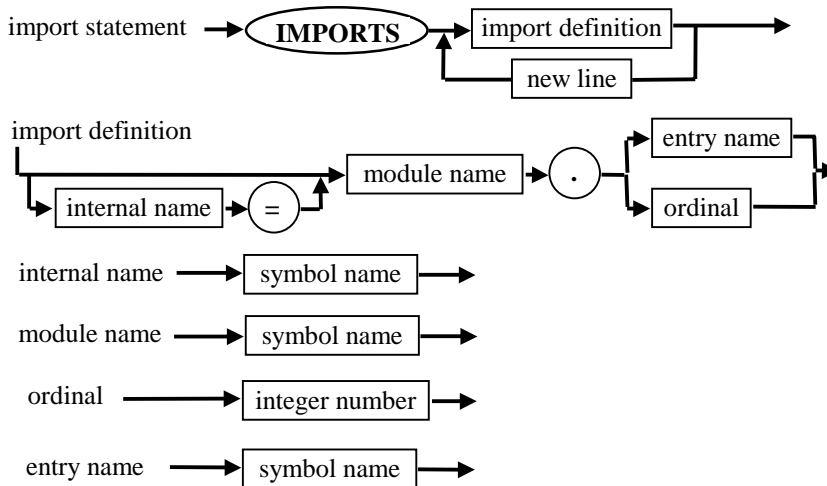
Example

EXPORTS

```
TestUnitInitProc = TestUnit@$Init @1  
InterfaceFn = TestUnit@InterfaceFn @2 RESIDENTNAME  
LowLevel 8
```

This example defines three export functions. The first two functions can be accessed either by their exported names or by an ordinal number. The last function runs with I/O privilege and therefore is given with the total size of the parameters - 8 words.

IMPORTS



This statement defines the names of the symbols imported for the application or library.

The term import refers to the process of declaring that a symbol is defined in another run-time module (a dynamic-link library).

Typically, the linker uses an import library (created by the **IMPLIB** utility) to resolve external references to dynamic-link symbols. However, the **IMPORTS** statement provides an alternative for resolving these references within a module.

The **IMPORTS** reserved word marks the beginning of the import definitions. This reserved word is followed by one or more import definitions, each on a separate line. The only limit on the number of import definitions is that the total amount of space required for their names must be less than 64K. Each import definition corresponds to a particular symbol:

- *Internal name* The name that the importing module uses to reference the symbol. Thus, *Internal name* appears in the source code of the importing module, although the symbol can have a different name in the module where it is defined. By default, *Internal name* is the same as *Entry name*.
- *Module name* The name of the application or library that contains the function.
- *Entry name* The function to be imported; can be a name or an ordinal value (ordinal values are set in an **EXPORTS** statement). If an ordinal value is given, then *Internal name* is required.

Note

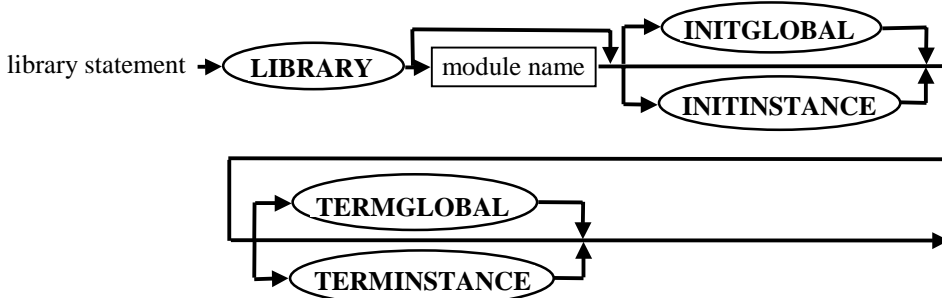
A given symbol has a name for each of three different contexts. The symbol has a name used by the exporting module (where it is defined), a name used as an entry point between modules and a name as it is used by the importing module (where it is called). If neither module uses the optional *Internal name* field, the symbol has the same name in all three contexts. If either of the modules uses the *Internal name* field, the symbol may have more than one distinct name.

Example

IMPORTS

```
Example.TestUnitInitProc
GetStatus = Example.LowLevel
InterfaceFn = Example.2
```

LIBRARY



This statement identifies the executable file as a dynamic-link library and optionally defines the name and library module initialisation required.

If *Module name* is given, it becomes the name of the library as it is known by OS/2. This name can be any valid file name. If *Module name* is not given, the name of the executable file - with the extension removed - becomes the name of the library.

- **INITGLOBAL** The library initialisation routine is called only when the library module is initially loaded into memory. This reserved word is the default flag for library initialisation. Using this reserved word without a termination flag implies **TERMGLOBAL** for DLLs with 32-bit entry points.
- **INITINSTANCE** The library initialisation routine is called each time a new process gains access to the library. Using this reserved word without a termination flag implies **TERMINSTANCE** for DLLs with 32-bit entry points.
- **TERMGLOBAL** The library termination routine is called only when the library module is unloaded from memory. This reserved word is the default flag for library termination. Using this reserved word without an initialisation flag implies **INITGLOBAL**.
- **TERMINSTANCE** The library termination routine is called each time a process relinquishes access to the library. Using this reserved word without an initialisation flag implies **INITINSTANCE**.

The termination flags can only apply to DLLs with 32-bit entry points.

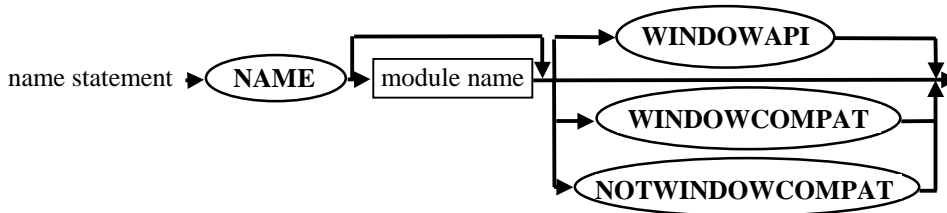
If the **LIBRARY** statement is included in a module definition file, the **NAME** statement must not appear. If no **LIBRARY** statement appears, the module definition file describes an application.

Example

The following example assigns the name *Clock* to the dynamic-link library and specifies that library initialisation be performed each time a new process gains access.

```
LIBRARY Clock INITINSTANCE
```


NAME



This statement identifies the executable file as an application and optionally defines its name and type.

If *Module name* is given, it becomes the name of the application as it is known by OS/2. This name can be any valid file name. If *Module name* is not given, the name of the executable file - with the extension removed - becomes the name of the application.

If the application type is given, it defines the type of the application:

- **WINDOWAPI** Presentation Manager application. The application uses the API provided by the Presentation Manager and must be executed in the Presentation Manager environment.
- **WINDOWCOMPAT** Application compatible with Presentation Manager. The application can run inside the Presentation Manager, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard and mouse functions supported in the Presentation Manager applications.
- **NOTWINDOWCOMPAT** Application that is not compatible with the Presentation Manager and must operate in a separate screen group from the Presentation Manager (Full Screen Session).

If the **NAME** statement appears, the **LIBRARY** statement cannot appear. If none of these statements appear, the module definition file is assumed to describe an application.

Example

The following example assigns the name *Clock* to the application being defined. It uses Presentation Manager API.

```
NAME Clock WINDOWAPI
```

OLD



This statement directs the linker to search another dynamic-link module for export ordinals. Exported names in the current module that match exported names in the **OLD** module are assigned ordinal values from that module unless one of the following conditions is in effect:

- The name in the **OLD** module has no ordinal value assigned.
- An ordinal value is explicitly assigned in the current module.

This statement is useful for preserving export ordinal values throughout successive versions of a dynamic-link module. The **OLD** statement has no effect on application modules.

Example

```
OLD 'MyOldDll.DLL'
```


Example

```
SEGMENTS
TEXT CLASS 'CODE' EXECUTEONLY CONFORMING
IO16 IOPL
'DATA' CLASS 'DATA' LOADONCALL READONLY
```

This example specifies segments named *TEXT*, *IO16* and *DATA*. The first segment is explicitly assigned class *CODE* and the second is assigned *CODE* by default. Each segment is given different attributes.

Example

```
SEGMENTS _CODE ALIAS
```

The statement above specifies that the segment *_CODE* can be called using 16-bit far calls and 32-bit near calls.

STACKSIZE



This option controls the stack size (in bytes) of the application. It is possible to specify any positive value as an *integer number*.

Example

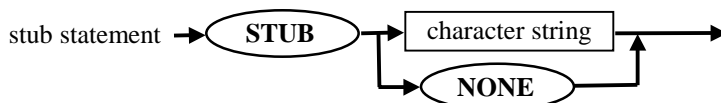
The following example sets the size of the stack segment to 16K.

```
STACKSIZE 16384
```

Note

You may not use this statement for Virtual Pascal programs, as the stack checking in the `{$$+}` state will not work properly. The size of the stack segment must be set by means of the `{$M StackSize}` compiler directive (see page 159 for details).

STUB



This statement adds a DOS executable file to the beginning of the application or library being created. The stub is invoked whenever the module is executed under DOS. Typically, the stub displays a message and terminates execution. By default, the linker adds its own standard stub for this purpose.

The *character string* specifies the DOS executable file to be added. Linker looks for it in the current directory and in the directories specified by the *PATH* environment variable.

The alternate reserved word **NONE** prevents linker from adding a default stub. **NONE** saves space in the file, but the program will hang the system if loaded in DOS, so it should be used for DLLs only.

Example

```
STUB 'STUB.EXE'
```

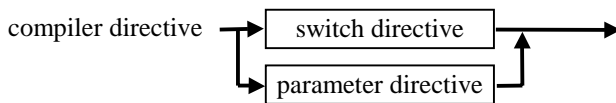
This example appends the DOS executable file STUB.EXE to the beginning of the module. STUB.EXE is executed when the module is run under DOS.

A P P E N D I X B

Compiler directives

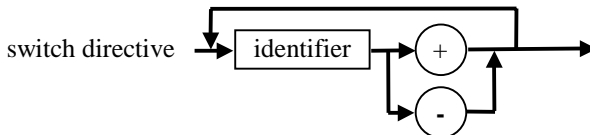
Compiler directives are used to provide information to and set syntax, code generation and other features of the compiler. Compiler directives are implemented as comments that starts with dollar sign (\$) followed by a directive name and optional parameters. Alternatively, an ampersand symbol (&) can be used instead of \$. This is intended to be used with compiler directives that are specific to Virtual Pascal, since such compiler directives are then ignored by other Pascal compilers. All Borland Pascal and 16-bit Delphi compiler directives are recognised by Virtual Pascal.

There are two types of compiler directives: global and local. Global directives must appear before the declaration part of the program, unit or library. Global directives affect the entire compilation process. Local directives can appear anywhere, they affect only the part of the source file after it.

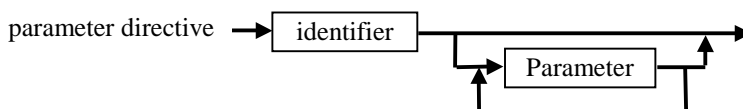


Compiler directives are further subdivided into two groups:

- Switch directives that act as toggles enabling or disabling certain compiler features. A plus (+) or minus (-) symbol follows the directive name and indicates the enabled or disabled state of the directive.



- Parameter directives are used to supply additional information to the compiler. An important subgroup of parameter directives are conditional directives that select the part of the source file that is to be compiled. For a description of conditional compilation refer to page 16.



The name of the directive must be followed by at least one whitespace and optionally one or more parameters.

Virtual Pascal specific directives are listed with a & sign while directives that are common for Borland and Virtual Pascal compilers are listed with a \$ sign.

\$A, &AlignData

Switch: Default=&AlignData+ Type=Local

Switches between byte- and double word-alignment of variables and typed constants. Data alignment speeds up execution. **\$A** and **&AlignData** are equivalent.

In the **&AlignData+** state, variables and typed constants are aligned according to the following table:

Size of the variable	Action
1	No alignment
2	Word (2-byte boundary) alignment
>=3	Double word (4-byte boundary) alignment

In the **&AlignData-** state, no alignment is performed. Variables and typed constants are simply placed at the next available address, regardless of their size.

&AlignCode

Switch: Default=&AlignCode+ Type=Local

Specifies whether to align entry points of procedures and functions at a double word boundary. Code alignment speeds up execution, particularly on newer processors, but makes the executables larger.

In the **&AlignCode+** state, all functions and procedures entry points are aligned at a double word (4-byte) boundary (i.e. the address is divisible by 4).

In the **&AlignCode-** state, no alignment is performed. Code for functions and procedures is simply placed at the next available address.

&AlignRec

Switch: Default=&AlignRec- Type=Local

Switches between byte- and double word-alignment of record and object fields. Field alignment speeds up execution.

In the **&AlignRec+** state, fields are aligned according to the following table:

Field size	Action
1	No alignment
2	Word (2-byte boundary) alignment
>=3	Double word (4-byte boundary) alignment

Note: this may affect the size of records containing byte and word sized variables.

In the **&AlignRec-** state, no alignment is performed. The record and object fields are simply placed at the next available address, regardless of their size.

&Alters

Parameter: Default=&Alters eax,ecx,edx Type=Local

See description of the **&Saves** directives on page 162.

&Asm

Switch: Default=&Asm- Type=Global

Specifies whether to produce assembly source for the unit.

