Free Pascal
Reference guide

Reference guide for Free Pascal, version 3.2.4-rc2
Document version 3.2.4
October 2026

Michaël Van Canneyt

Contents

1 Pascal Tokens **12**

1.1 Symbols . . . 12
1.2 Comments . . . 13
1.3 Reserved words . . . 14
1.3.1 Turbo Pascal reserved words . . . 14
1.3.2 Object Pascal reserved words . . . 15
1.3.3 Modifiers . . . 15
1.4 Identifiers . . . 16
1.5 Hint directives . . . 17
1.6 Numbers . . . 18
1.7 Labels . . . 19
1.8 Character strings . . . 19

2 Constants **21**

2.1 Ordinary constants . . . 21
2.2 Typed constants . . . 22
2.3 Resource strings . . . 23

3 Types **25**

3.1 Base types . . . 26
3.1.1 Ordinal types . . . 27
Integers . . . 28
Boolean types . . . 29
Enumeration types . . . 30
Subrange types . . . 32
Character types . . . 33
3.1.2 Floating-point types . . . 33
3.2 Character types . . . 33
3.2.1 Char or AnsiChar . . . 33
3.2.2 WideChar . . . 34
3.2.3 Other character types . . . 34

3.2.4 Single-byte String types . . . 34
Short strings . . . 35
Ansistrings . . . 36
Code page conversions . . . 38
RawByteString . . . 40
UTF8String . . . 40
3.2.5 Multi-byte String types . . . 41
UnicodeStrings . . . 41
WideStrings . . . 41
3.2.6 Constant strings . . . 42
3.2.7 PChar – Null terminated strings . . . 42
3.2.8 String sizes . . . 44
3.3 Structured Types . . . 45
Packed structured types . . . 45
3.3.1 Arrays . . . 46
Static arrays . . . 47
Dynamic arrays . . . 48
Array constants . . . 51
Dynamic array Type compatibility . . . 52
Dynamic array constructor . . . 53
Dynamic array constant expressions . . . 55
Packing and unpacking an array . . . 55
3.3.2 Record types . . . 55
Record-typed Constant . . . 57
Record layout and size . . . 57
Remarks and examples . . . 58
3.3.3 Set types . . . 59
3.3.4 File types . . . 60
3.4 Pointers . . . 60
3.5 Forward type declarations . . . 62
3.6 Procedural types . . . 63
3.7 Variant types . . . 67
3.7.1 Definition . . . 67
3.7.2 Variants in assignments and expressions . . . 68
3.7.3 Variants and interfaces . . . 69
3.8 Type aliases . . . 70
3.9 Managed types . . . 71

4 Variables **73**
4.1 Definition . . . 73

4.2 Declaration . . . 73
4.3 Scope . . . 75
4.4 Initialized variables . . . 75
4.5 Initializing variables using default . . . 77
4.6 Thread Variables . . . 78
4.7 Properties . . . 79

5 Objects **81**
5.1 Declaration . . . 81
5.2 Abstract and sealed objects . . . 82
5.3 Fields . . . 83
5.4 Class or Static fields . . . 84
5.5 Constructors and destructors . . . 85
5.6 Methods . . . 86
5.6.1 Declaration . . . 86
5.6.2 Method invocation . . . 87
Normal static methods . . . 87
Virtual methods . . . 87
Abstract methods . . . 89
Class or static methods . . . 90
5.7 Visibility . . . 91

6 Classes **92**
6.1 Class definitions . . . 92
6.2 Abstract and sealed classes . . . 96
6.3 Normal and static fields . . . 96
6.3.1 Normal fields/variables . . . 96
6.3.2 Class fields/variables . . . 97
6.4 Class instantiation: Creating a class instance . . . 99
6.5 Class destruction : Disposing of a class instance . . . 100
6.6 Methods . . . 101
6.6.1 Declaration . . . 101
6.6.2 Implementation . . . 101
6.6.3 Invocation . . . 102
6.6.4 Virtual methods . . . 102
6.6.5 Final methods . . . 103
6.6.6 Class methods . . . 103
6.6.7 Class type initialization . . . 104
6.6.8 Static class methods . . . 106
6.6.9 Covariant return type . . . 108
6.6.10 Message methods . . . 108

6.6.11 Using inherited . . . 109
6.7 Method overloading . . . 110
6.8 Properties . . . 111
6.8.1 Definition . . . 111
6.8.2 Indexed properties . . . 113
6.8.3 Array properties . . . 114
6.8.4 Default properties . . . 115
6.8.5 Published properties . . . 115
6.8.6 Storage information . . . 116
6.8.7 Overriding and redeclaring properties . . . 117
6.9 Class properties . . . 119
6.10 Nested types, constants and variables . . . 119

7 Interfaces **122**
7.1 Definition . . . 122
7.2 Interface identification: A GUID . . . 123
7.3 Interface implementations . . . 124
7.4 Interface inheritance . . . 125
7.5 Interface delegation . . . 126
7.6 Interfaces and COM . . . 129
7.7 Dispatch Interfaces . . . 129
7.8 CORBA and other Interfaces . . . 130
7.9 Reference counting . . . 131

8 Generics **132**
8.1 Introduction . . . 132
8.2 Generic type definition . . . 132
8.3 Generic type specialization . . . 135
8.4 Generic type restrictions . . . 137
8.5 Generic procedures and functions . . . 138
8.6 Delphi compatibility . . . 140
8.6.1 Syntax elements . . . 140
8.6.2 Record type restrictions . . . 141
8.6.3 Type overloads . . . 142
8.6.4 Name space considerations . . . 142
8.7 Type compatibility . . . 143
8.8 Using the default intrinsic . . . 145
8.9 A word about scope . . . 146
8.10 Operator overloading and generics . . . 148

9 Extended records **151**

9.1 Definition . . . 151
9.2 Extended record enumerators . . . 153
9.3 Record operators . . . 155

10 Class, Record and Type helpers **158**
10.1 Definition . . . 158
10.2 Restrictions on class helpers . . . 159
10.3 Restrictions on record helpers . . . 160
10.4 Considerations for (simple) type helpers . . . 161
10.5 A note on scope and lifetime for record and type helpers . . . 162
10.6 Inheritance . . . 164
10.7 Usage . . . 165

11 Objective-Pascal Classes **168**
11.1 Introduction . . . 168
11.2 Objective-Pascal class declarations . . . 168
11.3 Formal declaration . . . 170
11.4 Allocating and de-allocating Instances . . . 172
11.5 Protocol definitions . . . 173
11.6 Categories . . . 174
11.7 Name scope and Identifiers . . . 175
11.8 Selectors . . . 176
11.9 The `id` type . . . 176
11.10 Enumeration in Objective-C classes . . . 177

12 Expressions **178**
12.1 Expression syntax . . . 179
12.2 Function calls . . . 180
12.3 Set constructors . . . 183
12.4 Array constructors . . . 184
12.5 Value typecasts . . . 184
12.6 Variable typecasts . . . 185
12.7 Unaligned typecasts . . . 186
12.8 The @ operator . . . 186
12.9 Operators . . . 187
12.9.1 Arithmetic operators . . . 187
12.9.2 Logical operators . . . 188
12.9.3 Boolean operators . . . 189
12.9.4 String operators . . . 189
12.9.5 Dynamic array operators . . . 190
12.9.6 Set operators . . . 190

12.9.7 Relational operators . . . 192
12.9.8 Class operators . . . 193

13 Statements **196**
13.1 Simple statements . . . 196
13.1.1 Assignments . . . 196
13.1.2 Procedure statements . . . 198
13.1.3 Goto statements . . . 198
13.2 Structured statements . . . 199
13.2.1 Compound statements . . . 200
13.2.2 The `Case` statement . . . 200
13.2.3 The `If..then..else` statement . . . 202
13.2.4 The `For..to/downto..do` statement . . . 204
13.2.5 The `For..in..do` statement . . . 206
13.2.6 The `Repeat..until` statement . . . 214
13.2.7 The `While..do` statement . . . 214
13.2.8 The `With` statement . . . 215
13.2.9 Exception Statements . . . 217
13.3 Assembler statements . . . 217

14 Using functions and procedures **219**
14.1 Procedure declaration . . . 219
14.2 Function declaration . . . 220
14.3 Function results . . . 220
14.4 Parameter lists . . . 221
14.4.1 Value parameters . . . 222
14.4.2 Variable parameters . . . 223
14.4.3 Out parameters . . . 223
14.4.4 Constant parameters . . . 224
14.4.5 Open array parameters . . . 226
14.4.6 Macpas: UNIV typed parameters . . . 226
14.4.7 Array of const . . . 227
14.4.8 Untyped parameters . . . 229
14.4.9 Ellipsis parameter . . . 230
14.4.10 Managed types and reference counts . . . 230
14.5 Function overloading . . . 234
14.6 Forward declared functions . . . 234
14.7 External functions . . . 235
14.8 Assembler functions . . . 236
14.9 Modifiers . . . 237
14.9.1 alias . . . 237

14.9.2 cdecl . . . 238
14.9.3 compilerproc . . . 239
14.9.4 cppdecl . . . 239
14.9.5 export . . . 239
14.9.6 hardfloat . . . 239
14.9.7 inline . . . 239
14.9.8 interrupt . . . 240
14.9.9 iocheck . . . 240
14.9.10 local . . . 240
14.9.11 MS_ABI_Default . . . 240
14.9.12 MS_ABI_CDecl . . . 240
14.9.13 MWPascal . . . 241
14.9.14 noreturn . . . 241
14.9.15 nostackframe . . . 241
14.9.16 overload . . . 241
14.9.17 pascal . . . 243
14.9.18 public . . . 243
14.9.19 register . . . 243
14.9.20 safecall . . . 244
14.9.21 saveregisters . . . 244
14.9.22 softfloat . . . 244
14.9.23 stdcall . . . 244
14.9.24 SYSV_ABI_Default . . . 244
14.9.25 SYSV_ABI_CDecl . . . 244
14.9.26 VectorCall . . . 244
14.9.27 varargs . . . 245
14.9.28 winapi . . . 245
14.10 Unsupported Turbo Pascal modifiers . . . 245

15 Operator overloading **246**
15.1 Introduction . . . 246
15.2 Operator declarations . . . 246
15.3 Assignment operators . . . 248
15.4 Arithmetic operators . . . 252
15.5 Comparison operator . . . 253
15.6 In operator . . . 255
15.7 Logical operators . . . 255
15.8 Inc/Dec operators . . . 257
15.9 The enumerator operator . . . 257

16 Programs, units, blocks **259**

16.1 Programs . . . 259
16.2 Units . . . 261
16.3 Namespaces: Dotted Units . . . 263
16.4 Unit dependencies . . . 265
16.5 Blocks . . . 266
16.6 Scope . . . 267
16.6.1 Block scope . . . 268
16.6.2 Record scope . . . 268
16.6.3 Class scope . . . 268
16.6.4 Unit scope . . . 268
16.7 Libraries . . . 270

17 Exceptions **272**
17.1 The raise statement . . . 272
17.2 The try...except statement . . . 274
17.3 The try...finally statement . . . 275
17.4 Exception handling nesting . . . 276
17.5 Exception classes . . . 276

18 Using assembler **278**
18.1 Assembler statements . . . 278
18.2 Assembler procedures and functions . . . 278

A Pascal grammar **279**

List of Tables

3.1 Predefined integer types 28
3.2 Predefined integer ranges 28
3.3 Integer constant type mapping 29
3.4 Boolean types 30
3.5 Supported floating-point types 33
3.6 `PChar` pointer arithmetic 44
3.7 String memory sizes 45

9.1 Operator names 156

12.1 Precedence of operators 178
12.2 Binary arithmetic operators 187
12.3 Unary arithmetic operators 188
12.4 Logical operators 188
12.5 Boolean operators 189
12.6 Set operators 191
12.7 Relational operators 192
12.8 Class operators 193

13.1 Allowed C constructs in Free Pascal 197

14.1 Unsupported modifiers 245

About this guide

This document serves as the reference for the Pascal language as implemented by the Free Pascal compiler. It describes all Pascal constructs supported by Free Pascal, and lists all supported data types. It does not, however, give a detailed explanation of the Pascal language: it is not a tutorial. The aim is to list which Pascal constructs are supported, and to show where the Free Pascal implementation differs from the Turbo Pascal or Delphi implementations.

The Turbo Pascal and Delphi Pascal compilers introduced various features in the Pascal language. The Free Pascal compiler emulates these compilers in the appropriate mode of the compiler: certain features are available only if the compiler is switched to the appropriate mode. When required for a certain feature, the use of the `-M` command-line switch or `{$MODE }` directive will be indicated in the text. More information about the various modes can be found in the User's Guide and the Programmer's Guide.

Earlier versions of this document also contained the reference documentation of the system unit and objpas unit. This has been moved to the Unit Reference.

Notations

Throughout this document, we will refer to functions, types and variables with `typewriter` font. Files are referred to with a sans font: filename.

Syntax diagrams

All elements of the Pascal language are explained in syntax diagrams. Syntax diagrams are like flow charts. Reading a syntax diagram means getting from the left side to the right side, following the arrows. When the right side of a syntax diagram is reached, and it ends with a single arrow, this means the syntax diagram is continued on the next line. If the line ends on two arrows pointing to each other, then the diagram is ended.

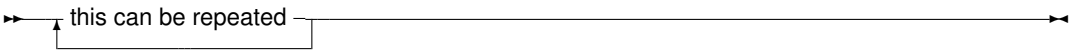
Syntactical elements are written like this



Keywords which must be typed exactly as in the diagram:



When something can be repeated, there is an arrow around it:



When there are different possibilities, they are listed in rows:



Note, that one of the possibilities can be empty:



This means that both the first or second possibility are optional. Of course, all these elements can be combined and nested.

For some syntactical elements you will find no definition in the various diagrams. This is because they are defined implicitly in the pascal parser: the tokens of the pascal language.

The following elements are defined implicitly:

string This is a literal string.

number This can be an integer number or a floating point number.

identifier This is basically a name the programmer has given to an item he wishes to use in his code. This item can be a type, a variable, a constant or even the name of a procedure or unit.

Comments or whitespace will not appear in the syntax diagrams.

About the Pascal language

The language Pascal was originally designed by Niklaus Wirth around 1970. It has evolved significantly since that day, with a lot of contributions by the various compiler constructors (Notably: Borland). The basic elements have been kept throughout the years:

- Easy syntax, rather verbose, yet easy to read. Ideal for teaching.
- Strongly typed.
- Procedural.
- Case insensitive.
- Allows nested procedures.
- Easy input/output routines built-in.

The Turbo Pascal and Delphi Pascal compilers introduced various features in the Pascal language, most notably easier string handling and object orientedness. The Free Pascal compiler initially emulated most of Turbo Pascal and later on Delphi. It emulates these compilers in the appropriate mode of the compiler: certain features are available only if the compiler is switched to the appropriate mode. When required for a certain feature, the use of the `-M` command-line switch or `{$MODE }` directive will be indicated in the text. More information about the various modes can be found in the User's Guide and Programmer's Guide.

Chapter 1

Pascal Tokens

Tokens are the basic lexical building blocks of source code: they are the "words" of the language: characters are combined into tokens according to the rules of the programming language. There are five classes of tokens:

reserved words These are words which have a fixed meaning in the language. They cannot be changed or redefined.

identifiers These are names of symbols that the programmer defines. They can be changed and re-used. They are subject to the scope rules of the language.

operators These are usually symbols for mathematical or other operations: +, -, * and so on.

separators This is usually white-space.

constants Numerical or character constants are used to denote actual values in the source code, such as 1 (integer constant) or 2.3 (float constant) or "String constant" (a string: a piece of text).

In this chapter we describe all the Pascal reserved words, as well as the various ways to denote strings, numbers, identifiers etc.

1.1 Symbols

Free Pascal allows all characters, digits and some special character symbols in a Pascal source file.



The following characters have a special meaning:

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

and the following character pairs too:

```
<< >> ** <> >< <= >= := += -= *= /= (* *) (. .) //
```

When used in a range specifier, the character pair `(.` is equivalent to the left square bracket `[`. Likewise, the character pair `.)` is equivalent to the right square bracket `]`. Moreover, a specifier can combine different forms of left and right parentheses. When used for comment delimiters, the character pair `(*` is equivalent to the left brace `{` and the character pair `*)` is equivalent to the right brace `}`. These character pairs retain their normal meaning in string expressions.

1.2 Comments

Comments are pieces of the source code which are completely discarded by the compiler. They exist only for the benefit of the programmer, so he can explain certain pieces of code. For the compiler, it is as if the comments were not present.

The following piece of code demonstrates a comment:

```
(* My beautiful function returns an interesting result *)
Function Beautiful : Integer;
```

This is an older form of comment notation, and nowadays the `{` and `}` characters are the standard notation:

```
{ My beautiful function returns an interesting result }
Function Beautiful : Integer;
```

The comment can also span multiple lines:

```
{
   My beautiful function returns an interesting result,
   but only if the argument A is less than B.
}
Function Beautiful (A,B : Integer): Integer;
```

Single line comments can also be made with the `//` delimiter:

```
// My beautiful function returns an interesting result
Function Beautiful : Integer;
```

The comment extends from the // character till the end of the line. This kind of comment was introduced by Borland in the Delphi Pascal compiler.

Free Pascal supports the use of nested comments. The following constructs are valid comments:

```
(* This is an old style comment *)
{  This is a Turbo Pascal comment }
// This is a Delphi comment. All is ignored till the end of the line.
```

The following are valid ways of nesting comments:

```
{ Comment 1 (* comment 2 *) }
(* Comment 1 { comment 2 } *)
{ comment 1 // Comment 2 }
(* comment 1 // Comment 2 *)
// comment 1 (* comment 2 *)
// comment 1 { comment 2 }
```

The last two comments *must* be on one line. The following two will give errors:

```
 // Valid comment { No longer valid comment !!
    }
```

and

```
 // Valid comment (* No longer valid comment !!
    *)
```

The compiler will react with a "invalid character" error when it encounters such constructs, regardless of the mode and feature switches.

Remark For compatibility reasons, nested comments are disabled by default in some modes (e.g. `TP` and `Delphi`). You can change this using the `MODESWITCH` directive with the value `NESTEDCOMMENTS`.

1.3 Reserved words

Reserved words are part of the Pascal language, and as such, cannot be redefined by the programmer. Throughout the syntax diagrams they will be denoted using a **bold** typeface. Pascal is not case sensitive so the compiler will accept any combination of upper or lower case letters for reserved words.

We make a distinction between Turbo Pascal and Delphi reserved words. In `TP` mode, only the Turbo Pascal reserved words are recognized, but the Delphi ones can be redefined. By default, Free Pascal recognizes the Delphi reserved words.

1.3.1 Turbo Pascal reserved words

The following keywords exist in Turbo Pascal mode

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

1.3.2 Object Pascal reserved words

The reserved words of Object Pascal (used in `Delphi` or `Objfpc` mode) are the same as the Turbo Pascal ones, with the following additional keywords:

```
as              finalization    library         raise
class           finally         on              resourcestring
dispinterface   initialization  out             threadvar
except          inline          packed          try
exports         is              property
```

1.3.3 Modifiers

The following is a list of all modifiers. They are not exactly reserved words in the sense that they can be used as identifiers, but in specific places, they have a special meaning for the compiler, i. e., the compiler considers them as part of the Pascal language.

```
absolute        external        noinline        safecall
abstract        far             noreturn        saveregisters
alias           far16           nostackframe    sealed
assembler       forward         oldfpccall      softfloat
bitpacked       generic         otherwise       specialize
break           helper          overload        static
cdecl           implements      override        stdcall
compilerproc    index           pascal          stored
continue        inline          platform        strict
cppdecl         interrupt       private         unaligned
cvar            iocheck         protected       unimplemented
default         local           public          varargs
deprecated      message         published       vectorcall
dynamic         mwpascal        read            virtual
enumerator      name            register        weakexternal
experimental    near            reintroduce     winapi
export          nodefault       result          write
```

Remark Predefined types such as `Byte`, `Boolean` and constants such as `maxint` are *not* reserved words. They are identifiers, declared in the system unit. This means that these types can be redefined in other units. The programmer is however not encouraged to do this, as it will cause a lot of confusion.

Remark As of version 2.5.1 it is possible to use reserved words as identifiers by escaping them with a & sign. This means that the following is possible

```
var
  &var : integer;

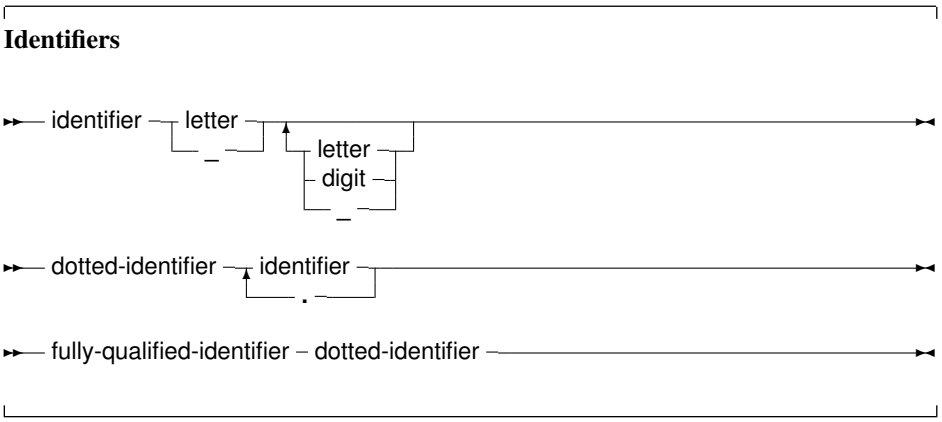
begin
  &var:=1;
  Writeln(&var);
end.
```

however, it is not recommended to use this feature in new code, as it makes code less readable. It is mainly intended to fix old code when the list of reserved words changes and encompasses a word that was not yet reserved (See also section 1.4, page 16).

1.4 Identifiers

Identifiers denote programmer defined names for specific constants, types, variables, procedures and functions, units, and programs. All programmer defined names in the source code – excluding reserved words – are designated as identifiers.

Identifiers consist of between 1 and 127 significant characters (letters, digits and the underscore character), of which the first must be a letter (a–z or A–Z), or an underscore (_). The following diagram gives the basic syntax for identifiers.



Dotted identifiers are an identifier which can contain a dot in the name. Fully qualified identifiers are dotted identifiers with the implied meaning that the first part is a unit name, and the second part is the name of an identifier in that unit. Both the unit identifier and the latter can of course also be dotted identifiers in themselves.

Like Pascal reserved words, identifiers are case insensitive, that is, both

```
myprocedure;
```

and

```
MyProcedure;
```

refer to the same procedure.

Remark As of version 2.5.1 it is possible to specify a reserved word as an identifier by prepending it with an ampersand (&). This means that the following is possible:

```
program testdo;

procedure &do;

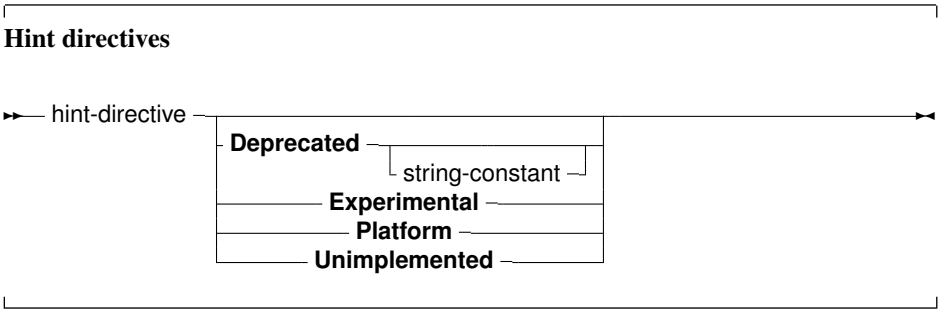
begin
end;

begin
  &do;
end.
```

The reserved word `do` is used as an identifier for the declaration as well as the invocation of the procedure `do`.

1.5 Hint directives

Most identifiers (constants, variables, functions or methods, properties) can have a hint directive appended to their definition:



Whenever an identifier marked with a hint directive is later encountered by the compiler, then a warning will be displayed, corresponding to the specified hint.

deprecated The use of this identifier is deprecated, use an alternative instead. The deprecated keyword can be followed by a string constant with a message. The compiler will show this message whenever the identifier is encountered.

experimental The use of this identifier is experimental: this can be used to flag new features that should be used with caution.

platform This is a platform-dependent identifier: it may not be defined on all platforms.

unimplemented This should be used on functions and procedures only. It should be used to signal that a particular feature has not yet been implemented.

The following are examples:

```
Const
  AConst = 12 deprecated;

var
  p : integer platform;

Function Something : Integer; experimental;

begin
  Something:=P+AConst;
end;

begin
  Something;
end.
```

This would result in the following output:

```
testhd.pp(11,15) Warning: Symbol "p" is not portable
testhd.pp(11,22) Warning: Symbol "AConst" is deprecated
testhd.pp(15,3) Warning: Symbol "Something" is experimental
```

Hint directives can follow all kinds of identifiers: units, constants, types, variables, functions, procedures and methods.

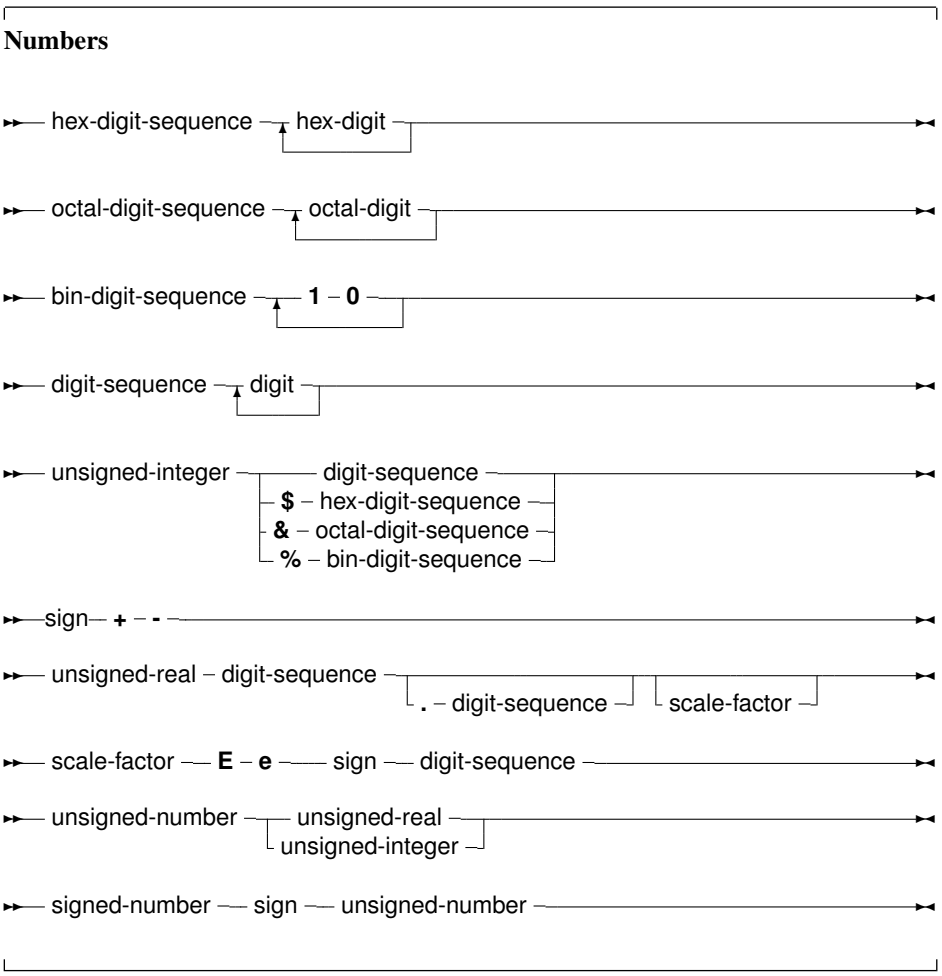
1.6 Numbers

Numbers are by default denoted in decimal notation. Real (or decimal) numbers are written using engineering or scientific notation (e. g. `0.314E1`).

For integer type constants, Free Pascal supports four formats:

1. Normal, decimal format (base 10). This is the standard format.
2. Hexadecimal format (base 16), in the same way as Turbo Pascal does. To specify a constant value in hexadecimal format, prepend it with a dollar sign (`$`). Thus, the hexadecimal `$FF` equals 255 decimal. Note that case is insignificant when using hexadecimal constants.
3. As of version 1.0.7, Octal format (base 8) is also supported. To specify a constant in octal format, prepend it with an ampersand (&). For instance 15 is specified in octal notation as `&17`.
4. Binary notation (base 2). A binary number can be specified by preceding it with a percent sign (`%`). Thus, `255` can be specified in binary notation as `%11111111`.

The following diagrams show the syntax for numbers.



Remark Octal and Binary notation are not supported in `TP` or `Delphi` compatibility mode.

1.7 Labels

A label is a name for a location in the source code to which can be jumped to from another location with a `goto` statement. A Label is a standard identifier or a digit sequence.



Remark The `-Sg` or `-Mtp` switches must be specified before labels can be used. By default, Free Pascal doesn't support `label` and `goto` statements. The `{$GOTO ON}` directive can also be used to allow use of labels and the goto statement.

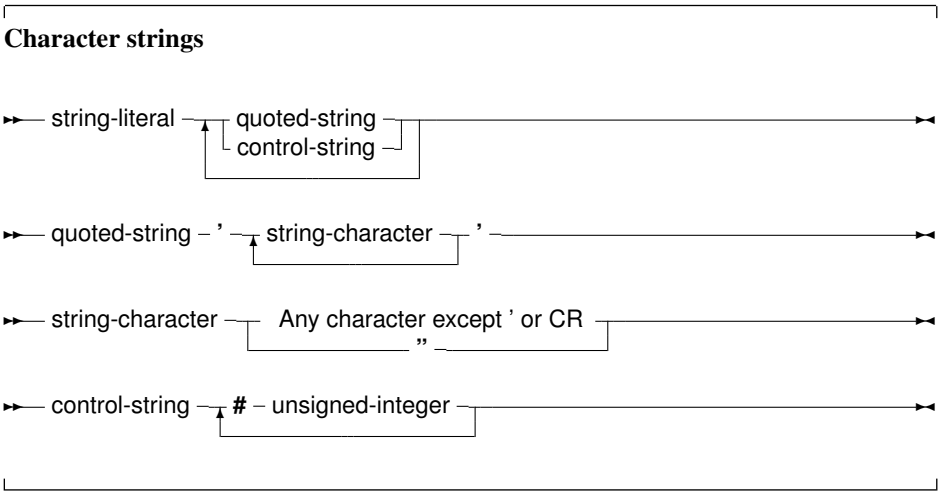
The following are examples of valid labels:

```
Label
  123,
  abc;
```

1.8 Character strings

A character string or string literal (string for short) is a sequence of zero or more characters (byte or word sized), enclosed in single quotes, and on a single line of the program source code: no literal carriage return or linefeed characters can appear in the string.

A character set with nothing between the quotes (`''`) is an empty string.



The string consists of standard, 8-bit ASCII characters or Unicode (normally UTF-8 encoded) characters. The `control string` can be used to specify characters which cannot be typed on a keyboard, such as `#27` for the escape character.

The single quote character can be embedded in the string by typing it twice. The C construct of escaping characters in the string (using a backslash) is not supported in Pascal.

The following are valid string constants:

```
'This is a pascal string'
''
'a'
'A tabulator character: '#9' is easy to embed'
```

The following is an invalid string:

```
'the string starts here
 and continues here'
```

The above string must be typed as:

```
'the string starts here'#13#10'   and continues here'
```

or

```
'the string starts here'#10'   and continues here'
```

on unices (including Mac OS X), and as

```
'the string starts here'#13'   and continues here'
```

on a classic Mac-like operating system.

It is possible to use other character sets in strings: in that case the codepage of the source file must be specified with the `{$CODEPAGE XXX}` directive or with the `-Fc` command line option for the compiler. In that case the characters in a string will be interpreted as characters from the specified codepage.

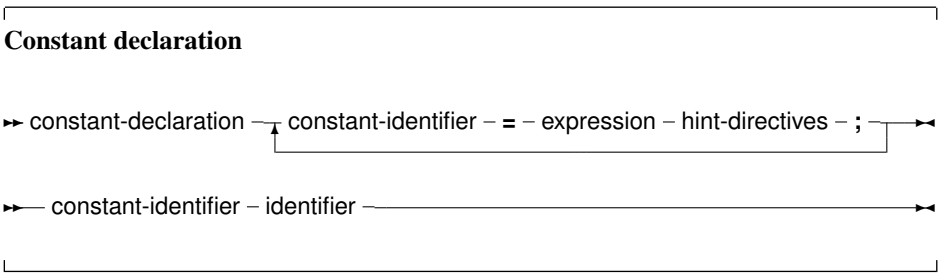
Chapter 2

Constants

Just as in Turbo Pascal, Free Pascal supports both ordinary and typed constants. They are declared in a constant declaration block in a unit, program or class, function or procedure declaration (section 16.5, page 266).

2.1 Ordinary constants

Ordinary constants declarations are constructed using an identifier name followed by an "=" token, and followed by an optional expression consisting of legal combinations of numbers, characters, boolean values or enumerated values as appropriate. The following syntax diagram shows how to construct a legal declaration of an ordinary constant.



The compiler must be able to evaluate the expression in a constant declaration at compile time. This means that most of the functions in the Run-Time library cannot be used in a constant declaration. Operators such as `+`, `-`, `*`, `/`, `not`, `and`, `or`, `div`, `mod`, `ord`, `chr`, `sizeof`, `pi`, `int`, `trunc`, `round`, `frac`, `odd` can be used, however. For more information on expressions, see chapter 12, page 178.

As with all identifiers, constants can be accompagnied by hint directives (section 1.5, page 17):

```
const
  MyPi = 3.14 deprecated;
```

When a previously declared ordinary constant is used in the code, the compiler will insert the actual value of the constant instead of the constant name. That is, the following two pieces of code are entirely equivalent:

```
Const
```

```
  One = 1;

begin
  Writeln(One);
end.
```

The above will produce the same code as if one had written:

```
begin
  Writeln(1);
end.
```

Only constants of the following types can be declared:

- Ordinal types
- Set types
- Pointer types (but the only allowed value is `Nil`).
- Real types
- `Char`,
- `String`

The following are all valid constant declarations:

```
Const
  e  = 2.7182818;  { Real type constant. }
  a  = 2;          { Ordinal (Integer) type constant. }
  c  = '4';        { Character type constant. }
  s  = 'This is a constant string'; {String type constant.}
  sc = chr(32)
  ls = SizeOf(Longint);
  P  = Nil;
  Ss = [1,2];
```

Assigning a value to an ordinary constant is not permitted. Thus, given the previous declaration, the following will result in a compiler error:

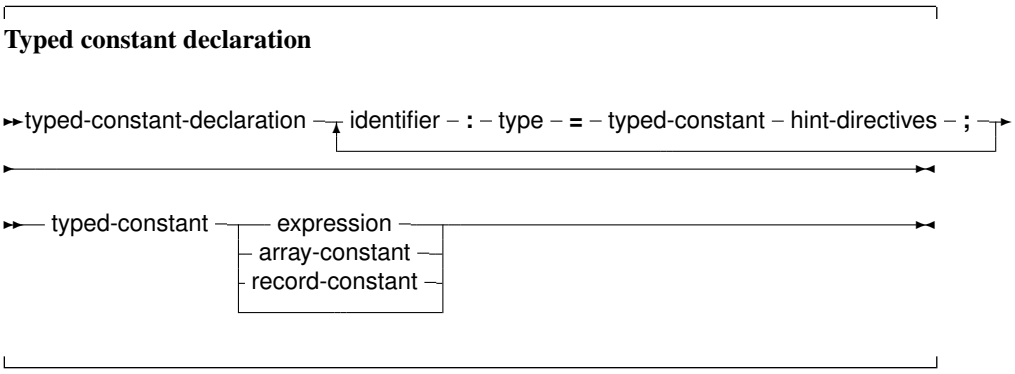
```
  s := 'some other string';
```

For string constants, the type of the string is dependent on some compiler switches. If a specific type is desired, a typed constant should be used, as explained in the following section.