In the **&Asm+** state, the compiler produces assembly source code in addition to an object file for the unit. Note, that the compiler is able to generate assembler source only if smart linking is disabled (**&SmartLink-** state), because the compiler generates an object module for each **var** and **const** section, procedure and function, etc., when smart linking is enabled.

In the **&Asm-** state, only an object file is produced.

See page 23 for more information about the compilation process.

The assembler source code produced is readable and can be compiled using a macro assembler like MASM or TASM.

\$B

Switch: Default=\$B- Type=Local

Switches between the two different models of code generation for **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 with **and** and **or** operators is guaranteed to be evaluated, even when the result of the entire expression is evident.

In the **\$B-** state, the compiler generates code for short-circuit boolean expression evaluation. This means that evaluation stops as soon as the result of the entire expression becomes known. Short-circuit boolean expression evaluation is based on the following facts:

- When at least one operand of an **or** operator is *True*, the result of the operator is *True*.
- When at least one operand of an **and** operator is *False*, then result of the operator is *False*.

If expressions do not have any side effects, this directive can be switched on to increase code speed. Side effects can occur, if the expression includes a function call and the function modifies the state of the program.

&Cdecl

Switch: Default=&Cdecl- Type=Local

These compiler directives set up the default language calling convention.

Directive	Calling convention
&Cdecl	C language
&Far16	16-bit Pascal
&StdCall	Standard call
Disabled	Pascal language

Specifying any directive in the disabled state (-) enables the default Pascal calling convention.

Using these directives is the only way to declare a procedural type with the calling convention other than Pascal.

For a particular procedure or function you can change the default calling convention by specifying one of the **cdecl**, **far16**, **stdcall** or **pascal** standard procedural directives. See also the section on Calling conventions.

&Comments

Switch: Default=&Comments- Type=Global

Enables nested { } comments.

In the **&Comments+** state, comments consisting of “curly brackets”, i.e. { and } can be nested up to 32 deep. In the default state, which is compatible with Borland Pascal and Delphi, comments cannot be nested.

Example:

The following code

```
{
type
```

```
{ tMyType is used to hold data }
tMyType = record
  fTest: Integer;
  fDummy: Integer;
end;
}
```

will compile in the **{&Comments+}** state, but will generate a syntax error in the default **{&Comments-}** state.

\$D

Switch: Default=\$D+ Type=Global

Specifies whether to produce debug information, consisting of a line number table for each procedure. The table defines the correspondence between object code addresses and source text line numbers.

In the **\$D+** state, debug information is generated, which allows the integrated debugger to trace the code of the unit and set breakpoints. When a run-time error occurs, the integrated debugger can show the statement that caused the error. For units, the debug information is recorded in a .VPI file. Debug information increases the size of .VPI files and takes up additional room when you compile programs that use the unit, but it only slightly affects the size and does not affect the speed of the executable program.

The state of **\$D** also controls whether local symbol information is generated. Local symbol information consists of the symbols in the module's implementation part (names and types of all local variables and constants in a module) and the symbols within the module's procedures and functions. This was controlled by the **\$L** directive in Borland Pascal.

In the **\$D+** state, local symbols are stored in the .VPI file for a given program or unit and the integrated debugger can be used to examine and modify the module's local variables. Calls to the module's procedures and functions can be examined via the Call Stack window.

In the **\$D-** state, no debug or symbol information is generated.

This switch is usually used with the **{&LocInfo}** switch.

\$DEFINE

Parameter: Type=Local

Defines a conditional symbol with the given *Name*.

Syntax:

```
$DEFINE Name
```

The defined symbol is recognised for the remainder of the compilation (current unit or program source file only), or until it appears in an **\$UNDEF** *Name* directive. The name specified can be examined by use of the **\$IFDEF** conditional compilation directive.

Note: **{\$DEFINE** *Name* **}** has no effect if *Name* is already defined.

See also the section on Conditional compilation.

&Delphi

Switch: Default=&Delphi+ Type=Local

Enables the Delphi Object Pascal language extensions.

In the **&Delphi+** state Delphi syntax is enabled.

The Delphi language extensions supported by Virtual Pascal can cause code written for Borland Pascal to fail compilation. In the **&Delphi-** state, the conflicting parts of the Delphi language extensions are disabled and you cannot use:

- New reserved words: **try**, **finally**, **except**, **finalization**, **class**, etc.
- Function result references via the *Result* variable.

See page 93 for an example of using *Result* and page 100 for information about exception handling.

&Dynamic

Parameter: Type=Global

Indicates that a program or library uses a unit version that is located in a dynamic link library (OS/2 target only).

Syntax:

&Dynamic *UnitName* [,*UnitName*]

or

&Dynamic *Import library name*

When compiling a program or a library, this directive can be used to specify which of the units used by the program or library are located in DLL(s). For these units the object file names are not specified in the linker response file. Instead, all definitions of interface symbols of these units should be located in an import library. The name of this import library should be specified using a **\$L** *FileName* directive.

Unit names can be separated by commas, spaces or semicolons. This directive must be placed after the program or library **uses** clause. It is illegal to use it in a unit.

Alternatively, the directive can specify the name of the import library. In this case, all external references that can be resolved using this library are dynamically linked, and no **\$L** directive is required.

See also the **{&Export *UnitName*}** directive on page 154.

\$ELSE

Parameter(conditional compilation): Type=Local

Compiles or ignores the source text that follows it.

Syntax:

\$ELSE

The **\$ELSE** directive marks an optional clause of a conditional-compilation block defined by a **\$IFDEF**, **\$IFNDEF** or **\$IFOPT** directives.

\$ELSE compiles the source code that follows it if the preceding **\$IFxxx** condition is *not* met.

If the **\$IFxxx** condition is met, **\$ELSE** ignores the source code that follows it.

\$ENDIF

Parameter(conditional compilation): Type=Local

Marks the end of a conditional-compilation block defined by **\$IFDEF**, **\$IFNDEF** or **\$IFOPT** directive.

Syntax:

\$ENDIF

One **\$ENDIF** is required for each **IFxxx** directive.

&Export

Parameter: Type=Local

Indicates that all interface symbols of the specified unit(s) should be exported.

Syntax:

&Export *UnitName* [,*UnitName*]

This directive is used in a library. It specifies the units, dynamic version of which can be used by other programs or libraries. Unit names can be separated by commas, spaces or semicolons.

This directive must be placed after the program's or library's uses clause. It is illegal to use it in a unit.

Alternative Syntax:

&Export:[*ByName* / *ByOrdinal* / *NoName*]

When used in this fashion, the directive specifies how entries should be exported: By name or by ordinal numbers. The three parameters have the following meaning:

ByName	forces the linker to export all interface symbols of units specified in {&Export UnitName} statements as well as all procedures exported by means of the export standard directive to be exported by name.
ByOrdinal	causes symbols to be exported by ordinal numbers. The names of the exported symbols are still included in the DLL and can be imported by name by creating another import library using the IMPLIB utility.
NoName	the same as ByOrdinal , except that names of exported symbols are not included in the DLL. Exported symbols can be imported by ordinal numbers only, using the import library generated by the linker. This causes the smallest DLLs to be created and is the default setting.

See also the **{&Dynamic UnitName}** directive on page 152.

&Far16

Switch: Default=&Far16- Type=Local

This compiler directive sets up the default language calling convention. Refer to **&Cdecl** on page 151 for more information.

&Frame

Switch: Default=&Frame+ Type=Local

Specifies whether to set up a stack frame for assembler procedures and functions. This directive should be placed before the **asm** reserved word that starts the statement part.

In the **&Frame+** state, a stack frame is generated, allowing the use of the integrated debugger to view the value of any local variable or argument (when both **\$D** and **\$L** are enabled). Local variables and parameters are accessed via the EBP stack frame register.

In the **&Frame-** state, a stack frame is not generated and the values of arguments and local variables of the procedure cannot be viewed in the integrated debugger. The current stack of calling functions and procedures is not accessible. Both local variables and parameters are accessed via the ESP stack pointer register, instead of EBP. This generates the fastest code, but may produce bugs that are hard to find. Accessing local variables or parameters is possible only if the code does not modify the stack using push/pop statements, since this changes the value of the ESP register. **&Frame-** should normally be used by experienced users when writing small assembler routines only.

The **&Frame** directive is usually used with a **{&Uses RegList}** directive (see page 164).

&G3, &G4, &G5

Switch: Default=\$G3+ Type=Local

Enables generation of code for the corresponding processor:

- **&G3** - 386
- **&G4** - 486
- **&G5** - Pentium.

These compiler directives are mutually exclusive, so if two or more directives are specified as enabled, the last one specified is used.

These directives are relevant to the built-in assembler and determine which instruction set is allowed, and also determine the primary target processor of the executable.

Virtual Pascal generates only 386 instructions and all programs compiled with Virtual Pascal will run on any of the processors. However, it uses this switch to determine the processor on which the program should run best and can be used to optimise the code for a Pentium at the expense of 386 machines by specifying **{&G5+}**, for example.

See also the **{&Optimise}** directive.

\$H

Switch: Default=\$H- Type=Local

Controls the meaning of the reserved word **string** and provides compatibility with long strings introduced in 32-bit Delphi.

In the enabled **\$H+** state, strings declared using the the **string** reserved word are long strings (*AnsiString*).

In the disabled **\$H-** state, strings declared using the **string** reserved word are normal Pascal strings (*ShortString*).

Note, that strings declared with a maximum length are always short strings, irrespective of the setting of the **\$H** directive.

\$I

Switch: Default=\$I+ Type=Local

Enables or disables the generation of code, checking the result of a call to an I/O procedure.

In the **\$I+** state, if an I/O procedure returns a non-zero I/O result, the program terminates displaying a run-time error message.

In the **\$I-** state, the standard *IOResult* function must be used to check for I/O errors.

\$I

Parameter: Type=Local

Instructs the compiler to include the named file in the compilation. If the extension is omitted, a default of .PAS is assumed.

Syntax:

\$I *FileName*

If *FileName* does not specify a directory, the compiler searches for the file

- first in the current directory
- then in the directories specified in the Options|Directories|Include directories input box (or in the directories you specified via the /I option on the command line of VPC).

The included file is inserted in the compiled text right after the **{\$I *FileName*}** directive.

HPFS, NTFS and VFAT long file names containing special symbols such as blanks are also supported. In this case the name should be enclosed in double quotes, like this:

{\$I "Very long HPFS file name.INC"}

Note: An include file cannot be specified in the middle of a statement part. All statements between the begin and end of a statement part must reside in the same source file.

\$IFDEF

Parameter(conditional compilation): Type=Local

Compiles the source text that follows it if *Name* is defined.

Syntax:

\$IFDEF *Name*

The **\$IFDEF** directive controls conditional compilation of the source file by checking the specified *Name*. If the *Name* has been defined by using a **\$DEFINE** directive (or by using the /D command-line option or *Name* is listed in the Compiler|Options|Conditional defines input box), **\$IFDEF** directs the compiler to continue with the source text immediately after the **\$IFDEF**. If the *Name* has not been defined, **\$IFDEF** directs the compiler to skip all source text up to the next **\$ELSE** or **\$ENDIF** directive.

\$IFNDEF

Parameter(conditional compilation): Type=Local

Compiles the source text that follows it if *Name* is *not* defined.

Syntax:

\$IFNDEF *Name*

The **\$IFNDEF** directive controls conditional compilation of the source file by checking the specified *Name*. If the *name* has not been defined or if its definition has been removed by using an **\$UNDEF** directive, **\$IFNDEF** directs the compiler to continue processing source up to the next **\$ELSE** or **\$ENDIF** directive. If the *Name* is defined, **\$IFNDEF** directs the compiler to skip to the next **\$ELSE** or **\$ENDIF** directive.

\$IFOPT

Parameter(conditional compilation): Type=Local

Compiles the source text that follows it if the switch compiler directive is currently in the specified state.

Syntax:

\$IFOPT *SwitchDir*

SwitchDir is the name of a switch compiler directive, followed by + or -, for example **\$IFOPT R+**.

This can be used to change and restore the setting of a directive:

// Save original setting of the **\$R** directive

```
{&IFOPT R+} {$DEFINE RCHCK} {$ELSE} {$UNDEF RCHCK} {$ENDIF}
```

```
{$R+} // Enable range checking for a bit of code
```

... code

// Restore original setting

```
{&IFDEF RCHCK} {$R+} {$ELSE} {$R-} {$ENDIF}
```

\$J

Switch: Default=\$J- Type=Local

Controls the write access to typed constants.

In the **\$J-** state, typed constants are treated as true constants that cannot be modified, passed as **var** parameters, etc.

In the default **\$J+** state, typed constants are treated as pre-initialised variables that can be modified. This setting is compatible with Borland Pascal.

\$L

Obsolete and ignored.

See the **\$D** directive on page 151.

\$L

Parameters: Type=Local

Instructs the compiler to link the named file with the program or dynamic link library being compiled.

Syntax:

\$L *FileName*

The **\$L** directive is used to use code written in other languages or to specify the name of an import library containing a list of entries defined in a DLL.

The named file must be either an OMF object file (.OBJ) or library (.LIB). All file names specified in the **\$L** directives are passed to the linker:

- Object files in the object file list
- Libraries in the library list. If the extension is omitted, a default of .LIB will be assumed and this library name is placed in the object file list.