Prior to version 1.9, Free Pascal did not correctly support 64-bit constants. As of version 1.9, 64-bit constants can be specified.

2.2 Typed constants

Sometimes it is necessary to specify the type of a constant, for instance for constants of complex structures (defined later in the manual). Their definition is quite simple.



In the case of an expression, the expression must be computable at runtime, just as for a normal constant, and the type of the expression must be assignment compatible to the type of the constant.

Contrary to ordinary constants, a value can be assigned to them at run-time. This is an old concept from Turbo Pascal, which has been replaced with support for initialized variables: For a detailed description, see section 4.4, page 75.

Support for assigning values to typed constants is controlled by the `{$J}` directive: it can be switched off, but is on by default (for Turbo Pascal compatibility). Initialized variables are always allowed.

Remark It should be stressed that typed constants are automatically initialized at program start. This is also true for *local* typed constants and initialized variables. Local typed constants are also initialized at program start. If their value was changed during previous invocations of the function, they will retain their changed value, i. e. they are not initialized each time the function is invoked.

2.3 Resource strings

A special kind of constant declaration block is the `Resourcestring` block. Resourcestring declarations are much like constant string declarations: resource strings act as constant strings, but they can be localized by means of a set of special routines in the objpas unit. A resource string declaration block is only allowed in the `Delphi` or `Objfpc` modes.

The following is an example of a resourcestring definition:

```
Resourcestring

  FileMenu = '&File...';
  EditMenu = '&Edit...';
```

All string constants defined in the resourcestring section are stored in special tables. The strings in these tables can be manipulated at runtime with some special mechanisms in the objpas unit.

Semantically, the strings act like ordinary constants; It is not allowed to assign values to them (except through the special mechanisms in the objpas unit). However, they can be used in assignments or expressions as ordinary string constants. The main use of the resourcestring section is to provide an easy means of internationalization.

More on the subject of resourcestrings can be found in the Programmer's Guide, and in the objpas unit reference.

Remark Note that a resource string which is given as an expression will not change if the parts of the expression are changed:

```
resourcestring
```

```
  Part1 = 'First part of a long string.';
  Part2 = 'Second part of a long string.';
  Sentence = Part1+' '+Part2;
```

If the localization routines translate `Part1` and `Part2`, the `Sentence` constant will not be translated automatically: it has a separate entry in the resource string tables, and must therefore be translated separately. The above construct simply says that the initial value of `Sentence` equals `Part1+' '+Part2`.

Remark Likewise, when using resource strings in a constant array, only the initial values of the resource strings will be used in the array: when the individual constants are translated, the elements in the array will retain their original value.

```
resourcestring
  Yes = 'Yes.';
  No = 'No.';

Var
  YesNo : Array[Boolean] of string = (No,Yes);
  B : Boolean;

begin
  Writeln(YesNo[B]);
end.
```

This will print "Yes." or "No." depending on the value of B, even if the constants Yes and No have been localized by some localization mechanism.

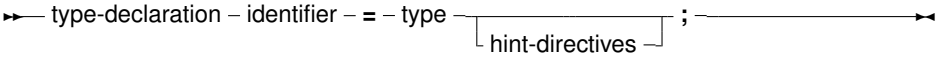
Chapter 3

Types

All variables have a type. Free Pascal supports the same basic types as Turbo Pascal, with some extra types from Delphi as well as some of its own.

The programmer can declare his own types, which is in essence defining an identifier that can be used to denote this custom type when declaring variables further in the source code. Declaring a type happens in a `Type` block (section 16.5, page 266), which is a collection of type declarations, separated by semicolons:

Type declaration



There are eight major kinds of types:

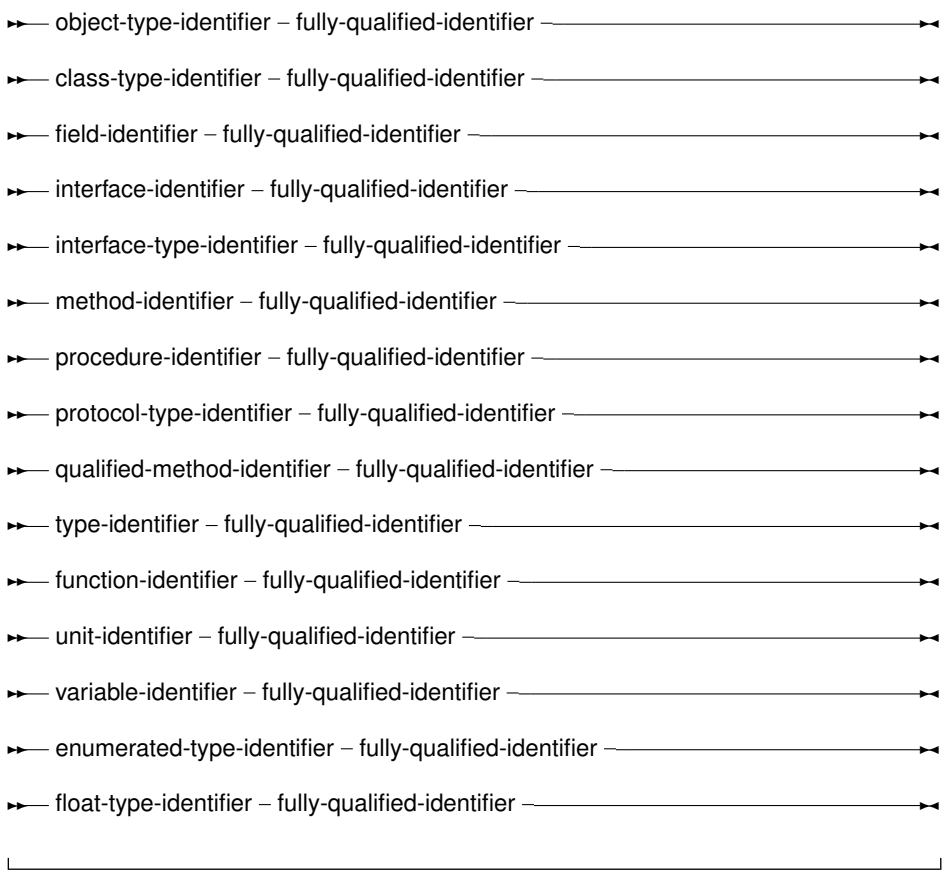
Types



Each of these cases will be examined separately.

A syntax diagram has no notion of the type of an identifier or expression. However, sometimes it is useful (or necessary) to know the type of an identifier. To make the syntax diagrams more clear, the type of an identifier has been added as a prefix to the name. The following is a list of such typed identifiers:

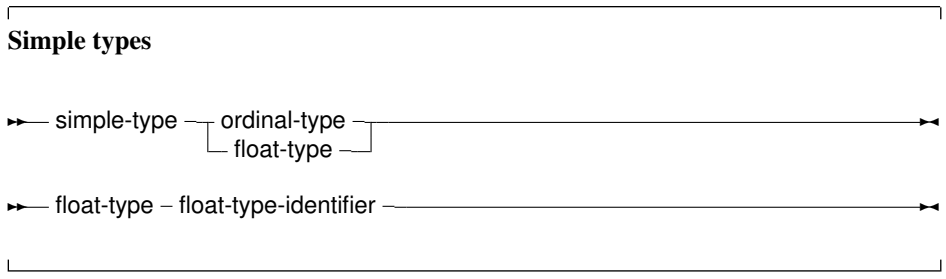
Typed identifiers



from a perspective of language syntax, they can all be replaced by `fully-qualified-identifier`

3.1 Base types

The base or simple types of Free Pascal are the Delphi types. We will discuss each type separately.



3.1.1 Ordinal types

With the exception of floating point value types, all base types are ordinal types. Ordinal types have the following characteristics:

1. Ordinal types are countable and ordered, i. e. it is, in principle, possible to start counting them one by one, in a specified order. This property allows the operation of functions as `Inc`, `Ord`, `Dec` on ordinal types to be defined.
2. Ordinal values have a smallest possible value. Trying to apply the `Pred` function on the smallest possible value will generate a range check error if range checking is enabled.
3. Ordinal values have a largest possible value. Trying to apply the `Succ` function on the largest possible value will generate a range check error if range checking is enabled.
4. Integer ordinal types are mutually assignment compatible: there is no need to typecast when assigning an ordinal value of one (e.g. smallint) to another (e.g. byte). If you assign a value to a variable of an ordinal type, and that value is actually out of the range of valid values for that type, this will not result in a run-time error, although it may be a logical error. Only when range checking is enabled will such an assignment result in a range check error.

The fact that ordinals are mutually assignment compatible means that the following program will work without errors when compiled without range checking:

```
var
   b: byte;
   si: integer;
   w: word;
begin
   b := 33;
   si := 512;
   b := si; // out of range (too big)
   writeLn(b);  // shows zero!

   b := 33;
   si := -512;
   b := si; // out of range (negative)
   writeLn(b);  // shows zero!

   w := 33;
   si := -9;
   w := si; // Out of range (negative)
   writeLn(w);  // shows 65527!

   b := 33;
   si := 512;
   b := byte(si); // out of range (too big)
   writeLn(b);  // shows zero!
end.
```

Only when you compile with range checking by specifying `-Cr` on commandline, or adding `{$R+}` to the source code, will the compiler insert code to check whether the assigned value is valid for the target ordinal type.

In the last example, a typecast is used, forcing the right-hand side of the assignment to the same type as the left-hand side. In that case, no range checks will be inserted, even when range checking is enabled.

Remark `Int64` and `QWord` are considered ordinal types on 64-bit CPUs. On 32-bit types they have some of the characteristics of ordinals, but they cannot be used e. g. in for loops.

Integers

A list of predefined integer types is presented in table (3.1).

Table 3.1: Predefined integer types

Name
Integer
Shortint
SmallInt
Longint
Longword
Int64
Byte
Word
Cardinal
QWord
ByteBool
WordBool
LongBool
QWordBool

The integer types, and their ranges and sizes, that are predefined in Free Pascal are listed in table (3.2).

Table 3.2: Predefined integer ranges

Type	Range	Size in bytes
Byte	0 .. 255	1
Shortint	-128 .. 127	1
Smallint	-32768 .. 32767	2
Word	0 .. 65535	2
Integer	either smallint or longint	size 2 or 4
Cardinal	longword	4
Longint	-2147483648 .. 2147483647	4
Longword	0 .. 4294967295	4
Int64	-9223372036854775808 .. 9223372036854775807	8
QWord	0 .. 18446744073709551615	8

The `integer` type is an alias to the smallint type in the default Free Pascal mode. It is an alias for the longint type in either Delphi or ObjFPC mode. The `cardinal` type is currently always mapped to the longword type.

Remark The compiler decides on the type of an integer constant based on the value: An integer constant gets the smallest possible signed type. The first match in table (3.3) is used.

Table 3.3: Integer constant type mapping

Range	Type
-128..127	Shortint
128..255	Byte
-32768..32767	Smallint
32768..65535	Word
-2147483648..2147483647	longint
2147483648..4294967295	Cardinal (longword)
-9223372036854775808 .. 9223372036854775807	Int64

That means constants in the range -128..127 are mapped to shortint, constants in range 128..255 are mapped to byte, etc. Constants in the range 2147483647..high(cardinal) are parsed as cardinals, and all decimal constants which do not fit either of the above ranges are parsed as 64-bit integer constants.

Remark In newer Delphi versions, the `longint` type is platform and CPU dependent. This is not so in FPC, where longint is 32-bit on all platforms.

As a pascal compiler, Free Pascal does automatic type conversion and upgrading in expressions where different kinds of integer types are used:

1. Every platform has a "native" integer size, depending on whether the platform is 8-bit, 16-bit, 32-bit or 64-bit. E. g. on AVR this is 8-bit.
2. Every integer smaller than the "native" size is promoted to a signed version of the "native" size. Integers equal to the "native" size keep their signedness.
3. The result of binary arithmetic operators (+, -, *, etc.) is determined in the following way:
   (a) If at least one of the operands is larger than the native integer size, the result is chosen to be the smallest type that encompasses the ranges of the types of both operands. This means that mixing an unsigned with a smaller or equal in size signed will produce a signed type that is larger than both of them.
   (b) If both operands have the same signedness, the result is the same type as them. The only exception is subtracting ($-$): in the case of unsigned $-$ unsigned subtracting produces a signed result in FPC (as in Delphi, but not in TP7).
   (c) Mixing signed and unsigned operands of the "native" int size produces a larger signed result. This means that mixing longint and longword on 32-bit platforms will produce an int64. Similarly, mixing byte and shortint on 8-bit platforms (AVR) will produce a smallint.

Boolean types

Free Pascal supports the `Boolean` type, with its two predefined possible values `True` and `False`. These are the only two values that can be assigned to a `Boolean` type. Of course, any expression that resolves to a `boolean` value, can also be assigned to a boolean type.

Table 3.4: Boolean types

Name	Size	Ord(True)
Boolean	1	1
Boolean8	1	1
Boolean16	2	1
Boolean32	4	1
Boolean64	8	1
ByteBool	1	Any nonzero value
WordBool	2	Any nonzero value
LongBool	4	Any nonzero value
QWordBool	8	Any nonzero value

In addition to the simple `Boolean` type, the additional `Boolean8`, `Boolean16`, `Boolean32` and `Boolean64` types exist. These are in fact integer types, which are assignment-compatible with the simple boolean type. As an integer, the values for `True` and `False` are 1 and 0. This can be used to interface with C code that defines a boolean of this size with values 0 and 1.

To make interfacing with C even easier, Free Pascal also supports the `ByteBool`, `WordBool`, `LongBool` and `QWordBool` types. These are of type `Byte`, `Word`, `Longint` or `Int64`, but are again assignment compatible with a `Boolean`. The only difference with the `Boolean8/16/32/64` types is in what values are considered true or false: The value `False` is equivalent to 0 (zero) and any nonzero value is considered `True` when converting to a boolean value. A boolean value of `True` is converted to `Not(0)` in case it is assigned to a variable of type `ByteBool`, `WordBool`, `LongBool` or `QWordBool`.

Assuming `B` to be of type `Boolean`, the following are valid assignments:

```
B := True;
B := False;
B := 1<>2;  { Results in B := True }
```

Boolean expressions are also used in conditions.

Remark In Free Pascal, boolean expressions are by default always evaluated in such a way that when the result is known, the rest of the expression will no longer be evaluated: this is called short-cut boolean evaluation.

In the following example, the function `Func` will never be called, which may have strange side-effects.

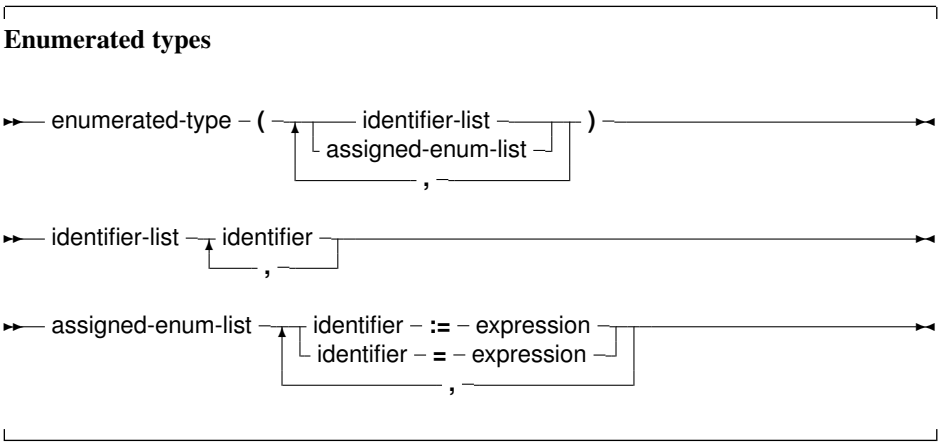
```
...
B := False;
A := B and Func;
```

Here `Func` is a function which returns a `Boolean` type.

This behavior is controllable by the `{$B }` compiler directive.

Enumeration types

Enumeration types are supported in Free Pascal. On top of the Turbo Pascal implementation, Free Pascal allows also a C-style extension of the enumeration type, where a value is assigned to a particular element of the enumeration list.



(see chapter 12, page 178 for how to use expressions) The expressions used in assigned enumerated elements must be known at compile time. So the following is a correct enumerated type declaration:

```
Type
  Direction = ( North, East, South, West );
```

A C-style enumeration type looks as follows:

```
Type
  EnumType = (one, two, three, forty := 40,fortyone);
```

or you can use

```
Type
  EnumType = (one, two, three, forty = 40,fortyone);
```

The latter notation is mandatory in mode `DELPHI`.

As a result, the ordinal number of `forty` is `40`, and not `3`, as it would be when the `':= 40'` wasn't present. The ordinal value of `fortyone` is then 41, and not `4`, as it would be when the assignment wasn't present. After an assignment in an enumerated definition the compiler adds 1 to the assigned value to assign to the next enumerated value.

As far as the compiler is concerned, the number of elements in an enumeration with assigned values is determined by the highest value in the list of values: in the above example, the number of elements is 42, but the elements with ordinal values 3 to 39 simply do not have a name.