If *FileName* does not contain a directory, the linker will search for it

- first in the current directory
- then in the directories specified in the Options|Directories|Library directories input box (or in the directories specified in the LIB environment variable, when using the command line compiler).

Long file names containing special symbols, such as blanks, are also supported. In this case the name should be enclosed in double quotes, like this:

```
{$L "Quite long file name.obj"}
```

&Linker

Parameters: Type=Local

Specifies module definition statements directly in the source code, without using an external .DEF file.

Syntax:

&Linker *Module definition statements*

Example:

```
{&Linker  
DESCRIPTION 'Test Program'  
STUB 'C:\DOS\DosStub.Exe'  
}
```

&LocInfo

Switch: Default &LocInfo- Type=Local

Enables or disables the generation of run-time location information (RTLTI).

When a unit is compiled in the **&LocInfo+** state, Virtual Pascal includes Location Information in the resulting .VPI file, consisting of line numbers and source code file names.

When the Generate Location Information switch on the Options-Linker page is enabled, the Location Information is included in the executable file (Only when using the internal linker). This information can be used at run-time to determine the exact location of an error, and can be an invaluable help when searching for a problem occurring only at a customer's site.

When Location Information is enabled, the default error handlers of the *System* and the *SysUtils* units output source file and line number information if the program terminates abnormally. This is implemented by calls to the *GetLocationInfo* function defined in the *System* unit, which returns the source file name and line number of a given code address for which Location Information is available.

\$M

Switch: Default \$M- Type=Local

Enables or disables the generation of run-time type information.

When a class is declared in the **\$M+** state, or derived from a class, declared in the **\$M+** state, the compiler generates run-time type information for fields, methods and properties that are declared in **published** sections.

If a class is declared in the **\$M-** state and is not derived from a class that was declared in the **\$M+** state, published sections are not allowed in the class.

\$M*Parameter: Default=\$M 16384 Type=Global*

Specifies the stack size of a program.

Syntax:

\$M *StackSize*<,*MaxStackSize*>

StackSize must be an integer number greater than 8192, specifying the size of the stack segment in bytes.

MaxStackSize optionally specifies the maximum allowed size of the stack in bytes.

Note: the **\$M** directive has no effect when used in a unit. The **\$M** directive is ignored in a library (a DLL always uses the stack of the application that calls it).

&Open32

Enables Windows compatibility

Switch: Default=&Open32- Type=Local

In the **&Open32+** mode, resource files (.RES) are treated as Windows (Win16 or Win32) resource files, even when linking for an OS/2 target. The built-in resource linker automatically converts these to OS/2 format prior to linking them. Additionally, the **stdcall** standard directive is equivalent to **cdecl** in this mode. This way, Windows header files specifying **stdcall** can be used without modification when using Open32 for OS/2, which uses the **cdecl** calling convention.

In the default **&Open32-** mode, no resource file conversion takes place and the **stdcall** standard directive denotes the **stdcall** calling convention.

Example:

```
{&Win32+}
  {$R Win16.res}    // Include Win16 resource in OS/2 executable
  {$R Win32.res}    // Include Win32 resource in OS/2 executable
{&Win32-}
  {$R OS2.res}      // Include native OS/2 resource in OS/2 executable
```

Using the Open32 library from IBM (which emulates a subset of the Win32 API in native OS/2), Delphi programs can be recompiled to an OS/2 target without changing source code or resource files, when using this directive.

&Optimise, &Optimize*Switch: Default=\$Optimise+ Type=Local*

Enables a set of optimisations, such as allocating the most frequently used local variables in CPU registers and “peephole” optimisations.

Enabling the **&Optimise+** (or **&Optimize+**) directive speeds up execution and makes the object code smaller. For debugging purposes, optimisations may be switched off.

When a local variable is kept in a register, the integrated debugger shows the contents of this CPU register instead of the memory location when evaluating the variable. However, the register may not always hold the local variable value, because it may be temporarily stored in memory while the register is used for another variable. This can cause unexpected problems when modifying the values of local variables. When modifying a local variable that is normally kept in a register, but has been temporarily stored in memory, modifying it will modify the register in which the debugger thinks it is stored - even if it is not. This limitation may be removed in a future version of the debugger; the temporary work-around is to disable optimisations when debugging complex code sequences.

In the **&Optimise-(&Optimize-)** state, local variables are kept in memory and the debugger always references the correct value.

See also the **{\$W}** and **{&Speed}** directives.

&OrgName

Switch: Default=&OrgName- Type=local

Specifies what names are associated with public symbols in object files.

In the **&OrgName+** state, public symbols are given their original names. This can be useful if a Pascal unit is linked into a program written in another language or uses a module written in another language. In this case, it is the responsibility of the programmer to ensure that the names of public symbols are unique; if not, a linker error will occur.

In the default **&OrgName-** state, the unit name and an "at" symbol (@) is added to the start of all public symbol names. This way, it is possible to use public symbols with identical names, as long as they are located in the different modules (units).

Note: the term *public* here refers to symbols declared in the interface part of the unit or global variables, typed constant, procedures and functions of the program or library.

See also the appendix on Naming conventions on page 187.

\$P

Switch: Default=\$P- Type=Global

Controls the meaning of variable parameters declared using the **string** reserved word.

In the **\$P+** state, variable parameters declared using the string reserved word are open string parameters. The actual parameter of an open string parameter can be a variable of any string type and within a procedure or function, the size attribute (maximum length) of the formal parameter will be the same as that of the actual parameter.

In the **\$P-** state, variable parameters declared using the string reserved word are normal variable parameters. Regardless of the setting of the **\$P** directive, the *OpenString* identifier can always be used to declare open string parameters.

See also the section on Open string parameters on page 123.

&PmType

Parameter: Type=Global

Overrides the default application type that is set in Options|Linker dialogue box.

Syntax:

&PmType *AppType*

Default:

uses the default application type taken from the Linker dialogue box.

AppType can be either **NOVIO**, **VIO** or **PM**.

Note: It is illegal to use this directive in a unit.

Parameter	Meaning
NOVIO	Not aware of Presentation Manager API (OS/2), runs in a separate full screen session
VIO	Compatible with Presentation Manager API, can run inside a PM window or in a separate full screen session
PM	Uses Presentation Manager API, must be executed inside PM

&PureInt

Switch: Default=&PureInt- Type=Global

Forces a unit to be a pure interface unit.

To supply a unit's object code without source code and be sure that it works even if the format of the .VPI files changes, a pure interface unit for should be created. To do this, it must contain **{&PureInt+}** directive at the start of the unit and the entire interface part of the unit. The implementation part of the unit should be removed, except for the **uses** clause (if applicable). If the unit has an initialisation part, the initial **begin** should be included as well:

implementation

uses

Dos,Crt;

begin

end.

If no initialisation part is present, the **begin** should be left out:

implementation

uses

Dos,Crt;

end.

The **{&PureInt+}** directive forces the compiler to ignore undefined forward references to procedures, functions and methods and assume that they are implemented in the original version of the unit, in the form of a .OBJ or .LIB file.

No object file or library is generated for a pure interface unit, meaning that a newer version of the compiler can be used to create an appropriate .VPI file for it and still use the object file or library produced from the original unit by the old version of the compiler.

In the default **{&PureInt-}** state, a unit is assumed to implement the interface part as usual.

Note: Those parts of the Virtual Pascal Run-Time Library that are derived from source code copyrighted by Borland Int are supplied as Pure Interface units; the full source code is supplied as upgrade patches that requires the original source code to be available.

\$Q

Switch: Default=\$Q- Type=Local

Controls the generation of overflow-checking code.

In the **\$Q+** state, certain integer operations are checked for overflow, such as: +, -, * *Abs, Sqr, Succ* and *Pred*. The code for each of these arithmetic operations is followed by additional code that verifies that the result is within the supported range. If an overflow check fails, the program terminates and displays a run-time error message.

In the **\$Q-** state, no overflow checking is performed. The **\$Q** switch is usually used in conjunction with the **\$R** switch. Enabling overflow checking slows down your program and makes it larger, so use **{&Q+}** only for debugging.

Note: for compatibility with Borland Pascal, **\$Q** does not affect the *Inc* and *Dec* standard procedures.

\$R

Switch: Default=\$R- Type=Local

Enables and disables the generation of range-checking code.

In the **\$R+** state, all array and string indexing expressions are verified to be within the allowed bounds. All assignments to scalar variables are checked to be within range. If a range check fails, the program terminates and displays a run-time error message.

In the **\$R-** state, no range-checking is done.

Enabling range-checking slows down the program and makes it larger, so use it only for debugging.

Note: for compatibility with Borland Pascal **\$R** does not affect the *Inc* and *Dec* standard procedures.

\$S

Switch: Default=\$S+ Type=Local

In the **\$S+** state, the compiler generates code at the beginning of each procedure or function, checking whether sufficient stack space for local variables and other temporary storage is available. If there is not enough stack space, the program terminates and displays a run-time error message.

In the **\$S-** state, when there is not enough stack space, a call to a procedure or function is likely to cause a stack fault, but the exact location of the error may not be identified and such errors can be very difficult to find.

Enabling stack checking slows down the program and makes it larger and as such should primarily be used for debugging. However, be sure to verify that the program does not contain code that risks overflowing the stack before disabling this directive.

&Saves

Parameter: Default=&Saves ebx,esi,edi Type=Local

Specifies which CPU registers that are saved (altered for the **&Alters** directive) by a procedure, function or **asm** statement.

Syntax:

&Saves RegisterList

&Alters RegisterList

RegisterList consists of one or more 32-bit general purpose register names (*eax, ebx, ecx, edx, esi* or *edi*) separated by commas, spaces or semicolons, **NONE** or **ALL**.

&Alters is complementary to **&Saves**. For example, if a procedure alters only the *eax* register, it saves *ebx, ecx, edx, esi* and *edi*, so **{&Alters eax}** and **{&Saves ebx,ecx,edx,esi,edi}** are equivalent.

The **&Saves (&Alters)** directive tells the compiler that the CPU registers specified in the *RegisterList* are *not* changed (changed for the **&Alters** directive) by the procedure or function. The compiler may use these registers for register variables, for storing temporary results across function or procedure calls, etc. If the code of the procedure or function uses these registers, they are saved on entry and restored on exit. The default setting is compatible with the SYSCALL calling convention, where all registers except *ebx, esi* and *edi* are expected to be altered.

Information about saved registers is included into the procedure or function type. For this reason, only procedures (functions) with equivalent argument lists (and return values) saving the same registers are

compatible. **&Saves (&Alters)** affects only the defining (first) declaration of the procedure or function.

The **&Saves (&Alters)** directive can be also used to specify registers that are preserved (changed) by **asm** statements. In this case, it should be specified inside the statement part before or just after the **asm** reserved word. It affects only **asm** statements located in the current statement part below and does not change the setting for procedures and functions.

Note: Functions that return an ordinal or a pointer type, constructor, destructor and dynamic method always change the value of the *eax* register regardless of the setting of this directive. **far16** procedures and functions always save *ebx*, *esi* and *edi* registers.

Warning: do not change the default setting for procedures and functions unless you are an expert assembler programmer.

&SmartLink

Switch: Default=&SmartLink- Type=Local

Specifies the type of the output file to be generated for a unit, .OBJ or .LIB.

In the **&SmartLink+** state, the smart linking facility is enabled. For each unit that has object code, the compiler generates an OMF library file (.LIB).

In the **&SmartLink-** state, the smart linking feature is disabled. For each unit that has object code, the compiler generates an OMF object file (.OBJ).

Enabling **&SmartLink** directive dramatically reduces the size of the produced executable file. When debugging DLLs, smart linking should be disabled, because the integrated debugger is unable to symbolically debug DLLs linked from smart linked units.

The **&SmartLink** directive is ignored if used in a program or library.

Note: Assembly output can be enabled (**{&Asm+}** state) in the **{&SmartLink-}** state only.

&Speed

Switch: Default=\$Speed- Type=Local

Selects the compiler's optimisation strategy.

In the **&Speed+** state, the compiler optimises the generated code for speed, choosing the fastest code sequence for a given task.

In the **&Speed-** state, the compiler optimises the generated code for size, choosing the smallest code sequence possible.

See also the **{&Optimise}** and **{\$W}** directives.

\$StdCall

Switch: Default=&StdCall- Type=Local

This compiler directive sets up the default language calling convention. See the description of **&Cdecl** directive on page 151 for more information.

\$T

Switch: Default=\$T- Type=Local

Controls the type of pointer values generated by the @ operator.

In the **\$T-** state, the result type of the @ operator is always an untyped pointer that is compatible with all pointer types.

In the **\$T+** state, when the @ operator is applied to a variable of type *T*, the type of the result is [^]*T*, which is compatible only with other pointers to this type of variable.

\$UNDEF

Parameter(conditional compilation): Type=Local

Undefines a previously defined conditional symbol of *Name*.

Syntax:

\$UNDEF *Name*

This directive removes the definition of the specified *Name*. It has the same effect for the rest of the compilation as if the *Name* was never defined.

Note: **\$UNDEF** *Name* directive has no effect if *Name* is not already defined.

&Use32

Parameter: Default=&Use32- Type=Local

Provides a shortcut for including the *Use32* unit in the uses clause. Units compiled in the **&Use32+** state implicitly include the *Use32* unit in the uses clause, after the *System* unit and before any units explicitly included.

This directive is primarily included for compatibility with Borland Pascal, since most programs written for 16-bit DOS will compile with Virtual Pascal when the *Use32* unit is included in the uses clause.

The *Use32* unit redefines the basic *Integer* and *Word* types to be 32-bit types identical to the *LongInt* type.

See also Discussion: Use32, Use16 and bits on page 30.

&Uses

Switch: Default=&Uses NONE Type=Local

Indicates which registers should be automatically pushed at the beginning of an assembler procedure or function and restored (popped) on exit.

Syntax:

&Uses *RegisterList*

RegisterList is one or several CPU register names (*eax, ebx, ecx, edx, esi, edi, ebp, ds, es, fs, gs, efl*) separated by commas, spaces or semicolons; **NONE** or **ALL**.

Using this directive can make assembler code clearer and easier to read.

Example:

Directive	Entry code	Exit code
{&Uses ebx,esi}	PUSH EBX	POP ESI
	PUSH ESI	POP EBX
{&Uses ALL}	PUSHAD	POPAD
{&Uses EFL}	PUSHFD	POPFD

\$V

Switch: Default=\$V+ Type=Local

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-** 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. You can not reliably check indexing of the actual parameter in the **\$R+** state, which is why the use of **\$V-** is not recommended. Use open string parameters instead (see the **\$P** switch directive on page 160).

\$W

Switch: Default=\$W+ Type=Local

Enables the generation of stack frames.

In the **\$W+** state a stack frame is generated for every procedure and function. This is useful for debugging, since only procedures and functions with a stack frame can be displayed in the call stack window.

In the **\$W-** state, the compiler does not generate a stack frame for procedures and functions that do not have parameters and local variables. This can decrease code size and increase performance.

See also the **{&Optimise}** and **{&Speed}** directives.

\$X

Switch: Default=\$X+ Type=Global

Enables or disables Virtual Pascal's extended syntax.

In the **\$X+** state, function calls can be used as statements, i.e. the result of a function call can be discarded (does not apply to built-in functions). Null-terminated strings are enabled by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays.

In the **\$X-** state, extended syntax is disabled.

\$Z

Switch: Default=\$Z- Type=Global

Controls the storage size of enumerated types.

In the **\$Z+** state, an enumerated type state is always stored as a double word (4 bytes).

In the **\$Z-** state, an enumerated type is stored as byte if type has no more than 256 values and as double word otherwise.

This directive is useful for interfacing with libraries written in other languages, like C or C++, where enumerated types are represented as double words.

&Zd

Switch: Default=&Zd- Type=Global

Controls the generation of line number information in the object file. It has the same meaning as the */Zd* command line switch that most command line compilers have.

In the **&Zd+** state, line number information is generated. This can be useful if you want to link your Pascal code into a program that is written in the other language. In this case you can trace Pascal source code in a standalone debugger, such as IPMD.

In the **&Zd-** state, line number information is not generated.

Note: setting of this directive *does not* affect the integrated debugger. It uses debug information recorded in the *.VPI* file of units compiled in the **\$D+** state.

The Open Debug API (ODAPI)

Virtual Pascal v2 by default includes support for native debugging of Win32 and OS/2 programs. This default functionality can be modified to suit your specific needs, or can be extended, through the VP ODAPI specification.

By using the functionality provided by the Virtual Pascal Open Debug API (ODAPI), it is possible to include support for debugging programs running on other platforms, across network connections or even direct cable connections.

In order to implement debugger support for more platforms or for changing the behaviour of the existing debugger function, the set of functions comprising ODAPI must be implemented as a DLL, to be used by the Virtual Pascal IDE when debugging applications. The interface to be supported is defined in the Pascal source file *VpDbgApi.Pas*, located in the *Source\Rtl* directory. The following is a discussion and description of the functions and what they are required to do.

The debugger DLL

A debugger DLL must export a single entrypoint, *SysDbgGetIDEInterface*, which is called by the IDE during startup. This call defines an additional set of entrypoints for the debugger to use during the debugging process; these are described in the section on The ODAPI IDE Interface.

Following this initial step, the debugger calls the *SysDbgInterface* procedure in the debugger DLL, making available a set of functions in the debugger that the DLL should call to notify the debugger of events such as program termination, thread activity, etc.

Having completed these two steps, the interface between the DLL and the debugger is established.

To produce a new debugger DLL or change the existing functionality of the debugger, create a copy of the *VpDbgApi.Pas* unit, make the desired changes, and compile/link the *VpDbgDll.Pas* library file. The updated DLL should replace the *VpDbgDll.Dll* file that ships with Virtual Pascal and will automatically will used the next time the IDE is started.

The ODAPI System Interface

The System Interface defines a set of functions that can and should be used by the debugger DLL both to get information only available to the debugger and the IDE, and to keep the debugger informed about events taking place in the debugger DLL.

The debugger DLL should call the *SysDbgSetInterface* procedure and store the result, which is a record of type *TSysDbgInterface* containing the entry points of a number of useful routines in the debugger itself.

The definition of the System interface can be found in *VpDbgApi.Pas*:

TSysDbgInterface = **record**

GetThreadParam: **function**(*No*: Integer): *PSysDbgThreadIds*;

ThreadCreated: **procedure**(*ThreadID*, *ThreadHandle*, *ThreadOrdinal*: Longint);

ThreadExited: **procedure**(*ThreadID*, *ExitCode*: Longint);

DllLoaded: **procedure**(*DllName*: PChar; *DllHandle*, *SegCount*: Longint; **const** *SegTable*: **array** of *TSysDbgSegDef*);

DllUnloaded: **procedure**(*DllHandle*: Longint);

```

ProcessExited: procedure(ExitCode, ExitType: Integer);
NotifyException: procedure(const DbgEvent: TSysDbgEvent);
StopOnException: function(Code: Longint): Boolean;
end;

```

Each field of the *TSysDbgInterface* structure is a function or procedure pointer, as can be seen from the definition above. The functions should be called from the debugger for the following purposes:

GetThreadParam	- Ask debugger for information about a thread
ThreadCreated	- Notify debugger that a thread was created
ThreadExited	- Notify debugger that a thread ended
DllLoaded	- Notify debugger that a DLL was loaded
DllUnloaded	- Notify debugger that a DLL was unloaded
ProcessExited	- Notify debugger that process ended
NotifyException	- Notify debugger that an exception was raised
StopOnException	- Ask debugger if exception is of interest

A debugger DLL must export a single entrypoint, *SysDbgGetIDEInterface*, which is called by the IDE during startup. This call defines an additional set of entrypoints for the debugger to use during the debugging process; these are described in the next section on the ODAPI IDE Interface.

The ODAPI IDE Interface

To extend the debugger of Virtual Pascal, the ODAPI IDE Interface should be used. Implement the functionality by writing a Dynamic Link Library *VpDbgDll.Dll* exporting an entrypoint, *SysDbgGetIDEInterface*, which is defined as follows:

```

procedure SysDbgGetIDEInterface(var IDEInt: TSysDbgIDEInterface);

```

When called, the procedure must change the *IDEInt* parameter to a valid *TSysDbgIDEInterface*, as defined in the following.

Each field of this record (except *SysDbgVersion*) is a function or procedure pointer and must point to valid code after the call.

```

TSysDbgIDEInterface = record
  SysDbgVersion: Longint;
  SysDbgPlatforms: TSysDbgPlatforms;
  SysDbgInitialize: procedure;
  SysDbgGetFlatInfo: procedure(var FlatInfo: TSysDbgFlatInfo);
  SysDbgSetInterface: procedure(var DbgInt: TSysDbgInterface);
  SysDbgStartProcess: function(const FileName, CmdLine: String; AppType: Longint; var
  ProcessID, SesID, EntryAddr: Longint): Longint;
  SysDbgTerminateProcess: function: Longint;
  SysDbgSelToFlat: function(Sel, ofs: Longint): Longint;
  SysDbgReadMemory: function(Sel, ofs: Longint; Buffer: Pointer; Size: Longint): Longint;
  SysDbgWriteMemory: function(Sel, ofs: Longint; Buffer: Pointer; Size: Longint): Longint;
  SysDbgReadRegisters: function(Regs: PSysDbgCPURegisters; FPUState:
  PSysDbgFSaveFormat): Boolean;
  SysDbgWriteRegisters: function(Regs: PSysDbgCPURegisters; FPUState:
  PSysDbgFSaveFormat): Boolean;
  SysDbgFreezeThread: function(const Regs: TSysDbgCPURegisters): Boolean;
  SysDbgResumeThread: function(const Regs: TSysDbgCPURegisters): Boolean;
  SysDbgGetThreadState: function(const Regs: TSysDbgCPURegisters; var State:
  TSysDbgThreadState): Boolean;

```

```
SysDbgSetWatchPoint: function(LinAddr: Longint; BkptLen,BkptType: Byte; ThreadID: Longint):  
Longint;  
SysDbgClearWatchPoint: procedure(Id: Longint);  
SysDbgExecute: procedure(Step: Boolean; Regs: TSysDbgCPURegisters; var DbgEvent:  
TSysDbgEvent);  
SysDbgWaitUserScreen: procedure(Delay: Longint);  
SysDbgSetHardMode: procedure(Hard: Boolean);  
SysDbgSwitchScreen: procedure(User: Boolean);  
end;
```

The debugger uses the fields of the *TSysDbgIDEInterface* for the following purposes:

- SysDbgVersion* - The version of the DLL
- SysDbgInitialize* - Initialization code for DLL
- SysDbgGetFlatInfo* - Get DS and CS selectors
- SysDbgSetInterface* - Define debugger interface
- SysDbgStartProcess* - Start debugging process
- SysDbgTerminateProcess* - Stop process
- SysDbgSelToFlat* - Translate Seg:Ofs to flat address
- SysDbgReadMemory* - Read memory bytes
- SysDbgWriteMemory* - Write memory bytes
- SysDbgReadRegisters* - Read registers or FPU state
- SysDbgWriteRegisters* - Write registers or FPU state
- SysDbgFreezeThread* - Freeze a thread
- SysDbgResumeThread* - Resume frozen thread
- SysDbgGetThreadState* - Get the state of a thread
- SysDbgSetWatchPoint* - Set a watch point
- SysDbgClearWatchPoint* - Unset a watch point
- SysDbgExecute* - Execute debug process
- SysDbgWaitUserScreen* - Wait for a while or keypress
- SysDbgSetHardMode* - (OS/2) Set hard/soft PM debug mode
- SysDbgSwitchScreen* - Switch to/from user screen

ODAPI Types

The following types are defined in the *VpDbgApi* unit, and are used to implement the Open Debug API. Please refer to the ODAPI Interface section for information about the *TSysDbgIDEInterface* type.

<i>Type</i>	<i>Description</i>
<i>TSysDbgFlatInfo</i>	Used to pass code and data segment selectors in the <i>SysDbgGetFlatInfo</i> function
<i>TSysDbgThreadIds</i>	Used to return thread identification information using the <i>TSysDbgInterface.GetThreadParam</i> function
<i>TSysDbgSegDef</i>	Used when passing DLL segment information to the <i>TSysDbgInterface.DLLLoaded</i> function
<i>TSysDbgEvent</i>	Used to pass debug event information to the <i>TSysDbgInterface.NotifyException</i> function and to return this information from the <i>SysDbgExecute</i> call

<i>TSysDbgInterface</i>	Used to define a set of functions in the debugger useable by the DLL: The ODAPI System Interface
<i>TSysDbgCPURegisters</i>	Thread ID/handle identifier and normal registers, used by the <i>SysDbgReadRegisters</i> , <i>SysDbgWriteRegisters</i> and various other functions to pass thread and register information
<i>TSysDbgFSaveFormat</i>	The state of the floating point unit (FPU), used by the <i>SysDbgReadRegisters</i> and <i>SysDbgWriteRegisters</i> functions
<i>TSysDbgThreadState</i>	Thread state, as returned by the <i>SysDbgGetThreadState</i> function

ODAPI Examples

The one ODAPI-specific example included with Virtual Pascal is included in the form of the source code for the debugger DLL used for debugging Win32 and OS/2 applications from the Virtual Pascal IDE.

The source code for this is included in full in *Source\Rtl\VpDbgApi.Pas*.

Error messages

This section details both Compiler error messages output by the Virtual Pascal compiler when it encounters and Run-time error messages, which are errors that can occur when the compiled program is executed.

Compiler error messages

#	Description
1	<i>Out of memory</i> This error occurs when the compiler has run out of memory. There is not enough memory available. Make more room for the OS/2 swap file or increase the Windows Virtual Memory figure and re-run the compiler.
2	<i>Identifier expected</i> An identifier was expected at this point. It might be that you tried to redeclare a reserved word. Reserved words are words that have a special meaning in Pascal, like asm , implementation , string , nil , etc. See chapter 1 for a list of Pascal identifiers.
3	<i>Unknown identifier</i> This identifier has either not been declared, or might not be visible within the current scope.
4	<i>Duplicate identifier</i> The identifier has already been defined with another meaning within the current scope, representing a constant, a variable, a type, a module name, etc.
5	<i>Syntax error</i> An illegal character was found in the source text. You might have forgotten the quotes around a string constant.
6	<i>Error in real constant</i> The floating point constant that you have typed is invalid. Examples of valid floating point constants are 10, 10.2 and -10.2e+3. See Chapter 1 for details.
7	<i>Error in integer constant</i> The integer constant that you have typed is invalid. If you want to supply a whole real number outside the maximum integer range you should follow it by a decimal point and a zero.
8	<i>String constant exceeds line</i> You have probably forgotten the ending quote in a string constant.
9	<i>Unexpected end of program</i> The compiler has encountered the final end of the main statement part in an include file. Probably, the begins and ends are unbalanced.
10	<i>Unexpected end of file</i> The error probably occurred because of one of the following: <ul style="list-style-type: none">• An include file ends in the middle of a statement part. Every statement part must be entirely contained in one file.• You forgot to close a comment. Check that all comments are closed, particularly in the {&Comments+} state.
11	<i>Line too long</i> The maximum line length is 255 characters.
12	<i>Type identifier expected</i> The identifier does not denote a type as it should.