When specifying such an enumeration type, it is important to keep in mind that the enumerated elements should be kept in ascending order. The following will produce a compiler error:

```
Type
  EnumType = (one, two, forty := 40, thirty := 30);
```

When using enumeration types it is important to keep the following points in mind:

1. The `Pred` and `Succ` functions cannot be used on enumeration types with assigned values. Trying to do this anyhow will result in a compiler error.
2. If the largest element in the list has an ordinal value greater than 256, the enumeration cannot be used to construct a set.

3. Enumeration types are stored using a default, independent of the actual number of values: the compiler does not try to optimize for space. This behavior can be changed with the `{$PACKENUM n}` compiler directive, which tells the compiler the minimal number of bytes to be used for enumeration types. For instance

```
Type
{$PACKENUM 4}
  LargeEnum = ( BigOne, BigTwo, BigThree );
{$PACKENUM 1}
  SmallEnum = ( one, two, three );
Var S : SmallEnum;
    L : LargeEnum;
begin
  WriteLn ('Small enum : ',SizeOf(S));
  WriteLn ('Large enum : ',SizeOf(L));
end.
```

   will, when run, print the following:

```
Small enum : 1
Large enum : 4
```

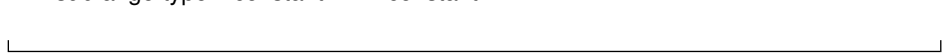
4. RTTI cannot be generated for assigned enumerated types (C-enumerations).

More information can be found in the Programmer's Guide, in the compiler directives section.

Subrange types

A subrange type is a range of values from an ordinal type (the *host* type). To define a subrange type, one must specify its limiting values: the highest and lowest value of the type.

Subrange types



Some of the predefined `integer` types are defined as subrange types:

```
Type
  Longint  = $80000000..$7fffffff;
  Integer  = -32768..32767;
  shortint = -128..127;
  byte     = 0..255;
  Word     = 0..65535;
```

Subrange types of enumeration types can also be defined:

```
Type
  Days = (monday,tuesday,wednesday,thursday,friday,
          saturday,sunday);
  WorkDays = monday .. friday;
  WeekEnd = Saturday .. Sunday;
```

Character types

A character type is also an ordinal type: characters are ordered, can be counted. There are 2 character types:

AnsiChar This is a 1-byte character. The interpretation of the character depends on the codepage.

WideChar This is a 2-byte character. The interpretation of the character depends on the codepage.

Characters can be used in a loop, one can use `prev` and `succ` on it, as well as `ord`.

For more information on characters, see section 3.2, page 33.

3.1.2 Floating-point types

Free Pascal uses the math coprocessor (or emulation) for all its floating-point calculations. The `Real` native type is processor dependent, but it is either a `Single` or a `Double`. Only the IEEE floating point types are supported, and these depend on the target processor and emulation options. The true Turbo Pascal compatible types are listed in table (3.5).

Table 3.5: Supported floating-point types

Type	Range	Significant digits	Size
Real	platform dependant	???	4 or 8
Single	1.5E-45 .. 3.4E38	7–8	4
Double	5.0E-324 .. 1.7E308	15–16	8
Extended	1.9E-4932 .. 1.1E4932	19–20	10
Comp	-2ĝ64+1 .. 2<ĝ63-1	19–20	8
Currency	-922337203685477.5808 .. 922337203685477.5807	19–20	8

The `Comp` type is, in effect, a 64-bit integer and is not available on all target platforms. To get more information on the supported types for each platform, refer to the Programmer's Guide.

The currency type is a fixed-point real data type which is internally used as an 64-bit integer type (automatically scaled with a factor 10000), this minimizes rounding errors. This type should be used with care: when used in expressions using e.g. multiplication, the evaluation of the expression may go wrong (losing precision) if intermediate results fall outside the currency range.

Note that not all floating point types are available on all platforms. The `single` float type is the only one guaranteed to be available on all platforms that have floating point support (so e.g. AVR does not have it). The `double` type is available on all platforms with coprocessor, and the `extended` type is available on all Intel x86 processors, except on the Windows 64-bit platform. More details on availability can be found in the Programmer's Guide.

3.2 Character types

3.2.1 Char or AnsiChar

Free Pascal supports the type `Char`. A `Char` is exactly 1 byte in size, and contains one ASCII character.

A character constant can be specified by enclosing the character in single quotes, as follows: `'a'` or `'A'` are both character constants.

A character can also be specified by its character value (commonly an ASCII code), by preceding the ordinal value with the number symbol (#). For example specifying `#65` would be the same as `'A'`.

Also, the caret character (`^`) can be used in combination with a letter to specify a character with ASCII value less than 27. Thus `^G` equals `#7` − G is the seventh letter in the alphabet. The compiler is rather sloppy about the characters it allows after the caret (anything with ASCII code <128 will work), but in general one should assume only letters.

When the single quote character must be represented, it should be typed two times successively, thus `''''` represents the single quote character.

To distinguish `Char` from `WideChar`, the system unit also defines the `AnsiChar` type, which is the same as the char type. In future versions of FPC, the `Char` type may become an alias for either `WideChar` or `AnsiChar`.

3.2.2 WideChar

Free Pascal supports the type `WideChar`. A `WideChar` is exactly 2 bytes in size, and contains one codepoint in UTF-16 encoding.

A Unicode character can be specified by its codepoint (an UTF-16 code), by preceding the ordinal value with the number symbol (#).

A normal ANSI (1-byte) character literal can also be used for a WideChar, the compiler will automatically convert it to a 2-byte UTF-16 codepoint.

The following defines some greek characters (phi, omega):

```
Const
  C3 : widechar = #$03A8;
  C4 : widechar = #$03A9;
```

The same can be accomplished by typecasting a word to WideChar:

```
Const
  C3 : widechar = widechar($03A8);
  C4 : widechar = widechar($03A9);
```

3.2.3 Other character types

Free Pascal defines some other character types in the system unit such as `UCS2Char`, `UCS4Char`, `UniCodeChar`. However, no special support for these character types exists, they have been defined for Delphi compatibility only.

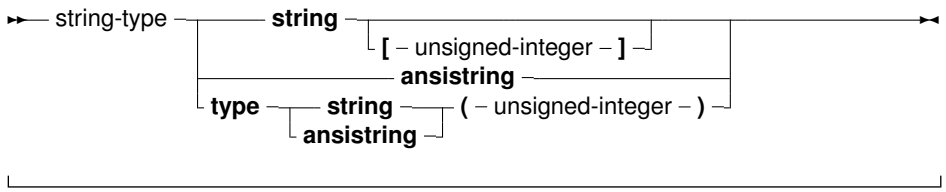
3.2.4 Single-byte String types

Free Pascal supports the `String` type as it is defined in Turbo Pascal: a sequence of single-byte characters with an optional size specification, these are called shortstrings. It also supports ansistrings (with unlimited length) and codepage information[1] as in Delphi.

To declare a variable as a string, use the following type specification:

String Type

[1] As of version 3.0 of Free Pascal



The optional size specifier in brackets, e.g. `String[128]`, has a twofold effect:

- The string is a ShortString.
- The string length is limited to the value given by the specifier.

The optional code page specifier in parentheses, e.g. `String(1252)`, has a twofold effect:

- The string is an ansistring with no explicit length limit.
- The string is associated with the codepage given by the specifier.

The size and code page specifiers are optional. A string declaration statement without size and code page specifiers is interpreted differently depending on the `{$H}` switch:

```
var
  A : String;
```

If no size and code page specifier are present, the above declaration can declare an ansistring or a short string - both are explained in detail in the following sections.

Whatever the actual type, single byte strings can be used interchangeably. The compiler always takes care of the necessary type conversions. Note, however, that the result of an expression that contains ansistrings and short strings will always be an ansistring.

Short strings

"Short strings" have a declared reserved memory size. This avoids memory management overhead and can have other advantages in special cases. It is also used for compatibility with Turbo Pascal, and they always use the system encoding.

Short strings can be declared in the following ways:

1. Length is explicitly specified in square brackets.
2. Predefined type `ShortString` specified.
3. The specified type is `String` and the `{$H}` switch is off.

In the last two cases, a maximum length of 255 is assumed, since the value can only be between 1 and 255, inclusive.

```
{$H-} // String is ShortString
var
  S1: String[10];  // can contain a maximum of 10 characters
  S2: ShortString; // can contain up to 255 characters
  S3: String;      // can contain up to 255 characters
```

Remark The value of the {$H} ({$LONGSTRINGS}) switch can be specified explicitly by a directive in the code, taken from compilation options, from a configuration file, or nothing specified (off by default).

Remark Note that ShortString is not a special string type in itself - it is just a pre-declared String[255]. Therefore, a maximum length cannot be specified (or overridden) for it:

```
var
  S1: String[10];      // correct
  S2: ShortString;     // correct
  S3: ShortString[10]; // error
```

User programs can also declare similar types:

```
type
  TUserName = String[20];
var
  S: TUserName;
```

The maximum string length is part of the type declaration and indicates the size of the reserved memory. Therefore, it cannot be changed during program execution.

The programmer must take into account the length of the data being assigned, as if the maximum length is exceeded, the excess portion is simply skipped. Even enabled range checking will not cause errors. This must be taken into account for all operations involving short strings (assignment, concatenation, calling SetLength, etc).

To get the maximum length of a short string, SizeOf can be used, which returns its value plus one.

Don't confuse this with getting the current data length. The actual length of the string can be obtained with the Length standard runtime routine:

```
var
  S: String[8];
begin
  S:='abcd';
  WriteLn(Length(S)); // 4
  WriteLn(SizeOf(S)); // 9 (8+1)
```

Note that the SetLength function also sets the current data length. A size greater than the maximum cannot be specified.

For short strings, the length is stored in the character at index 0. Old Turbo Pascal code relies on this, and it is implemented similarly in Free Pascal. To write more portable code (and independent of different string types), it is recommended not to rely on this feature, and to use the standard Length/SetLength functions to get/set the length of strings.

Ansistrings

Ansistrings are strings that have no length limit, and have a code page associated with them[2]. They are reference counted and are guaranteed to be null terminated.

Internally, an ansistring is treated as a pointer: the actual content of the string is stored on the heap, as much memory as needed to store the string content is allocated.

If no codepage is given in the declaration, the system codepage is assumed. What codepage this is, is determined by the DefaultSystemCodePage constant in the system unit.

[2] codepage was introduced in version 3.0 of Free Pascal

This is all handled transparently, i. e. they can be manipulated as a normal short string. Ansistrings can be defined using the predefined `AnsiString` type or using the `string` keyword in mode `{$H+}`.

Remark The null-termination does not mean that null characters (char(0) or #0) cannot be used: the null-termination is not used internally, but is there for convenience when dealing with external routines that expect a null-terminated string (as most C routines do).

If the `{$H}` switch is on, then a string definition using the regular `String` keyword that doesn't contain a length specifier, will be regarded as an ansistring as well. If a length specifier is present, a short string will be used, regardless of the `{$H}` setting.

If the string is empty (`''`), then the internal pointer representation of the string pointer is `Nil`. If the string is not empty, then the pointer points to a structure in heap memory.

The internal representation as a pointer, and the automatic null-termination make it possible to typecast an ansistring to a pchar. If the string is empty (so the pointer is `Nil`) then the compiler makes sure that the typecast pchar will point to a null byte.

Assigning one ansistring to another doesn't involve moving the actual string. A statement

```
S2:=S1;
```

results in the reference count of `S2` being decreased with 1, The reference count of `S1` is increased by 1, and finally `S1` (as a pointer) is copied to `S2`. This is a significant speed-up in the code.

If the reference count of a string reaches zero, then the memory occupied by the string is deallocated automatically, and the pointer is set to `Nil`, so no memory leaks arise.

When an ansistring is declared, the Free Pascal compiler initially allocates just memory for a pointer, not more. This pointer is guaranteed to be `Nil`, meaning that the string is initially empty. This is true for local and global ansistrings or ansistrings that are part of a structure (arrays, records or objects).

Remark Note that a function result in this regard is considered equivalent to a `var` parameter and hence will not be initialized to `Nil`. As a consequence it may point to a legitimate non-`Nil` ansistring when the function begins.

This does introduce an overhead. For instance, declaring

```
Var
  A : Array[1..100000] of string;
```

Will copy the value `Nil` 100,000 times into `A`. When `A` goes out of scope, then the reference count of the 100,000 strings will be decreased by 1 for each of these strings. All this happens invisible to the programmer, but when considering performance issues, this is important.

Memory for the string content will be allocated only when the string is assigned a value. If the string goes out of scope, then its reference count is automatically decreased by 1. If the reference count reaches zero, the memory reserved for the string is released.

If a value is assigned to a character of a string that has a reference count greater than 1, such as in the following statements:

```
S:=T;  { reference count for S and T is now 2 }
S[I]:='@';
```

then a copy of the string is created before the assignment. This is known as *copy-on-write* semantics. It is possible to force a string to have reference count equal to 1 with the `UniqueString` call:

```
S:=T;
R:=T; // Reference count of T is at least 3
```

```
  UniqueString(T);
  // Reference count of T is guaranteed 1
```

It's recommended to do this e. g. when typecasting an ansistring to a PChar var and passing it to a C routine that modifies the string.

Constant AnsiStrings have a reference count of `-1` and are treated specially.

The `Length` function must be used to get the length of an ansistring: the length is not stored at character 0 of the ansistring.

```
var S: AnsiString;
begin
  L:=length(S); // correct, for AnsiString and ShortString
  L:=ord(S[0]); // error, only works for ShortStrings
```

The same remark as for `Length` must be given: to set the length of an AnsiString, the `SetLength` function must be used.

```
var S: AnsiString;
begin
  SetLength(S,4); // corrent, for AnsiString and ShortString
  S[0]:=Char(4);  // error, only works for ShortString
```

Ansistrings are converted to short strings by the compiler if needed, this means that the use of ansistrings and short strings can be mixed without problems.

Ansistrings can be typecast to `PChar` or `Pointer` types:

```
Var P : Pointer;
    PC : PChar;
    S : AnsiString;

begin
  S :='This is an ansistring';
  PC:=Pchar(S);
  P :=Pointer(S);
```

There is a difference between the two typecasts. When an empty ansistring is typecast to a pointer, the pointer will be `Nil`. If an empty ansistring is typecast to a `PChar`, then the result will be a pointer to a zero byte (an empty string).

The result of such a typecast must be used with care. In general, it is best to consider the result of such a typecast as read-only, i. e. only suitable for passing to a procedure that needs a constant pchar argument.

It is therefore *not* advisable to typecast one of the following:

1. Expressions.
2. Strings that have reference count larger than 1. In this case you should call `Uniquestring` to ensure the string has reference count 1.

Code page conversions

Since strings have code page information associated with them, it is important to know which code page a string uses:

- Short strings always use the system code page.
- Plain ansistrings use the system code page.
- Single byte strings with declared code page use that code page.
- The `RawBytestring` type has no code page information associated with it.
- Constant strings have the code page of the source file. If none is specified the system codepage is used (CP_ACP).
  See Programmer's Guide, {$CODEPAGE } directive.

This code page is called the declared code page.

The compiler will convert the code page of strings as needed: When assigning a string, the actual codepage of the source string will be converted to the declared code page of the target string if the declared source and target code pages are different.

If a string with a declared page `SOURCE_CP` assigned to a string with declared code page `DEST_CP`, in a file with code page `CODE_CP` then the following describes the mechanism:

- if (`SOURCE_CP=CP_NONE`) or (`DEST_CP = CP_NONE`), see RawByteString.
- if (`CODE_CP <> CP_ACP`), then if (`DEST_CP = CP_ACP`) and (`SOURCE_CP = CODE_CP`) or vice versa, no conversion will occur, even if at run time `DefaultSystemCodePage` has a different value from `SOURCE_CP`.
  The reason for this "(`CODE_CP <>CP_ACP`)" condition is backward compatibility with previous FPC versions: While they did not support `AnsiStrings` with arbitrary code pages, they did always reinterpret `AnsiStrings` according to the current value of the system code page. Otherwise,
- if (`SOURCE_CP <> DEST_CP`), the string data will be converted from codepage `SOURCE_CP` to codepage `DEST_CP` before assignment, whereby `CP_ACP` will be interpreted as the current value of `DefaultSystemCodePage`. Otherwise,
- if (`SOURCE_CP = DEST_CP`), no codepage conversion will be performed.

These rules mean that it is perfectly possible for an `AnsiString` variable to get a code page that differs from its declared code page. E. g. in the third case `SOURCE_CP` could be `CP_ACP`, while after the assignment it may have a dynamic code page equal to `DefaultSystemCodePage`.

Note: as mentioned above, whether or not a potential code page conversion happens only depends on the declared code pages of the involved strings. This means that if you assign one AnsiString(X) to another AnsiString(X) and the former's dynamic code was different from X, the string data will not be converted to code page X by the assignment.

All this means that in the following code:

```
{$h+}
uses sysutils;

Type
  TString1 = Type String(1252);
  TString2 = Type String(1251);

Var
  A : TString1;
  B : TString2;
```

```
begin
  A:='123'+'345'+intToStr(123);
  B:=A;
  Writeln('B: "',B,'" : ',StringRefCount(B),' -> ',StringCodePage(B));
  Writeln('A: "',A,'" : ',StringRefCount(A),' -> ',StringCodePage(A));
end.
```

This will print:

```
B: "123345123" : 1 -> 1251
A: "123345123" : 1 -> 1252
```

During the assignment of `A` to `B`, the contents of string `A` is converted to the codepage of string `B`. Note that if a code page conversion takes place, the reference count mechanism is not used: a new string will be allocated.

This automated conversion of code pages can slow down the code seriously, so care must be taken to see to it that the code page conversions are limited to a minimum.

The code page of a string can be set explicitly using the `SetCodePage` routine of the system unit. Calling this routine will convert the value of a string to the requested code page.

Remark Code page conversions can result in loss of data: if a certain character cannot be represented in the target code page, the output for that character is undefined.

Remark When a string whose static code page equals the source file code page, to anything with code page CP_ACP (i. e., a plain ansistring, shortstring, or pchar), no conversion will be performed either. No code page conversion is done when s can result in loss of data: if a certain character cannot be represented in the target code page, the output for that character is undefined.