- 13 *Too many open files***
Too many files are open. The include files are nested too deeply.
- 14 *Invalid file name***
The file name is invalid or specifies a non-existing 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 another disk.
- 17 *Invalid compiler directive***
The following can cause this error:
- The compiler directive name is unknown.
 - The parameter supplied to the compiler directive is invalid.
 - A global compiler directive is used after compilation of the program's (unit's or library's) body has begun.
- 18 *Argument needs type override***
The expression needs to have a specific size or type supplied, since its size can not be determined from the context, for example:
`mov [ebx],1`
You can usually correct this error by using the **ptr** operator to set the size of the operand:
`mov dword ptr [ebx],1`
- 19 *Undefined type in pointer definition***
The type was referenced in a pointer-type declaration, 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 4G 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 up to 256 possible values.
- 24 *File components may not be files or objects***
The component type of a file type cannot be an object type or a file type, or any structured type with a file or object type component.
- 25 *Invalid string length***
The maximum length of a string declaration must be in the range 1..255.
- 26 *Type mismatch***
This error occurs because of one of the following:
- The types of the expression and the variable in an assignment statement are incompatible.
 - The type of the actual and the formal parameters when calling a procedure or function are incompatible.
 - Indexing an array with an expression type incompatible with the index type.
 - Using incompatible types of operands in an expression.
- 27 *Invalid subrange base type***
Only 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***
String types, real types, structured types and pointer types are not allowed here. Ordinal types include integers, enumerated types, booleans and character types.
- 30 ***Integer constant expected***
Only an integer constant is allowed here.
- 31 ***Constant expected***
Only a constant is allowed here.
- 32 ***Integer or real constant expected***
Only a numeric constant is allowed here.
- 33 ***Pointer type identifier expected***
The identifier does not denote a pointer type as it should.
- 34 ***Invalid function result type***
Files types are not allowed as function result type.
- 35 ***Label identifier expected***
The identifier does not denote a label as it should.
- 36 ***begin expected***
A **begin** was expected here, or else there is an error in the block structure of the unit, program or library.
- 37 ***end expected***
An **end** was expected here, or else there is an error in the block structure of the unit, program or library.
- 38 ***Integer expression expected***
The expression must be of an integer type.
- 39 ***Ordinal expression expected***
The expression must be of an ordinal type.
- 40 ***Boolean expression expected***
The expression must be of a boolean type.
- 41 ***Operand types do not match operator***
The operator cannot be applied to operands of this type. For example, *'Hello' * 3* is invalid.
- 42 ***Error in expression***
This expression does not satisfy the Pascal expression syntax. You might 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 record or object variable.
- 45 ***Operand types do not match***
The size of an instruction operand does not match either the other operand or one valid for the instruction, for example:
mov eax,ABC
where *ABC* is a variable of type *Byte*.

46 Address size conflict

Addressing mode (16 or 32-bit) of the instruction conflicts with addressing mode (Small or Large) of the operand, for example:

{ Variable is defined in 32-bit segment, that is why it is Large }

var

Abc: Byte;

asm

mov al,Abc[bx]

end;

[bx] is allowed only in 16-bit addressing mode, so an error is reported. The valid form of this instruction is:

mov al,small Abc[bx]

47 Invalid instruction

While assembling an instruction, you failed to supply an instruction mnemonic or mistyped it, for example:

eax,\$12345678

move eax,\$12345678

instead of *mov eax,\$12345678*

48 Library too big

The maximum size of the object library file, generated for a unit in the **{&SmartLink+}** state is 1M bytes. Please split your unit in two or more smaller ones.

49 Current procedure or function cannot be inlined

The current procedure or function cannot be inlined because of one of the following:

- It contains a nested procedure or function;
- Its declaration part contains a typed constant declaration;
- It has a value parameter that should be copied to the local stack.

50 do expected

The reserved word **do** does not appear where it should.

51 Argument to instruction has illegal size

The size of the argument that you supplied to the instruction is invalid, for example:

push QWord Ptr [ebx]

52 Invalid number of operands to instruction

You have specified either too many or too few operands for the instruction, for example:

mov eax,ebx,1 { Too many operands }

mov eax { Too few operands }

53 Invalid ALIGN boundary

The argument that you have supplied to the BASM *Align* pseudo instruction is invalid. The valid forms are:

Align 1 { No alignment }

Align 2 { Align code to a word (2-byte) boundary }

Align 4 { Align code to a dword (4-byte) boundary }

You cannot align to a boundary greater than 4, because the code segment itself is double word aligned.

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** *Object record too long*
The built-in linker has encountered an object file record or object library record that is too long. The maximum object record size is 4096 bytes.
The object or library file is probably corrupt.
- 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*
This error can occur for one of several reasons:
- The procedure or function was declared in the interface part of a unit, but its declaration never occurred in the implementation part.
 - The procedure or function was declared forward, but its definition was not found.
 - The method was declared in an object type, but its implementation was never found. It might be that you just forgot to put the object type name with a period before the method name.
- 60** *Cannot be modified*
You have attempted to modify a data element that cannot be changed, such as a constant, structured (array, record or object) or file variable.
- 61** *Invalid typecast*
Here are some possible sources of this error:
- In a variable typecast, the sizes of the variable reference and the destination type differ.
 - You are attempting to typecast an expression where only a variable reference is allowed.
- 62** *Division by zero*
This /, DIV, or MOD operation causes a division by zero.
- 63** *Invalid file type*
The file-handling procedure does not support the given file's type. For example, you might have made a call to *WriteLn* with a typed file or *FileSize* with a text file.
- 64** *Cannot Read or Write variables of this type*
Read and *ReadLn* can input variables of type *Character*, *Integer*, *Real* and *String*.
Write and *WriteLn* can output variables of types *Character*, *Integer*, *Real*, *String* and *Boolean* only.
- 65** *Pointer variable expected*
This variable must be of a pointer type.
- 66** *String variable expected*
This variable must be of a string type.
- 67** *String expression expected*
This expression must be of a string type.
- 68** *Circular unit reference*
Two units are not allowed to use each other in their interface parts, although they can use each other in the implementation parts. Rearrange your uses clauses so that circular references occur only in the implementation parts.
- 69** *Unit name mismatch*
The name of the unit found in the .VPI 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 this unit was compiled. Use *Make* or *Build* to automatically recompile the units that have changed.

71 *Internal stack overflow*

The compiler's internal stack is exhausted due to too many levels of nested statements. Rearrange the code to be less deeply nested.

72 *Unit file format error*

The .VPI file is somehow invalid. Make sure it is created by the current version of the Virtual Pascal. If you want to give to somebody an object file of the unit produced by Virtual Pascal, but do not want to give away the source code, Virtual Pascal supports the concept of *Pure Interface Units*. Create an interface unit, containing the interface part of the unit only and compile this by the newer version of the compiler. See the description of the `{&PureInt}` directive for details.

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 or object variable expected*

This variable must be of a record or object type.

76 *Constant out of range*

You are trying to do one of the following:

- Index an array with an out-of-range constant.
- Assign an out-of-range constant to a variable.
- Pass an out-of-range constant as a parameter to a procedure or function.

77 *File variable expected*

This variable must be of a file type.

78 *Pointer expression expected*

This expression must be of a pointer type.

79 *Integer or real expression expected*

This expression must be of an integer or a real type.

80 *Label not within current block*

You are trying to do one of the following:

- Reference a label that is outside the current block.
- Jump out of an ASM..END statement body.
- Define a Pascal label (declared in the Label declaration part) within an `asm..end` statement body.

81 *Label already defined*

The label already marks a statement.

82 *Undefined label in preceding statement part*

The label was declared and referenced in a 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. This may indicate that a type is being used as a variable or that the name of the program itself overrides an important identifier from another unit.
- "95 **".." *expected***
A subrange does not appear where it should.
- 96 ***Too many variables***
This error may occur, if :
- The total size of the global variables declared within a program or unit has exceeded 4G bytes.
 - The total size of the local variables declared within a procedure or function has exceeded 2G bytes.
- 97 ***Invalid for control variable***
The **for** statement control variable must be a simple variable defined either in the declaration part of the current subprogram, or be a global variable or typed constant.
- 98 ***Longint variable expected***
This variable must be of an *Longint* type.
- 99 ***File and procedure types are not allowed here***
A typed constant cannot be of a file 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 or object-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***
This variable must be of an integer or real type.
- 104 ***Ordinal variable expected***
This variable must be of an ordinal type.
- 105 ***Invalid array base type***
An empty record type must not be the base type of an array.
- 106 ***Character expression expected***
This expression must be of a character type.
- 107 ***external expected***
Virtual Pascal generates 32-bit code only. For this reason, procedures and functions declared as being **Far16** must be also declared as external.

108 *Overflow in arithmetic operation*

The result of the preceding arithmetic operation is not in the *Longint* range: 2147483648..2147483647. Correct the operation or use real-type values, for example *Comp*, instead of integer-type values.

109 *No enclosing for, while, or repeat statement*

The *Break* and *Continue* standard procedures cannot be used outside a **for**, **while**, or **repeat** statement.

110 *Local threadvar declarations are not allowed*

Variables located in thread local storage (TLS) can be defined only in the outermost scope of a program, a library or a unit.

threadvar definitions within procedures or functions are not allowed.

111 *Invalid or unsupported object file record*

The built-in linker has encountered an object record that is not valid or is not supported. The built-in linker supports only a subset of all object file records, the most notable exception being COMDEF records, used by many C/C++ compilers. Make sure that the object or library file is valid. If linking external object code produced by another compiler, use the OMF linker supplied with this compiler instead of the one built into Virtual Pascal. Note, that some compilers also generate proprietary record types which are supported by their linkers only.

112 *case constant out of range*

The **case** selector constants must be within the allowed range of the case selector expression. For example, if the selector expression has type *Byte*, selector constants must be within *Byte* range(0..255).

113 *Error in statement*

This symbol cannot start a statement. Most likely you have put semicolon (;) before the **else** part of a conditional statement.

114 *Syntax error in module definition file*

The module definition file or **&Linker** statement does not conform to the allowed syntax. Please refer to the module definition file syntax on page 136; note, that all statements must be entered in UPPER CASE.

115 *Duplicate case constant*

This case selector overlaps with a previous one, for example:

```
case C of
```

```
  '0'..'9','A'..'F': HandleHexDigit;
```

```
  'A'..'Z','a'..'z': HandleLetter; { You will get an error here }
```

```
end;
```

116 *Duplicate public symbol*

The same public symbol is declared in more than one object file module.

Make sure that the same object file is not linked twice (check the .LNK file in the output directory) and check the setting of the **{&OrgName}** compiler directive. In most cases, the default setting of **{&OrgName-}** should be used.

117 *Undefined external symbol*

The built-in linker did not find a corresponding public declaration for this external symbol.

When referencing symbols located in a DLL, make sure that the name of the import library is included in a **{&L LibName.Lib}** or **{&Dynamic LibName.Lib}** compiler directive.

Also check the setting of the **{&OrgName}** compiler directive and the **orgname** standard procedural directive if the symbol is a procedure or a function.

118 *Include files are not allowed here*

Every statement part must be entirely contained in one file.

- 119** *No inherited methods are accessible here*
You are using the **inherited** reserved word outside a method or in a method of an object type that has no ancestor.
- 120** *Segment too large*
The size of this segment exceeds 4GB.
- 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*
This construct follows the syntax of a variable reference, but does not denote a memory location.
You are probably calling a pointer function, but forgetting to dereference the result.
- 123** *Too many symbols*
The total size of all symbols in your project is greater than 4MB. This is the limit for this version of the compiler. If you are compiling with **{\$L+}**, try turning it off.
- 124** *Statement part too large*
Virtual Pascal has encountered a statement part that is too large to handle. Move sections of the statement part into one or more procedures - not only will you be able to compile the program, it will also benefit from enhanced readability.
- 125** *Undefined class in the preceding declaration*
The class name was declared in a forward declaration, but was not fully declared afterwards.
- 126** *Files must be var parameters*
You are attempting to declare a file type value parameter. File-type parameters must be **var** parameters.
- 127** *Invalid executable file*
The executable file specified as a stub is not a valid DOS executable.
- 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 compilation ended within a conditional compilation construct. There must be an equal number of **{\$IFxxx}**s and **{\$ENDIF}**s.
- 130** *Error in initial conditional defines*
The initial conditional symbols specified in Options|Compiler|Conditional defines (or in a /D command line option) are invalid. Virtual 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 an interface part or **forward** declaration does not match this header.
- 132** *Fixup overflow*
The fixups in the specified object module have overflowed. This error may occur because of an incorrect external reference in an external assembler module.
- 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 might be attempting to use a function that is available only at run time.
- 134** *Expression incorrectly terminated*
Virtual Pascal expects either an operator or the end of the expression at this point, but found neither.

- 135 *Invalid format specifier***
You are using an invalid format specifier, or the numeric argument of a format specifier is out of range (valid range is 2..18).
- 136 *Invalid indirect reference***
The statement attempts to make an invalid indirect reference.
- 137 *Structured variables are not allowed here***
You are attempting to perform a non-supported operation on a structured variable.
- 138 *Cannot evaluate without System unit***
The IDE cannot find the file SYSTEM.VPI. Enter directory name that contains SYSTEM.VPI in the Options|Directories|Unit Directories input box and re-run the IDE. The default is X:\VP\UNITS, where X: is the drive where VP is installed.
- 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) can not be accessed until you actually run the program.
It is not possible to access symbols in DLLs compiled in the **{SmartLink+}** state.
- 140 *Invalid floating-point operation***
An operation on two real type values produced an overflow or a division by zero.
- 141 *Duplicate ordinal number***
More than one symbol specifying the same ordinal number is being exported.
- 142 *Pointer or procedural variable expected***
The *Assigned* standard function requires the argument to be a variable of a pointer or procedural type.
- 143 *Invalid procedure or function reference***
You are attempting to call a procedure in an expression.
- 144 *Entry table is too large***
The format of the OS/2 LX executable allows a maximum of 65,535 entries in the entry table. This dynamic link library exceeds this limitation. Split it into two or more smaller DLLs.
- 145 *Invalid resource file***
The resource file being linked is corrupt or invalid. Try recompiling the .RC source file on the command line and retry. See also **{&Open32}**
- 146 *File access denied***
Typically, the operating system returns this error when you are either
- trying to use a read-only file as an output file, or
 - using a directory name as an output file
- 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, a library or a unit. Object type definitions within procedures and functions are not allowed.
- 149 *virtual expected***
The reserved word **virtual** is missing. You are defining a method, which was declared virtual in the ancestor object.
- 150 *Method identifier expected***
The identifier does not denote a method.
- 151 *Virtual constructors are not allowed***
Borland Pascal 7 style objects, declared using the **object** reserved word, must have static constructor methods only.
- 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 instruction you are trying to assemble has one or more operands that are not allowed, for example:

```
var Var1,Var2: Byte;
asm
  mov  Var1,Var2
  pop  Var1
  shld eax,ebx,Var1
end;
```

156 *Memory reference expected*

The assembler operand is not a memory reference, which is required here. Most likely you have forgotten to put square brackets around an index register operand, for example,

```
mov  eax,eax+ebx
instead of
```

```
mov  eax,[eax+ebx].
```

If you want to add *ebx* to *eax*, use

```
add  eax,ebx
instead.
```

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.

158 *Invalid register combination*

The memory reference does not contain one of the permitted combinations of base and index registers. The valid combinations are:

16-bit addressing mode: *[bx]*, *[bp]*, *[si]*, *[di]*, *[bx+si]*, *[bx+di]*, *[bp+si]*,*[bp+di]*;

32-bit addressing mode:

- any 32-bit general purpose register (*eax,ebx,ecx,edx,esi,edi,ebp,esp*);
- sum of any two 32-bit general purpose registers;
- one 32-bit general purpose register multiplied by 1, 2, 4 or 8 (except the *esp* register);

Local variables are always allocated on the stack and accessed via the *ebp* or *esp* register. The assembler automatically adds *[ebp]* in references to such variables, so even though a construct like *Local[si]* (where *Local* is a local variable) appears valid, it is not since the final operand would become *Local[ebp+si]*.

Note: If an assembler procedure or function has no stack frame (**&Frame**), local variables are accessed via the *esp* register, otherwise the *ebp* register is used.

159 *Invalid instruction for selected processor*

You are trying to assemble an instruction that is not valid for the selected processor. For example, the *CMPXCHG* instruction is invalid in the **{SG3+}** state and *CPUID* (Only available on Pentium and later CPUs) is invalid in the **{SG4+}** state.

160 Invalid symbol reference

This symbol cannot be accessed in an assembler operand. Possible causes are:

- You are attempting to access a standard procedure, a standard function, or the *Mem*, *MemW*, *MemL*, *Port*, *PortW* or *PortL* special arrays in an assembler operand.
- You are attempting to access a string, floating-point, or set constant in an assembler operand.
- You are attempting to access the *@Result* special symbol outside a function or in a function returning an integer or pointer value.
- You are attempting to generate a short JMP instruction that jumps to something other than a label.

161 Code generation error

The preceding statement part contains a LOOPNE, LOOPE, LOOP, JCXZ or JECXZ instruction that cannot reach its target label.

162 ASM expected

The reserved word **asm** does not appear where it should.

163 Duplicate dynamic method index

This dynamic method index has already been used by another method. You may be trying to override a dynamic method but have misspelled its name and thus introduced a new method.

164 Internal compiler error

The Virtual Pascal compiler has encountered an internal processing error. Please record its number and location and contact fPrint UK Ltd.

165 Duplicate resource identifier

This resource file contains a resource with a name or ID that has already been used by another resource

166 No dynamic unit information is available

An import library specified in a **{&Dynamic}** compiler directive has not been generated by the Virtual Pascal compiler.

For import libraries generated using the ImpLib utility, use the **{\$L}** directive instead.

172 Read or Write clause expected

A property definition must include at least a **read** or **write** specifier or both.

173 Cannot read a write-only property

It is not possible to read the value of a property that does not have a read specifier and consequently no read access method. You can only assign a new value to it.

174 Cannot assign to a read-only property

It is not possible to assign new value to a property that does not have a write specifier and consequently no write access method. You can only read the current value of it.

175 Cannot exit a finally block

finally statement list must always be executed. It is not possible to leave it by means of the *Exit* standard procedure.

176 Label and goto not at same nesting level

The label referenced by a **goto** statement must be in the same block as the **goto** statement.

177 on expected

In the exception block of a **try..except** statement you must use the reserved word **on** before the exception type identifier.

- 178** *Cannot mix class and object types*
Borland Pascal 7 style **objects** and Delphi style **classes** are not compatible. It is not possible to mix them - either inherit one type from the other or assign the value from a variable of one type to the other.
- 179** *procedure or function expected*
The declaration of a class method must start from the **class** reserved word and be followed by either of the reserved words **procedure** or **function**.
- 180** *Class type identifier expected*
In the exception block of the **try..except** statement the reserved word **on** must be followed by a class type identifier representing an exception class.
- 181** *Class expression expected*
The expression in the **raise** statement must be of a class instance type.
- 182** *Instance variable is not accessible here*
It is not possible to access the instance of a **class** in the body of a class procedure or function. The *Self* class pointer is not available either.
- 183** *Invalid method reference*
It is not possible to call non-class methods from within a class method, since non-class methods need an instance of the class to be specified.
- 184** *Default property must be an array property*
You have declared a default property that is not of an array type or without assigning it a value. You might have forgotten to define a value after the **default** directive.
- 185** *Class already has a default property*
Only one **default** property is allowed. It is not possible to change the **default** if the ancestor defines a **default** array property.
- 186** *Invalid message handler parameter list*
The message handler cannot be declared with the parameter list given. The message handler must have one variable parameter of any type.
- 187** *Method does not exist in base class*
You can override only methods that exist in the ancestor class. You might have misspelled the method name.
- 188** *Cannot override a static method*
Only **dynamic** or **virtual** method can be overridden.
- 189** *Property does not exist in base class*
Property overrides are allowed only for properties declared in the ancestor class. You might have misspelled the property name.
- 191** *Type not supported in expression list*
Only the following types can be used in **array of const** constructor:
- ordinal types;
 - float types;
 - pointer types;
 - string types;
 - class types;
 - class reference types;
- 192** *Property access method not found*
The property you are trying to evaluate is write-only. It is not possible to evaluate a property without a **read** access method.
- 195** *Exception raised during evaluation*
An exception occurred while evaluating an expression in the integrated debugger.
- 198** *raise not allowed outside except..end block*
You can only re-raise an exception within an **except** block.

200 **Published not allowed in this class**

If class is defined in the **{M-}** state and is not derived from a class that was declared in the **{M+}** state, **published** sections are not allowed in the class declaration.

201 ***This field cannot be published***

Only class type fields can be **published**.

202 ***This property cannot be published***

The property does not satisfy the requirements for a **published** property.

Run-time error messages

#	Description
1..88	<i>Operating system errors</i>
< 0	Runtime errors with numbers 1..88 are reported by the operating system. For operating system errors greater than 88, run-time error numbers are negative. Run the HELP command to obtain a definition and brief description of the error. For example, if you get run-time error #16, type the following line at the command prompt: <i>HELP 16</i>
100	<i>Disk read error</i> <i>Read</i> reports this error on a typed file if <ul style="list-style-type: none"> • you attempt to read past the end of the file; • An I/O error occurs during the read operation.
101	<i>Disk write error</i> <i>BlockWrite</i> , <i>Close</i> , <i>Flush</i> , <i>Write</i> and <i>WriteLn</i> report this error if <ul style="list-style-type: none"> • the disk becomes full; • I/O error occurs during the write operation.
102	<i>File not assigned</i> <i>Append</i> , <i>Reset</i> and <i>Rewrite</i> report this error if the file variable has not been assigned a name through a call to <i>Assign</i>
103	<i>File not open</i> <i>BlockRead</i> , <i>BlockWrite</i> , <i>Close</i> , <i>Eof</i> , <i>FilePos</i> , <i>FileSize</i> , <i>Flush</i> , <i>Read</i> , <i>Seek</i> , <i>Truncate</i> and <i>Write</i> report this error if the file is not open.
104	<i>File not open for input</i> <i>Eof</i> , <i>Eoln</i> , <i>Read</i> , <i>ReadLn</i> , <i>SeekEof</i> and <i>SeekEoln</i> report this error on a text file if the file is not open for input.
105	<i>File not open for output</i> <i>Write</i> and <i>WriteLn</i> report this error on a text file if the file is not open for output.
106	<i>Invalid numeric format</i> <i>Read</i> and <i>ReadLn</i> report this error if a numeric value read from a text file does not conform to the proper numeric format.
200	<i>Division by zero</i> The program attempted to divide a number by zero during a <i>/</i> , mod , or div operation.
201	<i>Range check error</i> This error is reported by statements compiled in the {SR+} state when one of the following situations arises: <ul style="list-style-type: none"> • The index of an array was out of range. • The program attempted to assign an out-of-range value to a variable. • The program attempted to pass an out-of-range value as a parameter to a procedure or function
202	<i>Stack overflow error</i> The program reports this error on entry to a procedure or function compiled in the {SS+} 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 StackSize} compiler directive or by specifying a large stack in the Options Compiler dialogue. The Stack Overflow error can also be caused by infinite recursion or by an assembly language procedure that does not maintain the stack properly.

203 Heap overflow error

New and *GetMem* report this error when there is not enough free space in the heap to allocate a block of the requested size.

In OS/2, this error occurs only when the MEMMAN command in the CONFIG.SYS file contains:

- NOSWAP parameter or
- SWAP and COMMIT parameters.

If you get a Heap Overflow error, make more room for the OS/2 swap file or use the SWAP MEMMAN parameter instead of NOSWAP.

In Windows, increase the Virtual Memory setting.

204 Invalid pointer operation

Dispose and *FreeMem* report this error if the pointer is nil or if it points to a location outside the heap.

205 Floating point overflow

A floating-point operation produced a number too large for the numeric coprocessor to handle. Normally this error does not occur unless you explicitly unmask overflow exception by clearing the OM bit in the coprocessor control word.

206 Floating point underflow

A floating-point operation produced an underflow. Normally this error does not occur unless you explicitly unmask underflow exception by clearing the UM bit in the coprocessor control word.

207 Invalid floating point operation

One of the following floating-point errors occurred:

- The real value passed to *Trunc* or *Round* could not be converted to an integer within the *Longint* range(-2147483648 to 2147483647).
- The argument passed to the *Sqrt* function was negative.
- The argument passed to the *Ln* function was zero or negative.
- A coprocessor stack overflow occurred.

Normally this error does not occur unless you explicitly unmask invalid operation exception by clearing the IM bit in the coprocessor control word.

208 Inexact floating point result

A floating-point operation produces a value that has partially lost its precision. For example, the result of the division 1/7 cannot be exactly stored in any binary format used by the coprocessor.

Normally this error does not occur unless you explicitly unmask precision exception by clearing the PM bit in the coprocessor control word.

209 Denormalised floating point operand

This error is reported when one or two operands of a floating point operation are denormalised numbers.

Normally this error does not occur unless you explicitly unmask denormal operand exception by clearing the DM bit in the coprocessor control word.

210 BP: Object not initialised

With range-checking on, you made a call to an object's virtual method, without initialising the object via a constructor call.

Delphi: Call to an abstract method

Your program tried to call an abstract virtual method.

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.

212 *Stream registration error*

This error is generated by the *RegisterType* procedure in the *Objects* unit. It indicates that one of the following errors has occurred:

- The *ObjType* field of the stream registration record is 0.
- 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 *TCollection* method is out of range.

214 *Collection overflow error*

This error is reported by a *TCollection* if an attempt is made to add an element when the collection cannot be expanded.