Remark Code page support requires quite some helper routines, these are implemented in the unicodestring manager. On windows, the system routines are used for this. On Unices, the `cwstring` unit can be used to link to the C library and use the C library conversion support. Alternatively, the `fpwidestring` unit contains a unicodestring manager implemented natively in Object Pascal.

RawByteString

The predefined `RawByteString` type is an ansistring string type without codepage information (`CP_NONE`):

```
Type
  RawByteString = type ansistring(CP_NONE);
```

It is treated specially in the sense that if the conversion routines encounter `CP_NONE` in a source or target string, no code page conversion is performed, the code page of the source string is preserved.

For this reason, most single-byte string routines in the system and sysutils units use the `RawByteString` type.

UTF8String

Single-byte code page strings can only store the characters available in that code page. Characters that are not present in the code page, cannot be represented by that string. The UTF-8 unicode encoding is an encoding that can be used with single-byte strings: The ASCII characters (ordinal value <128) in this encoding map exactly to the CP_ACP encoding. This fact is used to define a single byte string type that can contain all characters:

```
Type
  UTF8String = type AnsiString(CP_UTF8);
```

The `UTF8string` string type can be used to represent all Unicode characters. This power comes as a price, though. Since a unicode character may require several bytes to be represented in the UTF-8 encoding, there are two points to take care of when using UTF8String:

1. The character index – which retrieves a byte-sized char at a certain position – must be used with care: the expression `S[i]` will not necessarily be a valid character for a string `S` of type `UTF8String`.
2. The byte length of the string is not equal to the number of characters in the string. The standard function `length` cannot be used to get the character length, it will always return the byte length.

For all other code pages, the number of characters in a single-byte code page string is equal to the byte length of the string.

3.2.5 Multi-byte String types

For multi-byte string types, the basic character has a size of at least 2. This means it can be used to store a unicode character in UTF16 or UCS2 encoding.

UnicodeStrings

Unicodestrings (used to represent unicode character strings) are implemented in much the same way as ansistrings: reference counted, null-terminated arrays, only they are implemented as arrays of `WideChars` instead of regular `Chars`. A `WideChar` is a two-byte character (an element of a DBCS: Double Byte Character Set). Mostly the same rules apply for `UnicodeStrings` as for `AnsiStrings`. The compiler transparently converts UnicodeStrings to AnsiStrings and vice versa.

Similarly to the typecast of an Ansistring to a `PChar` null-terminated array of characters, a UnicodeString can be converted to a `PUnicodeChar` null-terminated array of characters. Note that the `PUnicodeChar` array is terminated by 2 null bytes instead of 1, so a typecast to a pchar is not automatic.

The compiler itself provides no support for any conversion from Unicode to ansistrings or vice versa. The system unit has a unicodestring manager record, which can be initialized with some OS-specific unicode handling routines. For more information, see the system unit reference.

A unicode string literal can be constructed in a similar manner as a widechar:

```
Const
  ws2: unicodestring = 'phi omega : '#$03A8' '#$03A9;
```

WideStrings

The `Widestring` type (used to represent unicode character strings in COM applications) is implemented in much the same way as `Unicodestring` on Windows, and on other platforms, they are simply the same type. If interaction with COM is not required, the `UnicodeString` type should be used.

On Windows, unlike `UnicodeString`, the `WideString` type is *not* reference counted, and are allocated with a special windows function which allows them to be used for OLE automation. This

means they are implemented as null-terminated arrays of `WideChars` instead of regular `Chars`. `WideString` obeys the same rules as for `UnicodeStrings`. Similar to unicodestrings, the compiler transparently converts WideStrings to AnsiStrings and vice versa.

For typecasting and conversion, the same rules apply as for the `UnicodeString` type.

Note that on windows, because a `WideString` string is allocated using a special windows function, the memory layout differs from `UnicodeString`. The length for instance is stored in bytes rather than characters.

3.2.6 Constant strings

To specify a constant string, it must be enclosed in single-quotes, just as a `Char` type, only now more than one character is allowed. Given that `S` is of type `String`, the following are valid assignments:

```
S := 'This is a string.';
S := 'One'+', Two'+', Three';
S := 'This isn''t difficult!';
S := 'This is a weird character : '#145' !';
```

As can be seen, the single quote character itself within the text is represented by 2 repetitions. Strange characters can be specified by their character value (usually an ASCII code). The example shows also that two strings can be added. The resulting string is just the concatenation of the first with the second string, without spaces in between them. Strings can not be subtracted, however.

Whether the constant string is stored as an ansistring or a short string depends on the settings of the `{$H}` switch.

3.2.7 PChar – Null terminated strings

Free Pascal supports the Delphi implementation of the `PChar` type. `PChar` is defined as a pointer to a `Char` type, but allows additional operations. The `PChar` type can be understood best as the Pascal equivalent of a C-style null-terminated string, i. e. a variable of type `PChar` is a pointer that points to an array of type `Char`, which is ended by a null-character (`#0`). Free Pascal supports initializing of `PChar` typed constants, or a direct assignment. For example, the following pieces of code are equivalent:

```
program one;
var P : PChar;
begin
  P := 'This is a null-terminated string.';
  WriteLn (P);
end.
```

Results in the same as

```
program two;
const P : PChar = 'This is a null-terminated string.';
begin
  WriteLn (P);
end.
```

These examples also show that it is possible to write *the contents* of the string to a file of type `Text`. Since it is equivalent to a pointer to a type `Char` variable, it is also possible to do the following:

```
Program three;
Var S : String[34];
    P : PChar;
begin
  S := 'This is a null-terminated string.'#0;
  P := @S[1];
  WriteLn (P);
end.
```

This will have the same result as the previous two examples.

Note that characters in String start with 1, not 0. This is important to consider when assigning a pointer:

```
P := @S;    // error, will point to the "length byte"
P := @S[0]; // error as above
P := @S[1]; // correct
```

As already shown, for a ShortString (unlike AnsiString), a terminating null byte must be explicitly added if you want to use it as a reference for a PChar. This should also be done when assigning empty strings:

```
var
  S: ShortString;
  P: PChar;
begin
  S := 'abcd';
  S := '12';
  P := @S[1];
  WriteLn(P);   // print "12cd" and possibly other garbage
  S := '12'+#0;
  WriteLn(P);   // print "12"
  S := #0;
  WriteLn(P);   // print ""
end.
```

Remark This example also demonstrates that a PChar is not a separate string, but a reference to an existing one (or even part of one). Therefore, changing the contents of `S` also affects `P`.

The maximum length of a short string must accommodate the entire text, including the terminating null byte.

Remark There are many ways to write the null character:

```
S:='abcd'#0;        // character constant
S:='abcd'+#0;       // character constant (alternative)
S:='abcd'+chr(0);   // function Chr
S:='abcd'+char(0); // type casting
```

Due to the nature of `PChar` strings, many standard string operations (e.g. concatenation) are unavailable. To work with them correctly, functions from the strings unit must be used as in the standard C library.

However, some still work. For example, to obtain string length (taking care of the null character), the standard `Length` function can be used if the `PChar` type is explicitly specified (or cast):

```
var
```

```
  S: String = 'ab'+#0+'cd';
  P: PChar;
begin
  P:=@S[1];
  WriteLn(Length(S));             // 5
  WriteLn(Length(P));             // 2
  WriteLn(Length(PChar(@S[1]))); // 2
```

Also note some peculiarities of pointer arithmetic. The operators + and − can be used to do operations on `PChar` pointers. In table (3.6), `P` and `Q` are of type `PChar`, and `I` is of type `Longint`.

Table 3.6: `PChar` pointer arithmetic

Operation	Result
`P + I`	Adds `I` to the address pointed to by `P`.
`I + P`	Adds `I` to the address pointed to by `P`.
`P - I`	Subtracts `I` from the address pointed to by `P`.
`P - Q`	Returns, as an integer, the distance between two addresses (or the number of characters between `P` and `Q`)

`PChar` doesn't necessarily point to the beginning of the string, but then you must ensure it points to the correct data. In this case, it will act as a string "truncated" from the beginning:

```
var
  S: String = 'abcd'#0;
  P: PChar;
begin
  P:=@S[3];
  WriteLn(P); // cd
```

Note that "extra" null characters within the string can even split it into "multiple" `PChar` strings:

```
var
  S: String = 'ab'+#0+'cd'+#0;
  P: PChar;
begin
  P:=@S[1];
  WriteLn(P); // ab
  P:=P+3;
  WriteLn(P); // cd
```

In general, it's recommended to be careful with them, especially when writing.

3.2.8 String sizes

The memory occupied by a string depends on the string type. Some string types allocate the string data in memory on the heap, others have the string data on the stack. Table table (3.7) summarizes the memory usage of the various string types for the various string types. In the table, the following symbolic constants are used:

1. `L` is the actual length of the string.

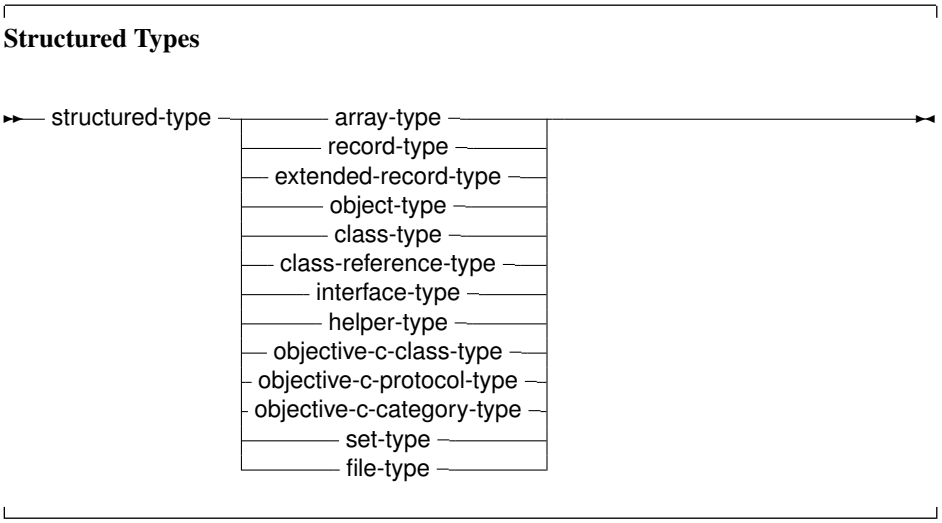
2. HS depends on the version of Free Pascal, but is 16 bytes as of Free Pascal 2.7.1.
3. UHS size is 8 bytes for all versions of Free Pascal.
4. On Windows, WHS size is 4 bytes for all versions of Free Pascal. On all other platforms, WHS equals UHS because the WideString type equals the UnicodeString type.

Table 3.7: String memory sizes

String type	Stack size	heap size
Shortstring	Declared length + 1	0
Ansistring	Pointer size	L + 1 + HS
Widestring	Pointer size	2*(L + 1) + WHS
UnicodeString	Pointer size	2*(L + 1) + UHS
Pchar	Pointer size	L+1

3.3 Structured Types

A structured type is a type that can hold multiple values in one variable. Structured types can be nested to unlimited levels. There are many possible structured types:



Each of these types will be treated in a separate section or even chapter.

Unlike Delphi, Free Pascal does not support the keyword Packed for all structured types. In the following sections each of the possible structured types is discussed. It will be mentioned when a type supports the packed keyword.

Packed structured types

When a structured type is declared, no assumptions should be made about the internal position of the elements in the type. The compiler will lay out the elements of the structure in memory as it thinks will be most suitable. That is, the order of the elements will be kept, but the location of the elements are not guaranteed, and is partially governed by the $PACKRECORDS directive (this directive is explained in the Programmer's Guide).

However, Free Pascal allows controlling the layout with the `Packed` and `Bitpacked` keywords. The meaning of these words depends on the context:

Bitpacked In this case, the compiler will attempt to align ordinal types on bit boundaries, as explained below.

Packed The meaning of the `Packed` keyword depends on the situation:

1. In `MACPAS` mode, it is equivalent to the `Bitpacked` keyword.
2. In other modes, with the `$BITPACKING` directive set to `ON`, it is also equivalent to the `Bitpacked` keyword.
3. In other modes, with the `$BITPACKING` directive set to `OFF`, it signifies normal packing on byte boundaries.

Packing on byte boundaries means that each new element of a structured type starts on a byte boundary.

The byte packing mechanism is simple: the compiler aligns each element of the structure on the first available byte boundary, even if the size of the previous element (small enumerated types, subrange types) is less than a byte.

When using the bit packing mechanism, the compiler calculates for each ordinal type how many bits are needed to store it. The next ordinal type is then stored on the next free bit. Non-ordinal types – which include but are not limited to – sets, floats, strings, (bitpacked) records, (bitpacked) arrays, pointers, classes, objects, and procedural variables, are stored on the first available byte boundary.

Note that the internals of the bitpacking are opaque: they can change at any time in the future. What is more: the internal packing depends on the endianness of the platform for which the compilation is done, and no conversion between platforms are possible. This makes bitpacked structures unsuitable for storing on disk or transport over networks. The format is however the same as the one used by the GNU Pascal Compiler, and the Free Pascal team aims to retain this compatibility in the future.

There are some more restrictions to elements of bitpacked structures:

- The address cannot be retrieved, unless the bit size is a multiple of 8 and the element happens to be stored on a byte boundary.
- An element of a bitpacked structure cannot be used as a var parameter, unless the bit size is a multiple of 8 and the element happens to be stored on a byte boundary.

To determine the size of an element in a bitpacked structure, there is the `BitSizeOf` function. It returns the size – in bits – of the element. For other types or elements of structures which are not bitpacked, this will simply return the size in bytes multiplied by 8, i. e., the return value is then the same as `8*SizeOf`.

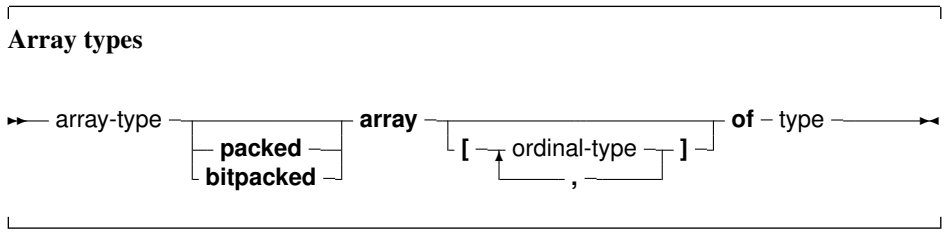
The size of bitpacked records and arrays is limited:

- On 32 bit systems the maximal size is 2^{29} bytes (512 MB).
- On 64 bit systems the maximal size is 2^{61} bytes.

The reason is that the offset of an element must be calculated with the maximum integer size of the system.

3.3.1 Arrays

Free Pascal supports arrays as in Turbo Pascal. Multi-dimensional arrays and (bit)packed arrays are also supported, as well as the dynamic arrays of Delphi:



Static arrays

When the range of the array is included in the array definition, it is called a static array. Trying to access an element with an index that is outside the declared range will generate a run-time error (if range checking is on). The following is an example of a valid array declaration:

```
Type
  RealArray = Array [1..100] of Real;
```

Valid indexes for accessing an element of the array are between 1 and 100, where the borders 1 and 100 are included. As in Turbo Pascal, if the array component type is in itself an array, it is possible to combine the two arrays into one multi-dimensional array. The following declaration:

```
Type
   APoints = array[1..100] of Array[1..3] of Real;
```

is equivalent to the declaration:

```
Type
   APoints = array[1..100,1..3] of Real;
```

The functions `High` and `Low` return the high and low bounds of the leftmost index type of the array. In the above case, this would be 100 and 1. You should use them whenever possible, since it improves maintainability of your code. The use of both functions is just as efficient as using constants, because they are evaluated at compile time.

When static array-type variables are assigned to each other, the contents of the whole array is copied. This is also true for multi-dimensional arrays:

```
program testarray1;

Type
  TA = Array[0..9,0..9] of Integer;

var
  A,B : TA;
  I,J : Integer;
begin
  For I:=0 to 9 do
    For J:=0 to 9 do
      A[I,J]:=I*J;
  For I:=0 to 9 do
    begin
    For J:=0 to 9 do
      Write(A[I,J]:2,' ');
    Writeln;
```

```
    end;
  B:=A;
  Writeln;
  For I:=0 to 9 do
    For J:=0 to 9 do
      A[9-I,9-J]:=I*J;
  For I:=0 to 9 do
    begin
    For J:=0 to 9 do
      Write(B[I,J]:2,' ');
    Writeln;
    end;
end.
```

The output of this program will be two identical matrices.

Dynamic arrays

As of version 1.1, Free Pascal also knows dynamic arrays: In that case the array range is omitted, as in the following example:

```
Type
  TByteArray = Array of Byte;
```

When declaring a variable of a dynamic array type, the initial length of the array is zero. The actual length of the array must be set with the standard `SetLength` function, which will allocate the necessary memory to contain the array elements on the heap.

The following example will set the length to 1000:

```
Var
  A : TByteArray;

begin
  SetLength(A,1000);
```

After a call to `SetLength`, valid array indexes are 0 to 999: the array index is always zero-based.

`SetLength` can also be used for multi-dimensional arrays. The following example will create a "rectangular" array:

```
Var
  A : Array of TByteArray;

begin
  SetLength(A,10,100);
```

After a call to `SetLength`, valid array indexes are 0 to 9 for the first dimension, and 0 to 99 for the second dimension.