215 *Arithmetic overflow error*

Reported by statements compiled in the **{Q+}** state when an integer arithmetic operation caused an overflow or the result was outside the supported range.

216 *Access violation*

This error is reported if you try to access memory that it is not legal for your application to access. The following practices cause access violation:

- Loading constant values into segment registers;
- Using segment registers for temporary storage;
- Writing to code segment;
- Accessing memory beyond the address space given to your application;
- Dereferencing **nil** pointers.

217 *Unhandled exception*

An exception has occurred for which an exception handler cannot be found.

219 *Invalid type cast*

The object on the left side of an **as** operator is not of the class given on the right side of the operator.

APPENDIX E

Naming conventions

To satisfy the scope rules of the Object Pascal language, the Virtual Pascal compiler uses the following naming conventions for public identifiers in object and library files:

UnitName@SymbolName
UnitName@ObjectName@MethodName

where

- *UnitName* is the name of the program, library or unit where the symbol *SymbolName* is defined. If a program is written without a heading (no program name is given), a default of *PROGRAM* is assumed.
- *SymbolName* is the name of a variable, typed constant, procedure or function.
- *ObjectName* is the name of an object or class type
- *MethodName* is the name of an object or class method.

All names keep their original case. However, for variables, typed constants, procedures and functions you can change the default naming convention by specifying the **{&OrgName+}** switch compiler directive. This forces the compiler to use the original name of the symbol, retaining its case. Its primary purpose is for declaring external OS/2 API functions - for examples of this, refer to the *Os2Base* unit source code. **{&OrgName+}** is also useful when interface variables, procedures or functions written in Virtual Pascal should be used by modules written in other languages.

The **orgname** standard procedural directive can be used to override the default setting of the **{&OrgName}** switch for a single procedure or function.

Index

A

- absolute
 - clause, 73
 - variables, 73
- abstract
 - directive, 55
 - methods, 55
- access specifiers
 - property definitions, 60
- actual parameter list, 83, 94
- actual parameters, 94, 111, 112
- AddExitProc procedure, 20
- address factor, 84
- address of (@) operator, 85
- address-of (@) operator, 84, 164
- ALIAS reserved word, 139
- aligning data, 149
- ancestor of an object/ class type, 48
- ancestors, 48
- and operator, 87
- arithmetic operators, 86
- array
 - types, 40
 - variables, 78
- array properties, 60
 - default, 61
- arrays, 40, 78
 - accessing elements, 40
 - indexing, 78
 - multidimensional, 40
 - number of elements, 40
 - of arrays, 40
 - port access, 92
 - valid index types, 40
 - zero-based character, 36, 41, 75
- array-type constant syntax, 74
- as operator, 63, 91
- asm
 - instructions, 104, 105–6
 - reserved word, 104
- assembler
 - assigning function result, 113
 - declaration syntax, 115
 - procedures and functions, 104
 - register convention, 104
 - altering, 162
 - resolving name conflict, 108
 - restrictions, 109
 - statements, 104
- assignment
 - compatibility, 70

- old-style object type, 94
- statements, 93

- assignment-compatibility
 - class reference types, 63
- AX register*, 113

B

- base type identifier, 34
- binary
 - arithmetic operators, 86
 - operators, 81
- bitwise logical operators, 86
- block
 - defined, 21
 - procedure and function, 111
 - scope, 23
 - syntax, 21
- Boolean
 - data type, 32
 - expression evaluation, 87
 - compiler switch, 151
 - complete, 151
 - complete, 87
 - short-circuit, 151
 - short-circuit, 87
 - logical operators, 87
- boolean data types, 31
- Break procedure, 97
- building an OS/2 application, 24
- built-in assembler
 - operands, 107
 - prefix, 105
 - procedures and functions, 105
 - pseudo-instructions, 106
 - reserved words, 107
- Byte data type, 30
- ByteBool data type, 32

C

- c
 - calling convention, 114
- call
 - models, 113
- calling conventions, 114
- calling VP/2 DLLs
 - from programs written in other languages, 135
- calls
 - near and far, 113
- Cardinal data type, 30, 57

-
- Carriage Return character, 9, 66
 - case
 - sensitivity in module definition files, 136
 - sensitivity of Virtual Pascal, 12
 - statement syntax, 96
 - Char data type, 31
 - character
 - arrays, 41
 - strings, 14, 137
 - checked typecast, 63
 - Chr function, 31
 - class
 - components, 47
 - operators, 91
 - class forward declaration, 50
 - class instance types, 43
 - class methods, 50, 120
 - class references, 50
 - class reserved word, 43, 120
 - class type compatibility, 51
 - class type declarations, 45
 - class types
 - construction, 48
 - domain, 48
 - instances, 48
 - class variables construction, 118
 - class variables destruction, 119
 - classes
 - ancestors, 48
 - components, 43
 - descendants, 48
 - inheritance, 48
 - properties, 58
 - class-reference types, 63
 - ClassType method, 63
 - CODE segment
 - attributes, 139
 - CODE statement, 138, 139
 - comments, 10
 - built-in assembler, 104
 - in module definition files, 136
 - common type of two operands, 87
 - Comp data type, 33, 34
 - comparing
 - packed strings, 90
 - PChar* pointers, 90
 - pointers, 90
 - sets, 91
 - simple types, 90
 - strings, 90
 - compiler directives, 11, 149
 - \$A**, 149
 - \$B**, 87, 151
 - \$D**, 11, 24, 152, 157, 165
 - \$DEFINE**, 17, 152
 - \$ELSE**, 17, 153, 178
 - \$ENDIF**, 17, 153, 178
 - \$H**, 155
 - \$I**, 9, 155, 156
 - \$IFDEF**, 16, 152, 153, 156, 178
 - \$IFNDEF**, 16, 153, 156, 178
 - \$IFOPT**, 16, 153, 157, 178
 - \$J**, 157
 - \$L**, 25, 26, 115, 153, 154, 157, 178
 - \$M**, 52, 76, 147, 158, 159, 183, 184
 - \$P**, 39, 122, 123, 160
 - \$Q**, 161, 186
 - \$R**, 40, 123, 161, 162, 165, 184
 - \$S**, 77, 133, 147, 162, 184
 - \$StdCall**, 163
 - \$T**, 84, 163
 - \$UNDEF**, 17, 152, 156, 164
 - \$V**, 123, 164
 - \$W**, 165
 - \$X**, 36, 37, 41, 83, 88, 90, 112, 165
 - \$Z**, 31, 165
 - &AlignCode**, 150
 - &AlignData**, 77, 149, 150
 - &AlignRec**, 43, 150
 - &Alters**, 104, 150, 162
 - &Asm**, 150, 163
 - &Cdecl**, 114, 151
 - &Comments**, 11, 151, 170
 - &Delphi**, 72, 93, 100, 112, 152
 - &Dynamic**, 128, 133, 153
 - &Export**, 128, 130, 131, 153, 154
 - &Far16**, 114, 151, 154
 - &Frame**, 109, 154
 - &G3,&G4,&G5**, 155
 - &Linker**, 129, 133, 136, 158
 - &LocInfo**, 26, 158
 - &Open32**, 159
 - &Optimise**, 109, 159
 - &OrgName**, 160, 187
 - &PmType**, 160
 - &PureInt**, 25, 161
 - &Saves**, 104, 162
 - &SmartLink**, 24, 150, 163
 - &Speed**, 163
 - &StdCall**, 114
 - &Use32**, 21, 30, 164
 - &Uses**, 105, 109, 164
 - &Zd**, 24, 165
 - global and local, 149
 - compiler error messages, 170
 - complete Boolean evaluation, 88
 - component visibility, 52
 - private, 52
 - protected, 52
 - public, 52
 - published, 52
 - components and scope, 51
 - compound statement, 22, 95
 - syntax, 95
 - concatenation, 88
 - conditional compilation, 16
 - conditional statement syntax, 95
 - CONFORMING segment attribute, 138

constant
 address expression, 73, 76
 declaration part syntax, 22
 declarations, 27
 defined, 27
 expressions, 27
 parameters, 121
 string expressions, assigning to PChar, 36
constant declaration, 73, 74, 173
constants, 27
 address, 76
 array-type, 74
 procedural type, 76
 record-type, 75
 simple-type, 73
 string-type, 73
 structured-type, 74
 typed, 73, 76
 untyped, 27
constructor reserved word, 117
constructor syntax, 117
constructors, 117
 class reference types and, 64
 class variables, 118
 declaration, 116
 Fail, 118
 open array, 124
 virtual, 118
Continue procedure, 97
control characters, 9
CPU386
 symbol, 17
CPU87
 symbol, 17
Currency data type, 33, 34

D

data
 alignment, 77
 segment, 76
 maximum size, 77
data segment, 139
DATA segment attributes, 140
DATA statement, 138, 140
debugging
 extending with ODAPI, 166
 optimisation and, 159
decimal notation, 13
default directive, 61, 62
DefaultHandler method, 57
descendant of an object/class type, 48
DESCRIPTION statement, 141
Destroy destructor, 119. *See also* Free method
destructor reserved word, 117
destructor syntax, 119
destructors, 119

 declaration, 116
digit syntax diagram, 11
direct memory access, 80
directives
 abstract, 55
 assembler, defined, 107
 compiler, 16, 149
 default, 61, 62
 export, 113
 forward, 111, 113
 index, 61
 inline, 116
 nodefault, 62
 override, 53
 standard, 12
 stored, 62
 virtual, 53, 55
DLL
 benefits of using, 127
 calling from other languages, 135
 data segment, 131
 example, 133
 important notes, 133
 methods of creating, 127
 restrictions on debugging, 131
 structure of, 20
 syntax, 127
 termination, 131
domain
 object and class types, 48
DOS
 stub program, 147
Dos unit, 23
double address operator (@@), 85
Double data type, 33, 34
dynamic
 importing, 128, 129
 methods, 55
 object variables, 49
 variable, 79
dynamic link libraries, 20, 127. *See also* DLL
Dynamic Method Table, 55

E

empty
 record, 42
 set, 64
 string, 15
enumerated types, 31
environment variables
 LIB, 158
 Manually initialising, 135
 PATH, 147
error messages, 170
Examples
 ODAPI, 169

-
- exception
 - handler, 100
 - on..do, 103
 - raise, 101
 - try..except, 102
 - try..finally, 103
 - Exception class, 100
 - exceptions, 100
 - Exclude procedure, 65
 - EXECUTEONLY segment attribute, 138
 - EXECUTEREAD segment attribute, 138
 - EXETYPE statement, 141
 - exponents, 34
 - export directive, 113, 130
 - exporting procedures and functions, 127
 - exporting symbols, 127
 - exporting symbols from a DLL, 130
 - using **{&Export}** compiler directive, 131
 - using **export** directive, 130
 - using **exports** clause, 130
 - using module definition files, 131
 - exports clause, 127
 - syntax, 130
 - EXPORTS statement, 141
 - expression syntax, 81–83
 - expressions, 81–92
 - assembler, 110
 - constant, 27
 - constant address, 73
 - order of evaluation, 87
 - Extended data type, 33, 34
 - extended syntax, 35, 36, 37, 41, 165
 - external
 - declaration, 115
 - directive, 115, 127, 129
 - procedures and functions, 129
-
- F**
- factor syntax, 82
 - Fail
 - constructor failure, 118
 - False boolean constant, 31
 - far declaration, 113
 - far16 calling convention, 114
 - field
 - designator syntax, 79
 - list of records, 41
 - of a record, 79
 - fields
 - in record types, 42
 - object and class types, 43
 - file
 - types, 65
 - file types. *See also* files
 - FileRec records, 67
 - files
 - .DEF, 25, 158
 - .LIB, 24, 26, 157, 161, 163
 - .LNK, 25, 177
 - .LX, 25
 - .OBJ, 24, 161, 163
 - .RES, 26, 159
 - .VPI, 24, 152, 161, 165
 - import library, 26
 - module definition, 136–38
 - text, 66
 - typed, 65
 - untyped, 66
 - finalisation part of a unit, 20
 - fixed part of records, 41
 - floating point
 - numbers, 14
 - types, 32
 - internal representation, 34
 - for statement syntax, 98
 - formal
 - parameter list syntax, 120
 - formal parameters, 83, 94, 111, 112
 - forward
 - declarations, 113
 - directive, 111, 114
 - forward class declaration, 50
 - Free method, 120
 - FreeInstance method, 119
 - function
 - calls, 83
 - extended syntax and, 83
 - declarations, 112
 - assembler, 115
 - external, 115
 - inline, 116
 - headings, 112
 - results, 112, 113
 - functions, 105, 111. *See also* procedures and functions
 - built-in assembler, 105
 - far, 113
 - Length, 38
 - near, 113
 - nested, 69
 - recursive, 113
 - Result variable, 93
 - SetLength, 39
 - SetString, 39
 - SizeOf, 123
 - standard, 66
-
- G**
- GetMem procedure, 35, 79
 - global
 - compiler directives, 149
 - procedure pointers, 68
-

subsystem initialisation, 131
variables, 76
goto statement syntax, 94

H

Halt procedure, 20
hex digits, 11
hexadecimal
 constants, 14
 numbers, 14
High function, 28, 38, 123