In difference with static multi-dimensional arrays, dynamic arrays do not need to be "rectangular", i. e. the various elements can have different lengths:

```
var
  a: array of array of array of LongInt;
  i, j, k: LongInt;
```

```
begin
  SetLength(a, 10, 5);
  SetLength(a[5], 3);

  for i := Low(a) to High(a) do
    for j := Low(a[i]) to High(a[i]) do begin
      SetLength(a[i, j], i * 10 + j);
      for k := Low(a[i, j]) to High(a[i, j]) do
        a[i, j, k] := i * 10000 + j * 100 + k;
    end;

  for i := Low(a) to High(a) do begin
    for j := Low(a[i]) to High(a[i]) do begin
      for k := Low(a[i, j]) to High(a[i, j]) do
        Writeln(a[i, j, k]);
      Writeln('-------');
    end;
    Writeln('=======');
  end;
end.
```

Note that the length of the array is set in elements, not in bytes of allocated memory (although these may be the same). The amount of memory allocated is the size of the array multiplied by the size of 1 element in the array. The memory will be disposed of at the exit of the current procedure or function.

It is also possible to resize the array: in that case, as much of the elements in the array as will fit in the new size, will be kept. The array can be resized to zero, which effectively resets the variable.

At all times, trying to access an element of the array with an index that is not in the current length of the array will generate a run-time error.

Dynamic arrays are reference counted: assignment of one dynamic array-type variable to another will let both variables point to the same array. Contrary to ansistrings, an assignment to an element of one array will be reflected in the other: there is no copy-on-write. Consider the following example:

```
Var
  A,B : TByteArray;

begin
  SetLength(A,10);
  A[0]:=33;
  B:=A;
  A[0]:=31;
```

After the second assignment, the first element in B will also contain 31.

It can also be seen from the output of the following example:

```
program testarray1;

Type
  TA = Array of array of Integer;

var
  A,B : TA;
  I,J : Integer;
```

```
begin
  Setlength(A,10,10);
  For I:=0 to 9 do
    For J:=0 to 9 do
      A[I,J]:=I*J;
  For I:=0 to 9 do
    begin
    For J:=0 to 9 do
      Write(A[I,J]:2,' ');
    Writeln;
    end;
  B:=A;
  Writeln;
  For I:=0 to 9 do
    For J:=0 to 9 do
      A[9-I,9-J]:=I*J;
  For I:=0 to 9 do
    begin
    For J:=0 to 9 do
      Write(B[I,J]:2,' ');
    Writeln;
    end;
end.
```

The output of this program will be a matrix of numbers, and then the same matrix, mirrored.

As remarked earlier, dynamic arrays are reference counted: if in one of the previous examples A goes out of scope and B does not, then the array is not yet disposed of: the reference count of A (and B) is decreased with 1. As soon as the reference count reaches zero the memory, allocated for the contents of the array, is disposed of.

The `SetLength` call will make sure the reference count of the returned array is 1, that is, if two dynamic array variables were pointing to the same memory they will no longer do so after the setlength call:

```
program testunique;

Type
  TA = array of Integer;

var
  A,B : TA;
  I : Integer;

begin
  Setlength(A,10);
  For I:=0 to 9 do
    A[I]:=I;
  B:=A;
  SetLength(B,6);
  A[0]:=123;
  For I:=0 to 5 do
    Writeln(B[I]);
end.
```

It is also possible to copy and/or resize the array with the standard `Copy` function, which acts as the

copy function for strings:

```
program testarray3;

Type
  TA = array of Integer;

var
  A,B : TA;
  I : Integer;

begin
  Setlength(A,10);
  For I:=0 to 9 do
      A[I]:=I;
  B:=Copy(A,3,6);
  For I:=0 to 5 do
    Writeln(B[I]);
end.
```

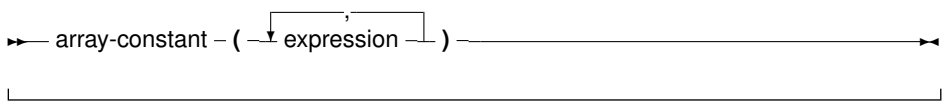
The `Copy` function will copy six elements of the array to a new array. Starting at the element at index 3 (i. e. the fourth element) of the array.

The `Length` function will return the number of elements in the array. The `Low` function on a dynamic array will always return 0, and the `High` function will return the value `Length-1`, i. e., the value of the highest allowed array index.

Array constants

For typed constants or initialized variables that have as type a static or dynamic array, a constant value must be specified. A constant array value is specified using round brackets `(` and `)`, with the elements in between:

Constant array value



As an example:

```
uses sysutils;

var
  A : TBytes = (0,1,2);
  B : Array[1..3] of byte = (0,1,2);

begin
end.
```

For dynamic arrays you can also use an array constructor in code:

```
var
  A : TBytes;

begin
  A:=TBytes.Create(0,1,2);
end.
```

The array constructor cannot be used in an initialized constant.

The constant value for an array cannot be used in an expression, only in a constant or variable declaration. So the following will not work:

```
var
  A : TBytes;

begin
  A:=(0,1,2);
end.
```

Dynamic array Type compatibility

Object Pascal is a strictly typed language. Two technically distinct types are sometimes considered assignment compatible (i. e. a value of one type can be assigned to a variable of another type) under certain circumstances. Dynamic arrays are considered assignment compatible when they use the same element type. That means that the following will compile:

```
{$mode objfpc}

Type
  TA = Array of Integer;
  TB = Array of Integer;

Var
  A : TA;
  B : TB;

begin
  SetLength(A,1);
  A[0]:=1;
  B:=A;
end.
```

But the following will not, even though the integer and word types are assignment compatible:

```
{$mode objfpc}

Type
  TA = Array of Word;
  TB = Array of Integer;

Var
  A : TA;
  B : TB;
```

```
begin
  SetLength(A,1);
  A[0]:=1;
  B:=A;
end.
```

Note that you can always typecast a dynamic array to a pointer.

```
type
  TIntArr = array of LongInt;
  PLongInt = ^LongInt;

var
  xs: TIntArr;
  p: PLongInt;
begin
  setlength(xs, 3);
  xs[0] := 69;
  xs[1] := 10;
  xs[2] := 5;
  p := PLongInt(xs); // <-- cast dynamic array to pointer
  writeln(xs[0], xs[1], xs[2]); // 69105
  writeln(p^, p[1], p[2]); // 69105
end.
```

The opposite typecast is also allowed by the compiler, but must not be used since the automatic referencing and length determination will not work correctly. The following program will print '3' for the length of the byte array, which is of course wrong:

```
type
  TIntArr = array of LongInt;
  PLongInt = ^LongInt;
  TByteArr = Array of Byte;

var
  xs: TIntArr;
  p: PLongInt;
begin
  setlength(xs, 3);
  xs[0] := 69;
  xs[1] := 10;
  xs[2] := 5;
  p := PLongInt(xs);
  Writeln(Length(TByteArr(P))); // Cast to dynamic array
end.
```

Dynamic array constructor

As of version 3.0 of Free Pascal, Dynamic array types have a constructor. This is intrinsic, the compiler provides it. Up to version 2.6.4, the only way to initialize a dynamic array was as follows:

```
Type
  TIntegerArray = Array of Integer;
```

```
var
  A : TIntegerArray;

begin
  SetLength(A,3);
  A[0]:=1;
  A[1]:=2;
  A[3]:=3;
  Writeln(Length(A));
end.
```

As of version 3.0 of Free Pascal, an dynamic array can be initialized using a constructor-like syntax. The constructor is called `Create`, and accepts as parameters a variable number of parameters of the element type of the array type. This means the above initialization can now be done as:

```
Type
  TIntegerArray = Array of Integer;

var
  A : TIntegerArray;

begin
  A:=TIntegerArray.Create(1,2,3);
  Writeln(Length(A));
end.
```

Note that this will not work for dynamic arrays for which no type was created. That is, the following will not work:

```
var
  A : Array of Integer;

begin
  A:=Array of Integer.Create(1,2,3);
  Writeln(Length(A));
end.
```

This approach also works recursively for multi-dimensional arrays:

```
Type
  TIntegerArray = Array of Integer;
  TIntegerArrayArray = Array of TIntegerArray;

var
  A : TIntegerArrayArray;

begin
  A:=TIntegerArrayArray.Create(TIntegerArray.Create(1,2,3),
                               TIntegerArray.Create(4,5,6),
                               TIntegerArray.Create(7,8,9));
  Writeln('Length ',length(A));
end.
```

However, since it is a constructor (code is run at run-time) it is not possible to use this in an initialized variable syntax. That is, the following will not work:

```
Type
  TIntegerArray = Array of Integer;

var
  A : TIntegerArray = TIntegerArray.Create(1,2,3);

begin
  Writeln('Length ',length(A));
end.
```

Dynamic array constant expressions

As of version 3.2 of the compiler, an array can be constructed using an array expression in an explicit assignment or in an initialized variable. However, the expression is different. In an assignment statement, it resembles a set expression, in an initialized variable, the same syntax as for a constant array of fixed length must be used:

```
Type
  TIntegerArray = Array of Integer;

var
  A : TIntegerArray = (1,2,3);
  B : TIntegerArray;

begin
  B:=[3,4,5];
end.
```

Packing and unpacking an array

Arrays can be packed and bitpacked. Two array types which have the same index type and element type, but which are differently packed are not assignment compatible.

However, it is possible to convert a normal array to a bitpacked array with the `pack` routine. The reverse operation is possible as well; a bitpacked array can be converted to a normally packed array using the `unpack` routine, as in the following example:

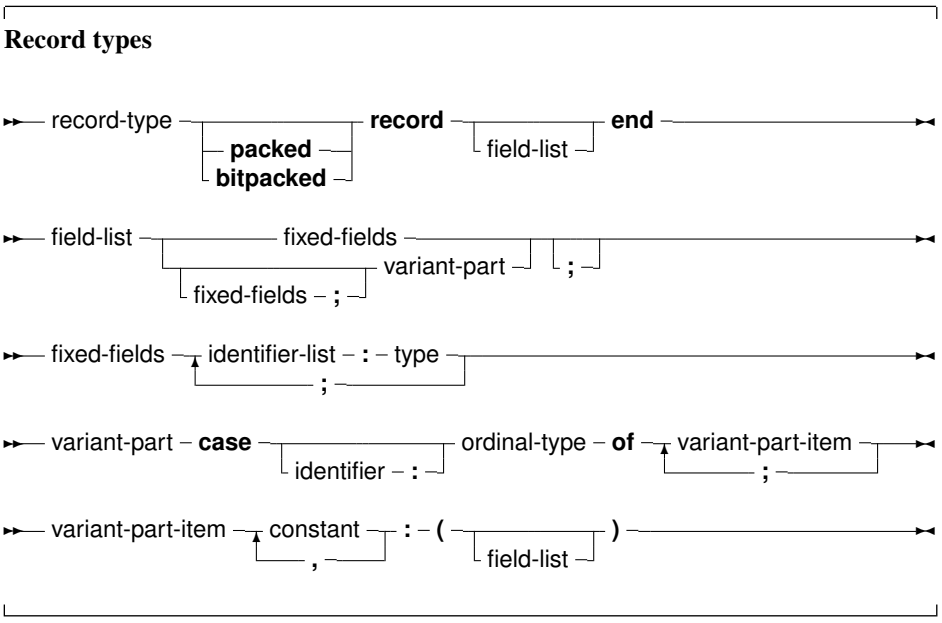
```
Var
  foo : array [ 'a'..'f' ] of Boolean
    = ( false, false, true, false, false, false );
  bar : bitpacked array [ 42..47 ] of Boolean;
  baz : array [ '0'..'5' ] of Boolean;

begin
  pack(foo,'a',bar);
  unpack(bar,baz,'0');
end.
```

More information about the `pack` and `unpack` routines can be found in the system unit reference.

3.3.2 Record types

Free Pascal supports fixed records and records with variant parts. The syntax diagram for a record type is



So the following are valid record type declarations:

```
Type
  Point = Record
          X,Y,Z : Real;
          end;
  RPoint = Record
          Case Boolean of
          False : (X,Y,Z : Real);
          True : (R,theta,phi : Real);
          end;
  BetterRPoint = Record
          Case UsePolar : Boolean of
          False : (X,Y,Z : Real);
          True : (R,theta,phi : Real);
          end;
```

The variant part must be last in the record. The optional identifier in the case statement serves to access the tag field value, which otherwise would be invisible to the programmer. It can be used to see which variant is active at a certain time[3]. In effect, it introduces a new field in the record.

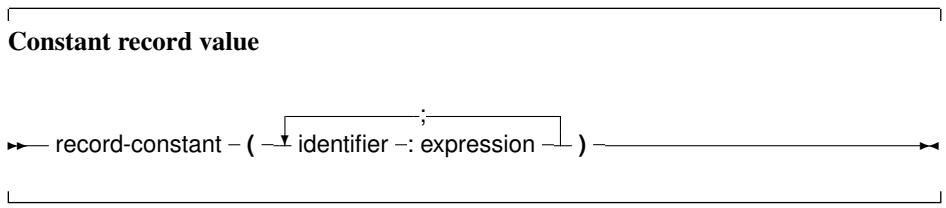
Remark It is possible to nest variant parts, as in:

```
Type
  TMyRec = Record
          X : Longint;
          Case byte of
            2 : (Y : Longint;
                 case byte of
                 3 : (Z : Longint);
                 );
          end;
```

[3]However, it is up to the programmer to maintain this field.

Record-typed Constant

When initializing a variable (or constant) of type record, a special syntax must be used: for each field in the record, the name must be specified, followed by a colon and an expression for the field value. The compiler must be able to evaluate the expression at compile time. The values of the field are separated by a semicolon.



For the `TMyRec` record above, the following would initialize all fields:

```
var
  R1: TMyRec = (x: 1; y: 2; z: 3);
```

You can omit some fields from the end, or all of them:

```
var
  R2: TMyRec = (x: 1; y: 2);
  R3: TMyRec = (x: 1);
  R4: TMyRec = ();
```

In this case, the compiler will display a warning:

```
Warning: Some fields coming after "y" were not initialized
```

In `Delphi` (and `DelphiUnicode`) mode, any fields can be omitted, but the order must be the same:

```
var
  R5: TMyRec = (x: 1; z: 3);
  R6: TMyRec = (y: 2; z: 3);
  R7: TMyRec = (z: 3);
```

The following examples will return an error for any mode:

```
var
  R8: TMyRec = (z: 3; y: 2; x: 1); // ERROR
  R9: TMyRec = (z: 3; x: 1);       // ERROR
```

Record layout and size

The layout and size of a record is influenced by five aspects:

- The size of its fields.
- The alignment requirements of the types of the fields, which are platform-dependent. Note that the alignment requirements of a type inside a record may be different from those of a separate variable of that type. Additionally, the location of a field inside a record may also influence its type's alignment requirements.

- The currently active {$ALIGN N} or {$PACKRECORDS N} setting (these settings override each other, so the last one specified is the active one; note that these directives do not accept exactly the same arguments, see the Programmer's Guide for more information).
- The currently active {$CODEALIGN RECORDMIN=X} setting.
- The currently active {$CODEALIGN RECORDMAX=X} setting.

The layout and size of variant parts in records is determined by replacing them with a field whose type is a record with as first element a field of the tag field type *if an identifier was declared for this tag field*, followed by the elements of the biggest variant.

Field F2's offset in a record is equal to the sum of the previous field F1's offset and F1's size, rounded up to a multiple of F2's required alignment. This required alignment is calculated as follows:

- The required alignment is set to the default alignment of the field's type, possibly adjusted based on the fact that this type occurs in a record and on the field's location in the record.
- If the required alignment is smaller than the currently active {$CODEALIGN RECORDMIN=X} setting, it is changed to this X value.
- If the currently active {$ALIGN N} or {$PACKRECORDS N} setting is
  - a numerical value: if the required alignment is larger than N, it is changed to N. I. e., if N is 1, all fields will be placed right after each other.
  - RESET or DEFAULT: the resulting required alignment is target dependent.
  - C: the required alignment is adjusted according to the rules specified in the official ABI for the current platform.
  - POWER/POWERPC, MAC68K: the alignment value's adjustment is determined by following the official ABI rules for resp. the (Classic) Macintosh PowerPC or Macintosh 680x0 platforms.

The size of a record is equal to the sum of the record's last field's offset and this field's size, rounded up to a multiple of the record's required alignment. The record's required alignment is calculated as follows:

- The required alignment is set to the alignment of the record's field with the largest alignment, as determined while laying out the record.
- If the current {$ALIGN N} or {$PACKRECORDS N} setting is different from C and the required alignment is larger than than the currently active {$CODEALIGN RECORDMAX=X}, the required alignment is changed to X.
- If the current {$ALIGN N} or {$PACKRECORDS N} setting is equal to C, the required alignment is determined by following the official ABI rules.

Remarks and examples

Free Pascal also supports a "packed record", which is a record where all the elements are byte-aligned. As a result, the two following declarations are equivalent:

```
{$PackRecords 1}
Trec2 = Record
  A : Byte;
  B : Word;
  end;
{$PackRecords default}
```

and

```
Trec2 = Packed Record
  A : Byte;
  B : Word;
  end;
```

Note the `{$PackRecords Default}` after the first declaration to restore the default setting!

Given the platform-dependent nature of how records are laid out in memory, the only way to ensure a compatible layout across platforms (assuming that all fields are declared using a type with the same meaning across these same platforms) is by using `{$PACKRECORDS 1}`.

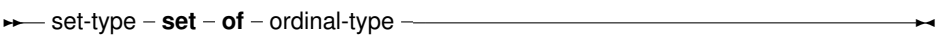
In particular, if a typed file with records, produced by a Turbo Pascal program, must be read, then chances are that attempting to read that file correctly will fail. The reason is that Free Pascal's default `{$PACKRECORDS N}` setting is not necessarily compatible with Turbo Pascal's. It can be changed to `{$PACKRECORDS 1}` or `{$PACKRECORDS 2}` depending on the setting used in the Turbo Pascal program that create the file (although it may still fail with `{$PACKRECORDS 2}` due to different type alignment requirements between 16 bit MSDOS and your current platform).

The same remark goes for Delphi: exchanging data is only guaranteed to be possible if both the producer and consumer use a packed record, or if they are on the same platform and use the same `{$PACKRECORDS X}` setting.

3.3.3 Set types

Free Pascal supports the set types as in Turbo Pascal. The prototype of a set declaration is:

Set Types



Each of the elements of `SetType` must be of type `TargetType`. `TargetType` can be any ordinal type with a range between `0` and `255`. A set can contain at most `256` elements. The following are valid set declaration:

```
Type
  Junk = Set of Char;
  Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

Var
  WorkDays : Set of days;
```

Given these declarations, the following assignment is legal:

```
WorkDays := [Mon, Tue, Wed, Thu, Fri];
```

Several operations can be done on sets: taking unions or differences, adding or removing elements, comparisons. These are documented in section 12.9.6, page 190

How the compiler stores sets depends on the mode and can be controlled with a directive. For more information, see the Programmer's Guide.

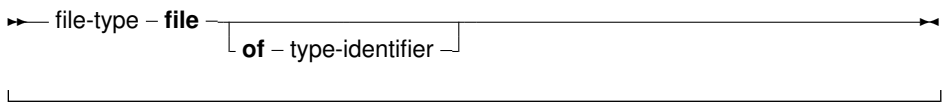
Note that if you use an enumeration with assigned values, if the upper bound is larger than 256, you cannot construct a set with it.

3.3.4 File types

File types are types that store a sequence of some base type, which can be any type except another file type. It can contain (in principle) an infinite number of elements. File types are used commonly to store data on disk. However, nothing prevents the programmer, from writing a file driver that stores its data for instance in memory.

Here is the type declaration for a file type:

File types



If no type identifier is given, then the file is an untyped file; it can be considered as equivalent to a file of bytes. Untyped files require special commands to act on them (see `Blockread`, `Blockwrite`). The following declaration declares a file of records:

```
Type
  Point = Record
    X,Y,Z : real;
    end;
  PointFile = File of Point;
```

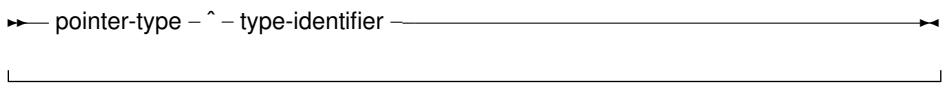
Internally, files are represented by the `FileRec` record, which is declared in the Dos or SysUtils units.

A special file type is the `Text` file type, represented by the `TextRec` record. A file of type `Text` uses special input-output routines. The default `Input`, `Output` and `StdErr` file types are defined in the system unit: they are all of type `Text`, and are opened by the system unit initialization code.

3.4 Pointers

Free Pascal supports the use of pointers. A variable of the pointer type contains an address in memory, where the data of another variable may be stored. A pointer type can be defined as follows:

Pointer types



As can be seen from this diagram, pointers are typed, which means that they point to a particular kind of data. The type of this data must be known at compile time.

Dereferencing the pointer (denoted by adding ^ after the variable name) behaves then like a variable. This variable has the type declared in the pointer declaration, and the variable is stored in the address that is pointed to by the pointer variable. Consider the following example:

```
Program pointers;
type
```

```
  Buffer = String[255];
  BufPtr = ^Buffer;
Var B  : Buffer;
    BP : BufPtr;
    PP : Pointer;
etc..
```

In this example, `BP` *is a pointer to* a `Buffer` type; while `B` *is* a variable of type `Buffer`. `B` takes 256 bytes memory, and `BP` only takes 4 (or 8) bytes of memory: enough memory to store an address.

The expression

```
BP^
```

is known as the dereferencing of `BP`. The result is of type `Buffer`, so

```
BP^[23]
```

Denotes the 23rd character in the string pointed to by `BP`.

Remark Free Pascal treats pointers much the same way as C does. This means that a pointer to some type can be treated as being an array of this type.

From this point of view, the pointer then points to the zeroeth element of this array. Thus the following pointer declaration

```
Var p : ^Longint;
```

can be considered equivalent to the following array declaration:

```
Var p : array[0..Infinity] of Longint;
```

The difference is that the former declaration allocates memory for the pointer only (not for the array), and the second declaration allocates memory for the entire array. If the former is used, the memory must be allocated manually, using the `Getmem` function. The reference `P^` is then the same as `p[0]`. The following program illustrates this maybe more clear:

```
program PointerArray;
var i : Longint;
    p : ^Longint;
    pp : array[0..100] of Longint;
begin
  for i := 0 to 100 do pp[i] := i; { Fill array }
  p := @pp[0];                     { Let p point to pp }
  for i := 0 to 100 do
    if p[i]<>pp[i] then
      WriteLn ('Ohoh, problem !')
end.
```

Free Pascal supports pointer arithmetic as C does. This means that, if `P` is a typed pointer, the instructions

```
Inc(P);
Dec(P);
```

will increase, respectively decrease the address the pointer points to with the size of the type `P` is a pointer to. For example

```
Var P : ^Longint;
...
 Inc (p);
```

will increase `P` by 4, because 4 is the size of a longint. If the pointer is untyped, a size of 1 byte is assumed (i. e. as if the pointer were a pointer to a byte: `^byte`.)

Normal arithmetic operators on pointers can also be used, that is, the following are valid pointer arithmetic operations:

```
var  p1,p2 : ^Longint;
     L : Longint;
begin
  P1 := @P2;
  P2 := @L;
  L := P1-P2;
  P1 := P1-4;
  P2 := P2+4;
end.
```

Here, the value that is added or subtracted *is* multiplied by the size of the type the pointer points to. In the previous example `P1` will be decremented by 16 bytes, and `P2` will be incremented by 16.

3.5 Forward type declarations

Programs often need to maintain a linked list of records. Each record then contains a pointer to the next record (and possibly to the previous record as well). For type safety, it is best to define this pointer as a typed pointer, so the next record can be allocated on the heap using the `New` call. In order to do so, the record should be defined something like this:

```
Type
  TListItem = Record
    Data : Integer;
    Next : ^TListItem;
  end;
```

When trying to compile this, the compiler will complain that the `TListItem` type is not yet defined when it encounters the `Next` declaration: This is correct, as the definition is still being parsed.

To be able to have the `Next` element as a typed pointer, a "Forward type declaration" must be introduced:

```
Type
  PListItem = ^TListItem;
  TListItem = Record
    Data : Integer;
    Next : PTListItem;
  end;
```

When the compiler encounters a typed pointer declaration where the referenced type is not yet known, it postpones resolving the reference till later. The pointer definition is a "Forward type declaration".

The referenced type should be introduced later in the same `Type` block. No other block may come between the definition of the pointer type and the referenced type. Indeed, even the word `Type`

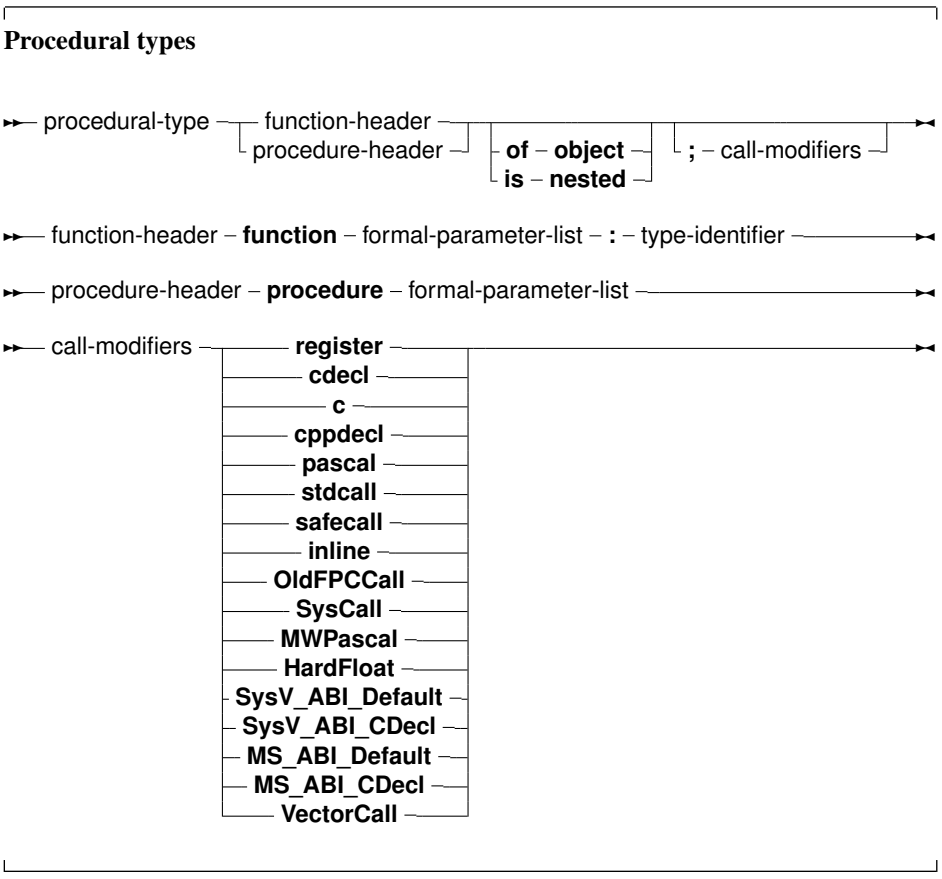
itself may not re-appear: in effect it would start a new type-block, causing the compiler to resolve all pending declarations in the current block.

In most cases, the definition of the referenced type will follow immediately after the definition of the pointer type, as shown in the above listing. The forward defined type can be used in any type definition following its declaration.

Note that a forward type declaration is only possible with pointer types and classes, not with other types.

3.6 Procedural types

Free Pascal has support for procedural types, although it differs a little from the Turbo Pascal or Delphi implementation of them. The type declaration remains the same, as can be seen in the following syntax diagram:



For a description of formal parameter lists, see chapter 14, page 219. The two following examples are valid type declarations:

```
Type TOneArg = Procedure (Var X : integer);
     TNoArg = Function : Real;
var proc : TOneArg;
    func : TNoArg;
```

One can assign the following values to a procedural type variable:

1. `Nil`, for both normal procedure pointers and method pointers.
2. A variable reference of a procedural type, i. e. another variable of the same type.
3. A global procedure or function address, with matching function or procedure header and calling convention.
4. A method address.

Given these declarations, the following assignments are valid:

```
Procedure printit (Var X : Integer);
begin
  WriteLn (x);
end;
...
Proc := @printit;
Func := @Pi;
```

From this example, the difference with Turbo Pascal is clear: In Turbo Pascal it isn't necessary to use the address operator (`@`) when assigning a procedural type variable, whereas in Free Pascal it is required. In case the `-MDelphi` or `-MTP` switches are used, the address operator can be dropped.

Remark The modifiers concerning the calling conventions must be the same as the declaration; i. e. the following code would give an error:

```
Type TOneArgCcall = Procedure (Var X : integer);cdecl;
var proc : TOneArgCcall;
Procedure printit (Var X : Integer);
begin
  WriteLn (x);
end;
begin
Proc := @printit;
end.
```

Because the `TOneArgCcall` type is a procedure that uses the cdecl calling convention.

In case the `is nested` modified is added, then the procedural variable can be used with nested procedures. This requires that the sources be compiled in macpas or ISO mode, or that the `nestedprocvars` modeswitch be activated:

```
{$modeswitch nestedprocvars}
program tmaclocalprocparam3;

type
  tnestedprocvar = procedure is nested;

var
  tempp: tnestedprocvar;

procedure p1( pp: tnestedprocvar);
begin
  tempp:=pp;
  tempp
end;
```

```
procedure p2( pp: tnestedprocvar);
var
  localpp: tnestedprocvar;
begin
  localpp:=pp;
  p1( localpp)
end;

procedure n;
begin
  writeln( 'calling through n')
end;

procedure q;

var qi: longint;

  procedure r;
  begin
    if qi = 1 then
      writeln( 'success for r')
    else
      begin
      writeln( 'fail');
      halt( 1)
    end
  end;

begin
  qi:= 1;
  p1( @r);
  p2( @r);
  p1( @n);
  p2( @n);
end;

begin
  q;
end.
```

In case one wishes to assign methods of a class to a variable of procedural type, the procedural type must be declared with the `of object` modifier.

The two following examples are valid type declarations for method procedural variables (also known as event handlers because of their use in GUI design):

```
Type TOneArg = Procedure (Var X : integer) of object;
     TNoArg = Function : Real of object;
var
  oproc : TOneArg;
  ofunc : TNoArg;
```

A method of the correct signature can be assigned to these functions. When called, `Self` will be pointing to the instance of the object that was used to assign the method procedure.

The following object methods can be assigned to `oproc` and `ofunc`:

```
Type
  TMyObject = Class(TObject)
    Procedure DoX (Var X : integer);
    Function DoY: Real;
  end;

Var
  M : TMyObject;

begin
  oproc:=@M.DoX;
  ofunc:=@M.DOY;
end;
```

When calling `oproc` and `ofunc`, `Self` will equal `M`.

This mechanism is sometimes called `Delegation`.

Remark When comparing two variables of type method, only the method's address is compared, not the instance pointer. That means that the following program will print `True`:

```
Type
  TSomeMethod = Procedure  of object;

  TMyObject = Class(TObject)
    Procedure DoSomething;
  end;

Procedure TMyObject.DoSomething;

begin
  Writeln('In DoSomething');
end;

var
  X,Y : TMyObject;
  P1,P2 : TSomeMethod;

begin
  X:=TMyObject.Create;
  Y:=TMyObject.Create;
  P1:=@X.DoSomething;
  P2:=@Y.DoSomething;
  Writeln('Same method : ',P1=P2);
end.
```

If both pointers must be compared, a typecast to `TMethod` must be done, and both pointers should be compared. `TMethod` is defined in the system unit as follows:

```
TMethod = record
  Code : CodePointer;
  Data : Pointer;
end;
```

The following program will therefore print False:

```
Type
  TSomeMethod = Procedure  of object;

  TMyObject = Class(TObject)
    Procedure DoSomething;
  end;

Procedure TMyObject.DoSomething;

begin
  Writeln('In DoSomething');
end;

var
  X,Y : TMyObject;
  P1,P2 : TMethod;

begin
  X:=TMyObject.Create;
  Y:=TMyObject.Create;
  P1:=TMethod(@X.DoSomething);
  P2:=TMethod(@Y.DoSomething);
  Writeln('Same method : ',(P1.Data=P2.Data) and (P1.Code=P1.Code));
end.
```

3.7 Variant types

3.7.1 Definition

As of version 1.1, FPC has support for variants. For maximum variant support it is recommended to add the variants unit to the uses clause of every unit that uses variants in some way: the variants unit contains support for examining and transforming variants other than the default support offered by the System or `ObjPas` units.

The type of a value stored in a variant is only determined at runtime: it depends what has been assigned to the variant. Almost any simple type can be assigned to variants: ordinal types, string types, int64 types.

Structured types such as sets, records, arrays, files, objects and classes are not assignment-compatible with a variant, as well as pointers. Interfaces and COM or CORBA objects can be assigned to a variant (basically because they are simply a pointer).

This means that the following assignments are valid:

```
Type
  TMyEnum = (One,Two,Three);

Var
  V : Variant;
  I : Integer;
  B : Byte;
  W : Word;
  Q : Int64;
```

```
  E : Extended;
  D : Double;
  En : TMyEnum;
  AS : AnsiString;
  WS : WideString;

begin
  V:=I;
  V:=B;
  V:=W;
  V:=Q;
  V:=E;
  V:=En;
  V:=D;
  V:=AS;
  V:=WS;
end;
```

And of course vice-versa as well.

A variant can hold an array of values: All elements in the array have the same type (but can be of type "variant"). For a variant that contains an array, the variant can be indexed:

```
Program testv;

uses variants;

Var
  A : Variant;
  I : integer;

begin
  A:=VarArrayCreate([1,10],varInteger);
  For I:=1 to 10 do
    A[I]:=I;
end.
```

For the explanation of `VarArrayCreate`, see Unit Reference.

Note that when the array contains a string, this is not considered an "array of characters", and so the variant cannot be indexed to retrieve a character at a certain position in the string.

3.7.2 Variants in assignments and expressions

As can be seen from the definition above, most simple types can be assigned to a variant. Likewise, a variant can be assigned to a simple type: If possible, the value of the variant will be converted to the type that is being assigned to. This may fail: Assigning a variant containing a string to an integer will fail unless the string represents a valid integer. In the following example, the first assignment will work, the second will fail:

```
program testv3;

uses Variants;

Var
```

```
  V : Variant;
  I : Integer;

begin
  V:='100';
  I:=V;
  Writeln('I : ',I);
  V:='Something else';
  I:=V;
  Writeln('I : ',I);
end.
```

The first assignment will work, but the second will not, as `Something else` cannot be converted to a valid integer value. An `EConvertError` exception will be the result.

The result of an expression involving a variant will be of type variant again, but this can be assigned to a variable of a different type – if the result can be converted to a variable of this type.

Note that expressions involving variants take more time to be evaluated, and should therefore be used with caution. If a lot of calculations need to be made, it is best to avoid the use of variants.

When considering implicit type conversions (e. g. byte to integer, integer to double, char to string) the compiler will ignore variants unless a variant appears explicitly in the expression.

3.7.3 Variants and interfaces

Remark Dispatch interface support for variants is currently broken in the compiler.

Variants can contain a reference to an interface – a normal interface (descending from `IInterface`) or a dispatchinterface (descending from `IDispatch`). Variants containing a reference to a dispatch interface can be used to control the object behind it: the compiler will use late binding to perform the call to the dispatch interface: there will be no run-time checking of the function names and parameters or arguments given to the functions. The result type is also not checked. The compiler will simply insert code to make the dispatch call and retrieve the result.