I

identifier
 as labels, 15
 defined, 13
 qualified, 13
 scope of, 22
if statement syntax, 95
implementation part
 of a unit, 19
 syntax, 19
ImpLib utility, 129
Import library, 129
 creating, 129
 using with **\$L**, 157
importing symbols from a DLL, 128
 dynamically, 129
 statically, 128
IMPORTS statement, 143
in operator, 90, 91
Include procedure, 65
index
 clause, 127, 129, 130
 syntax, 78
 types valid in arrays, 40
index directive, 61
index specifiers, 61
indexing PChar pointers, 36
indices in arrays, 40
inherited reserved call, 119
inherited reserved word, 118
inherited statement, 56
initialisation
 part of a unit, 20
initialised variables, 73
inline
 declarations, 116
 directive, 116
 restrictions, 116
INPURE segment attribute, 139
Integer data type, 29
integer numbers, 13
integer types, 29

interface part
 of a unit, 19
 syntax, 19
interface part of a unit, 128
IOPL segment attribute, 138
is operator, 63, 91

L

label
 declaration part syntax, 21
 scope of, 22
 statement, 93
labels
 built-in assembler, 105
 defined, 15
Length function, 38
length of
 a program line, 9
 a string type value, 37
 character strings, 14
 identifiers, 13
letters, defined, 11
lexical elements, 9
libraries
 dynamic link, 127
library
 dynamic link, 20
LIBRARY statement, 144
Line Feed character, 9, 66
linking
 dynamic, 18
 executable files, 136
 object files, 157
 process, 25
 smart, 24
 static, 18
literals, defined, 13
LOADONCALL segment attribute, 138
local
 assembler labels, 105
 compiler directives, 149
 variables, 76
Location information, 26, 158
LongBool data type, 32
Longint data type, 30
Low function, 28, 123

M

MaxInt constant, 30
MaxLongint constant, 30
Mem array, 80
mem arrays syntax, 80
MemL array, 80
memory

allocation, 76
 MemW array, 80
 message directive, 56
 message dispatching, 57
 message handler
 implementations, 56
 message handler
 declarations, 56
 message ID, 56
 method
 activations, 57
 declarations, 116
 designator, 57
 pointer, 68
 pointers, 68
 methods
 abstract, 55
 class, 120
 constructors, 116
 defined, 43
 destructors, 116
 dynamic, 55
 message handler, 56
 static, 53
 virtual, 53
 MIXED1632 segment attribute, 139
 mod operator, 86
 modular programming, 18
 module definition file, 136
 MULTIPLE segment attribute, 139
 multiple-data DLLs. *See also* subroutine libraries

N

name clause, 127, 129, 130
 NAME statement, 145
 naming conventions, 187
 near declaration, 113
 New procedure, 35, 79
 nodefault directive, 62
 NOIOPL segment attribute, 138
 NONCONFORMING segment attribute, 138
 NONE segment attribute, 139
 NONSHARED segment attribute, 139
 not operator, 87
 null-terminated strings, 35
 NUL character, 35
 pointers and, 35
 number, 13–14
 numbers
 decimal, 13
 floating point, 14
 hexadecimal, 13
 integer, 14
 syntax, 14

O

object
 components, 47
 object and class scope, 23
 object files (.OBJ), 24
 linking with, 157
 object model
 new, 43
 old, 43
 old-style
 virtual method and, 53
 object models
 summary of, 64
 object reserved word, 43
 object type
 constants, 75
 object type compatibility, 51
 object type declarations, 43
 object types
 construction, 48
 domain, 48
 instances, 48
 objects
 ancestors, 48
 components, 43
 creating, 118
 descendants, 48
 destroying, 119
 inheritance, 48
 partially constructed, 120
 ODAPI, 166
 debugger DLL, 166
 Examples, 169
 IDE Interface, 167
 System Interface, 166
 types, 168
 Types, 168
 OLD statement, 145
 open array constructors, 124
 Open Debug API, ODAPI, 166
 open parameters, 123
 array, 123
 string, 40, 123, 160
 type variant, 124
 OpenString identifier, 39, 123, 160
 operands, 81
 built-in assembler, 107
 operators, 81
 @ (address-of), 84, 85
 @@ (double address-of), 85
 and, 87
 arithmetic, 86
 as, 63, 91
 bitwise logical, 86
 Boolean logical, 87
 class, 91
 div, 86

- is, 63, 91
- mod, 86
- not, 87
- or, 87
- PChar, 88
- relational, 89
- rules of precedence, 85
- set, 89
- shl, 87
- shr, 87
- string, 88
- types of, 85
- xor, 87

Optimising code

- code optimisation, 159, 163
- select processor, 155
- stack frames, 165

Ord function, 28, 31

ordinal number, 28, 31

ordinal types, 28

OS/2

- dynamic import, 129
- Presentation Manager, 129
- subsystems, 131
- Windows resource files, 159

OS2

- symbol, 17

overflow checking, 161

override directive, 53

P

packed

- reserved word, 37
- string type, 41

parameter

- formal, 94

parameter directives, 149

parameters, 120

- actual, 83, 94
- constant, 121
- formal, 83
- open, 123
 - array, 123
 - string, 123
- passing rules, 121
- untyped, 122
- value, 121
- variable, 122

partially constructed objects, 120

passing parameters

- by reference, 121
- by value, 121

passing string variables of varying sizes, 40

PChar

- data type, 35
- operators, 88

pointers

- indexing, 36
- string constant expression and, 36
- zero-based character arrays and, 41

pointer

- (^) symbol, 79
- types, 34
- values, 79
- variables, 79

Pointer data type, 35

pointers

- comparing, 90

Port array, 92

port arrays, 92

Port-arrays

- I/O privilege and, 138

PortL array, 92

PortW array, 92

Pred function, 28

PRELOAD segment attribute, 138

private

- components, 52
- identifiers, 19

procedural type

- constants, 76
- variables, 68

procedural types, 67

- type compatibility of, 68

procedure

- declaration syntax, 111
- declarations
 - assembler, 115
 - external, 115
 - forward, 113
 - inline, 116
- heading, 111
- statements, 94

procedure and function declaration part, 22

procedures, 111. *See also* procedures and functions

- built-in assembler, 105
- far, 113
- near, 113
- nested, 69

procedures and functions. *See also* procedures;

- functions
- importing, 128
- inline, 116
- nested, 69
- recursive, 111

program

- block, 21
- comments, 10
- lines, maximum length, 9
- structure, 18
- syntax, 18

properties, 47, 58

- array, 60
- defined, 43
- definitions, 58

- index specifiers, 61
- overrides, 63
- storage specifiers, 62
- read-only, 60
- read-write, 60
- write-only, 60
- protected
 - components, 52
- pseudo instructions, 106
- Ptr function, 35
- public
 - components, 52
 - identifiers, 19
- published
 - components, 52, 158

Q

- qualified
 - method
 - identifiers, 84
 - method
 - identifiers, 79
 - method activation, 58
 - method designator, 58
 - method identifier, 116
- qualifier syntax, 78

R

- raise statement syntax, 101
- READONLY segment attribute, 138
- READWRITE segment attribute, 138
- real
 - data type, 32
 - numbers, 32
- Real data type, 33, 34
- record
 - fields, 75, 79
 - scope, 23
 - types, 41
 - variant part declaration, 42
- records, 42, 75, 79
- record-type constant syntax, 75
- registers
 - AX, 113
 - EAX, 113
 - EBP, 109
 - EIP, 107
 - ESP, 109
- repeat statement syntax, 97
- repetitive statement syntax, 96
- reserved words, 12
 - ALIAS, 139
 - built-in assembler, 107
 - defined, 12

- list of, 12
- module definition file, 136
- resident option in exports clause, 130
- resource files
 - Open32 and, 159
- Result
 - variable, 93
- RET instruction, built-in assembler, 107
- returning
 - Char values, 31
 - the ordinal number of a value, 28
- RunError procedure, 20
- run-time error messages, 184
- Run-time type information, 158

S

- scope
 - block, 23
 - object and class scope, 23
 - of a component identifier, 117
 - of a label, 22
 - of an identifier, 22
 - rules of, 22
 - unit, 23
- segment
 - attributes, 138
- SEGMENTS statement, 138, 146
- Self identifier, 117, 120
- Self parameter, 50, 51, 57, 120
- set
 - constructor syntax, 83
 - membership testing, 91
 - operators, 89
 - types, 64
- sets. *See also* set
 - comparing, 91
- shl operator, 87
- short-circuit Boolean evaluation, 87
- ShortInt data type, 30
- shr operator, 87
- significand, 34
- simple
 - expression syntax, 81
 - types
 - comparing, 90
- simple-typed constants, 73
- Single data type, 34
- SINGLE segment attribute, 139
- single symbols, 12
- single-data DLLs. *See also* subsystems
- SmallInt data type, 30
- SmallWord data type, 30
- source file, 9
- space characters, 9
- stack
 - checking switch directive, 162

frames, built-in assembler, 110
 overflow, 77
 switch directive, 162
 passing parameters and, 121
 segment, 76
 DLLs and, 133
 size, 159
 STACKSIZE statements, 147
 standard
 directives, 12
 procedure or function used as a procedural value,
 69
 statement part syntax, 22
 statements, 93
 assembler, 104
 assignment, 93
 case, 96
 compound, 95
 conditional, 95
 exception, 100
 for, 98
 goto, 94
 if, 95
 labels, 93
 procedure, 94
 raise, 101
 repeat, 97
 repetitive, 96
 try...except, 102
 try...finally, 103
 while, 97
 with, 99
 static
 importing, 128
 linking, 18
 methods, 53
 object variables, 49
 storage specifiers, 62
 stored directive, 62
 storing null-terminated strings, 41
 strict string parameter checking, 165
 string
 operator, 88
 type, 37
 typed constants, 73
 variables, 78
 string variables passing, 39
 strings. *See also* string
 comparing, 39, 90
 concatenating, 39, 88
 long, 38. *See also* types:AnsiString
 null-terminated, 35
 short, 37
 strict parameter checking of, 164
 whitespaces and, 9
 structured types, 37
 STUB statements, 147
 subrange type, 32
 subroutine libraries, 132

subsystems, 131
 Succ function, 28
 summary of the two object models, 64
 switch compiler directives, 149
 symbol
 local information, 152
 name, 137
 pairs, 12
 symbols, 12, 15
 built-in assembler, 108
 conditional, 17
 syntax
 extended, 165
 System unit, 18, 21, 29, 133
 SysUtils unit, 30, 100

T

tag field in a record, 42
 term
 syntax, 81
 testing set membership, 91
 text files, 66
 Text type, 66
 TFileRec records, 67
 thread local storage, 77
threadvar declarations, 77
 TObject class declaration, 48
 tokens, 9
 trapping I/O errors, 155
 True boolean constant, 31
 try...except statement syntax, 102
 try...finally statement syntax, 103
 TTextRec records, 67
 TVarRec type, 125
 type. *See also* types
 compatibility, 70
 declaration, 28
 declaration part syntax, 22
 identity, 69
 Type variant open array parameters, 124
 typecasts
 of untyped parameter, 122
 value, 84
 variable, 79
 typed
 constant declaration, 73
 files, 65
 types, 28
 AnsiString, 37, 38, 155
 array, 40
 boolean, 31
 Boolean, 32
 Byte, 30
 Cardinal, 30
 Char, 31
 class-reference, 63

Comp, 33
 compatible, 70
 Currency, 33
 Double, 33
 enumerated, 31
 Extended, 33
 file, 65
 floating point, 32
 internal representation, 34
 identical, 69
 integer, 29
 predefined, 29
 size and range, 29
Integer, 29
 LongBool, 32
 Longint, 30
 object and class, 43
 ordinal, 28
 packed string, 41
 PChar, 35
 Pointer, 34
 procedural, 67
 Real, 33
 record, 41
 set, 64
 ShortInt, 30
 ShortString, 155
 Single, 33
 SmallInt, 30
 SmallWord, 30
 string, 37
 structured, 37
 subrange, 32
 TVarRec, 125
 Word, 29
 WordBool, 32

U

ultimate ancestor, 48
 unary operators, 81
 unit
 scope, 23
 syntax, 18
 units, 18
 identifier, 19
 implementation part, 19
 initialisation and finalisation parts, 20
 interface part, 19
 uses clause, 21
 untyped
 constants, 27
 files, 66
 parameters, 122
 USE32
 symbol, 17
 Use32 unit, 30, 164

uses clause, 21
 Utilities
 ImpLib, 129

V

value
 parameters, 121
 typecast syntax, 84
 var
 parameters, 122
 variable. *See also* variables
 declaration part syntax, 22
 declaration syntax, 72
 parameters, 122
 reference
 qualifier, 77
 syntax, 77
 typecast declaration, 79
 variables
 absolute, 73
 array-type, 78
 class, 79
 class reference, 79
 class type, 49
 dynamic, 79
 global, 76
 local, 76
 object, 79
 pointer, 79
 record, 79
 string-type, 78
 with an initial value, 73
 variant part of records, 42
 VER21 symbol, 17
 virtual
 constructor, 118
 directive, 53
 directive, 55
 methods, 53
 Virtual Method Table, 50
 VIRTUALPASCAL symbol, 17
 VpDbgDll, 166
 VpSysLow unit, 129

W

while statement syntax, 97
 whitespaces, 9
 Win32
 dynamic import, 130
 symbol, 17
 with statement syntax, 99
 with statements, 51, 58, 117
 Word data type, 29
 WordBool data type, 32

X

xor operator, 87

Z

zero-based character arrays, 35, 36, 41