This means basically, that you can do the following on Windows:

```
Var
  W : Variant;
  V : String;

begin
  W:=CreateOleObject('Word.Application');
  V:=W.Application.Version;
  Writeln('Installed version of MS Word is : ',V);
end;
```

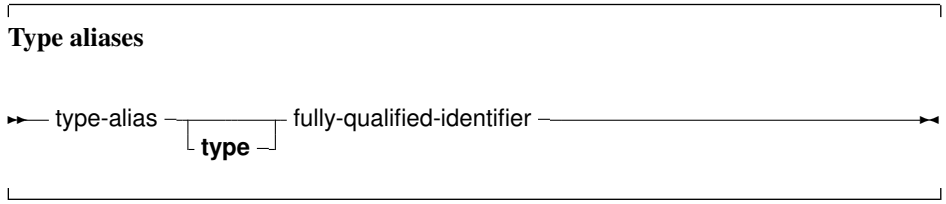
The line

```
  V:=W.Application.Version;
```

is executed by inserting the necessary code to query the dispatch interface stored in the variant `W`, and execute the call if the needed dispatch information is found.

3.8 Type aliases

Type aliases are a way to give another name to a type, but can also be used to create real new types. Which of the two depends on the way the type alias is defined:



The first case is just a means to give another name to a type:

```
Type
  MyInteger = Integer;
```

This creates a new name to refer to the `Integer` type, but does not create an actual new type. That is, two variables:

```
Var
  A : MyInteger;
  B : Integer;
```

Will actually have the exact same type from the point of view of the compiler (namely: `Integer`). The definition of `A` is identical to

```
Var
  A : Integer;
```

The above presents a way to make types platform independent, by only using the alias types, and then defining these types for each platform individually. Any programmer who then uses these custom types doesn't have to worry about the underlying type size: it is opaque to him. It also allows to use shortcut names for fully qualified type names. e. g. define `system.longint` as `Olongint` and then redefine `longint`.

The alias is frequently seen to re-expose a type:

```
Unit A;

Interface

Uses B;

Type
  MyType = B.MyType;
```

This construction is often seen after some refactoring, when moving some declarations from unit `A` to unit `B`, to preserve backwards compatibility of the interface of unit `A`.

The second case is slightly more subtle:

```
Type
  MyInteger = Type Integer;
```

This not only creates a new name to refer to the `Integer` type, but actually creates a new type. That is, two variables:

```
Var
  A : MyInteger;
  B :  Integer;
```

Will not have the same type from the point of view of the compiler. However, these two types will be assignment compatible. That means that an assignment

```
  A:=B;
```

will work.

The difference can be seen when examining type information:

```
If TypeInfo(MyInteger)<>TypeInfo(Integer) then
  Writeln('MyInteger and Integer are different types');
```

The compiler function `TypeInfo` returns a pointer to the type information in the binary. Since the two types `MyInteger` and `Integer` are different, they will generate different type information blocks, and the pointers will differ.

There are three consequences of having different types:

1. That they have different typeinfo, hence different RTTI (Run-Time Type Information).
2. They can be used in function overloads, that is

   ```
   Procedure MyProc(A : MyInteger); overload;
   Procedure MyProc(A : Integer); overload;
   ```

   will work. This will not work with a simple type alias.
3. They can be used in operator overloads, that is

   ```
   Operator +(A,B : MyInteger) : MyInteger;
   ```

   will work too.

3.9 Managed types

By default, pascal types are unmanaged. That means that variables must be explicitly initialized, finalized, memory allocated and so on. However, in Object Pascal, several types are managed, which means that the compiler initializes and finalizes variables of this type: this is necessary, for instance for reference counted data types.

The following types are managed:

AnsiString They are initialized to `Nil`.

UnicodeString They are initialized to `Nil`.

WideString They are initialized to `Nil`.

Interface They are initialized to `Nil`.

Dynamic arrays They are initialized to `Nil`.

Variants They are initialized to the empty value (`VarIsEmpty` will return `True`).

And any record or array whose elements contain managed types.

Class instances containing managed types are also initialized, but the class instance pointer itself is not.

Variables of managed types will also be finalized: this means, in general, that their reference count will be decreased at the latest at the end of the current scope.

Remark Note that no assumption should be made about the exact time of this finalization. All that is guaranteed that they are finalized when they go out of scope.

Chapter 4

Variables

4.1 Definition

Variables are explicitly named memory locations with a certain type. When assigning values to variables, the Free Pascal compiler generates machine code to move the value to the memory location reserved for this variable. Where this variable is stored depends on where it is declared:

- Global variables are variables declared in a unit or program, but not inside a procedure or function. They are stored in fixed memory locations, and are available during the whole execution time of the program.
- Local variables are declared inside a procedure or function. Their value is stored on the program stack, i. e. not at fixed locations.

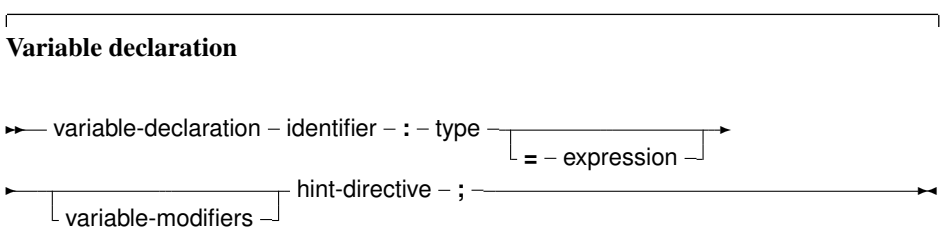
The Free Pascal compiler handles the allocation of these memory locations transparently, although this location can be influenced in the declaration.

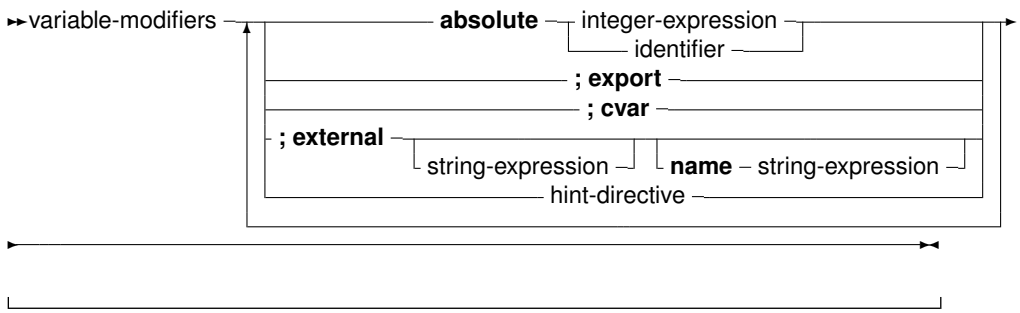
The Free Pascal compiler also handles reading values from or writing values to the variables transparently. But even this can be explicitly handled by the programmer when using properties.

Variables must be explicitly declared when they are needed. No memory is allocated unless a variable is declared. Using a variable identifier (for instance, a loop variable) which is not declared first, is an error which will be reported by the compiler.

4.2 Declaration

The variables must be declared in a variable declaration block of a unit or a procedure or function (section 16.5, page 266). It looks as follows:





This means that the following are valid variable declarations:

```
Var
  curterm1 : integer;

  curterm2 : integer; cvar;
  curterm3 : integer; cvar; external;

  curterm4 : integer; external name 'curterm3';
  curterm5 : integer; external 'libc' name 'curterm9';

  curterm6 : integer absolute curterm1;

  curterm7 : integer; cvar;  export;
  curterm8 : integer; cvar;  public;
  curterm9 : integer; export name 'me';
  curterm10 : integer; public name 'ma';

  curterm11 : integer = 1 ;

  curterm12 : integer absolute $00123456;
  curterm13 : integer absolute @curterm12+8;

  curterm14 : integer absolute 'a';
```

The difference between these declarations is as follows:

1. The first form (`curterm1`) defines a regular variable. The compiler manages everything by itself.
2. The second form (`curterm2`) declares also a regular variable, but specifies that the assembler name for this variable equals the name of the variable as written in the source, using C mangling of the assembler name (meaning it gets prefixed with a _ character on some platforms).
3. The third form (`curterm3`) declares a variable which is located externally: the compiler will assume memory is located elsewhere, and that the assembler name of this location is specified by the name of the variable, as written in the source. The name may not be specified.
4. The fourth form is completely equivalent to the third, it declares a variable which is stored externally, and explicitly gives the assembler name of the location. If `cvar` is not used, the name must be specified.
5. The fifth form is a variant of the fourth form, only the name of the library in which the memory is reserved is specified as well.

6. The sixth form declares a variable (`curterm6`), and tells the compiler that it is stored in the same location as another variable (`curterm1`).
7. The seventh form declares a variable (`curterm7`), and tells the compiler that the assembler label of this variable should be determined as the C compiler for the current platform would determine it, additionally it must be made public. i. e. it can be referenced from other object files.
8. The eighth form (`curterm8`) is equivalent to the seventh: "public" is an alias for "export".
9. The ninth and tenth form are equivalent: they specify the assembler name of the variable.
10. the eleventh form declares a variable (`curterm11`) and initializes it with a value (1 in the above case).
11. The twelfth form declares a variable (curterm6a), and tells the compiler that it is stored at the absolute address $00123456 (hexadecimal).
12. The thirteenth form uses an integer expression that resolves to a constant value. Note that you can take an address of a previously defined variable as base offset.
13. the fourteenth form shows you can use an assembler name (label) as address.

Note that assembler names must be unique. It's not possible to declare or export two variables with the same assembler name. In particular, do not attempt to export variables with a public name that starts with `FPC_`; the compiler uses some internal system routines with this name.

As for any identifier, variables can be accompagnied by hint directives (section 1.5, page 17):

```
var
  curold : integer deprecated;
```

4.3 Scope

Variables, just as any identifier, obey the general rules of scope. In addition, initialized variables are initialized when they enter scope:

- Global initialized variables are initialized once, when the program starts.
- Local initialized variables are initialized each time the procedure is entered.

Note that the behavior for local initialized variables is different from the one of a local typed constant. A local typed constant behaves like a global initialized variable.

4.4 Initialized variables

By default, simple variables in Pascal are not initialized after their declaration. Any assumption that they contain zero or any other default value is erroneous: They can contain rubbish. To remedy this, the concept of initialized variables exists. The difference with normal variables is that their declaration includes an initial value, as can be seen in the diagram in the previous section.

Remark Two exceptions to this rule exist:

1. Managed types are an exception to this rule: Managed types are always initialized with a default value: in general this means setting the reference count to zero, or setting the pointer value of the type to `Nil`. See section 3.9, page 71

2. Global variables are initialized with the equivalent of zero.

Note that the behavior of zeroing out certain variables can result in invalid content for variables:

```
Type
  TWeekDays =
    (monday,tuesday,wednesday,thursday,friday,saturday,sunday);
  TWeekend = saturday..sunday;

var
  W : TWeekend;

begin
  Writeln(W);
end.
```

The above will result, when run, in an error:

```
Runtime error 107 at $000000000040024A
$000000000040024A
$000000000042BF70
$00000000004001D2
```

Therefore it is *Highly recommended* to always initialize variables before using them.

This can be easily done in the declaration of the variables. Given the declaration:

```
Var
  S : String = 'This is an initialized string';
```

The value of the variable following will be initialized with the provided value. The following is an even better way of doing this:

```
Const
  SDefault = 'This is an initialized string';

Var
  S : String = SDefault;
```

Initialization is often used to initialize arrays and records. For arrays, the initialized elements must be specified, surrounded by round brackets, and separated by commas. The number of initialized elements must be exactly the same as the number of elements in the declaration of the type. As an example:

```
Var
  tt : array [1..3] of string[20] = ('ikke', 'gij', 'hij');
  ti : array [1..3] of Longint = (1,2,3);
```

For constant records, each element of the record that you wish to initialize must be specified in the form `Field: Value`, separated by semicolons, and surrounded by round brackets.. You can omit fields that you don't wish to initialize, in fact you can skip all fields. If you skip fields, the compiler will emit a warning.

As an example:

```
Type
  Point = record
    X,Y : Real
    end;
Var
  Origin : Point = (X:0.0; Y:0.0);
  Partial : Point = (X:0.0);
  Empty : Point = ();
```

The above declarations will result in the following warnings:

```
iv.pp(7,27) Warning: Some fields coming after "X" were not initialized
iv.pp(8,20) Warning: Some fields coming after "" were not initialized
```

The order of the fields in a constant record needs to be the same as in the type declaration, otherwise a compile-time error will occur.

Remark It should be stressed that initialized variables are initialized when they come into scope, in difference with typed constants, which are initialized at program start. This is also true for *local* initialized variables. Local initialized variables are initialized whenever the routine is called. Any changes that occurred in the previous invocation of the routine will be undone, because they are again initialized.

Remark Care should be taken when using initialized pointer types such as `PChars`. In the following examples, `S` is a pointer, pointing to a block of constant (read-only) program data. Assigning a character in the string will therefore not work. Assigning `S` itself will of course work. The first routine will give an error, the second not:

```
procedure foo1;
var
  s: PChar = 'PChar';
begin
  s[0] := 'a';
end;

procedure foo2;
var
  s: PChar;
begin
  s := 'PChar';
  s[0] := 'a';
end;
```

4.5 Initializing variables using default

Some variables must be initialized because they contain managed types. For variables that are declared in the `var` section of a function or in the main program, this happens automatically. For variables that are allocated on the heap, this is not necessarily the case.

For this, the compiler contains the `Default` intrinsic. This function accepts a type identifier as the argument, and will return a correctly initialized variable of that type. In essence, it will zero out the whole variable.

The following gives an example of its use:

```
type
```

```
  TRecord = record
    i: LongInt;
    s: AnsiString;
  end;

var
  i: LongInt;
  o: TObject;
  r: TRecord;
begin
  i := Default(LongInt); // 0
  o := Default(TObject); // Nil
  r := Default(TRecord); // ( i: 0; s: '')
end.
```

The case where a variable is allocated on the heap, is more interesting:

```
type
  TRecord = record
    i: LongInt;
    s: AnsiString;
  end;

var
  i: ^LongInt;
  o: ^TObject;
  r: ^TRecord;
begin
  i:=GetMem(SizeOf(Longint));
  i^ := Default(LongInt); // 0
  o:=GetMem(SizeOf(TObject));
  o^ := Default(TObject); // Nil
  r:=GetMem(SizeOf(TRecord));
  r^ := Default(TRecord); // ( i: 0; s: '')
end.
```

It works for all types, except the various file types (or complex types containing a file type).

Remark

- For generics, the use of `Default` is especially useful, since the type of a variable may not be known during the declaration of a generic. For more information section 8.8, page 145.
- Function results are available as a `Result` identifier, and as such resemble variables. They are not variables, but are treated as passed-by-reference parameters. They are therefore not initialized.

4.6 Thread Variables

For a program which uses threads, the variables can be really global, i. e. the same for all threads, or thread-local: this means that each thread gets a copy of the variable. Local variables (defined inside a procedure) are always thread-local. Global variables are normally the same for all threads. A global variable can be declared thread-local by replacing the `var` keyword at the start of the variable declaration block with `Threadvar`:

```
Threadvar
  IOResult : Integer;
```

If no threads are used, the variable behaves as an ordinary variable. If threads are used then a copy is made for each thread (including the main thread). Note that the copy is made with the original value of the variable, *not* with the value of the variable at the time the thread is started.

Threadvars should be used sparingly: There is an overhead for retrieving or setting the variable's value. If possible at all, consider using local variables; they are always faster than thread variables.

Threads are not enabled by default. For more information about programming threads, see the chapter on threads in the Programmer's Guide.

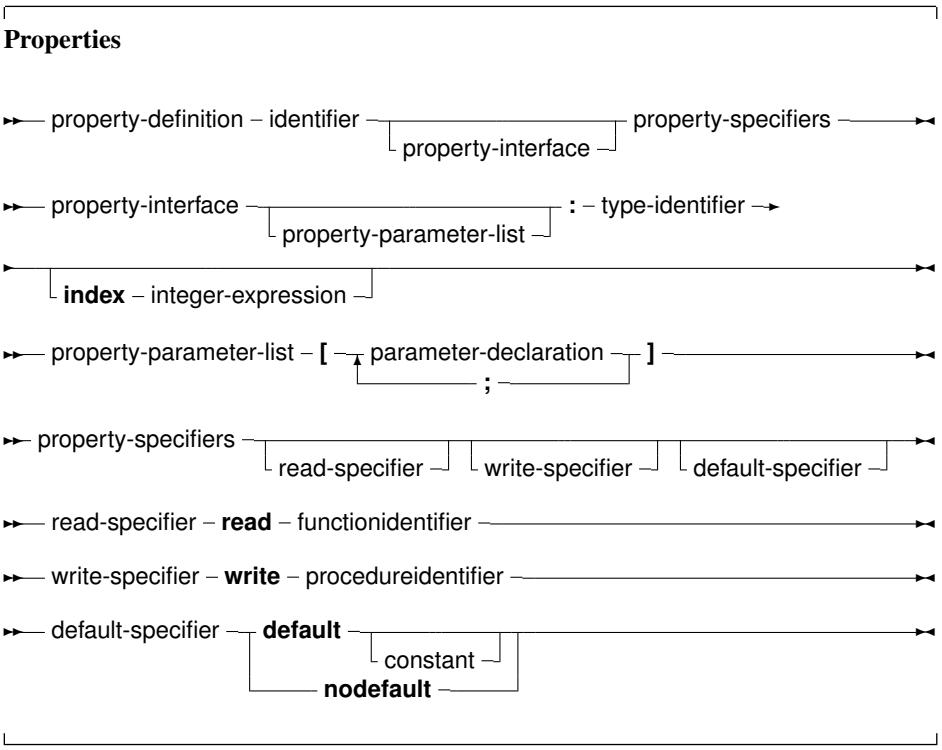
4.7 Properties

A global block can declare properties, just as they could be defined in a class. The difference is that the global property does not need a class instance: there is only one instance of this property. Other than that, a global property behaves like a class property. The read/write specifiers for the global property must also be regular procedures, not methods.

The concept of a global property is specific to Free Pascal, and does not exist in Delphi. `ObjFPC` mode is required to work with properties.

The concept of a global property can be used to "hide" the location of the value, or to calculate the value on the fly, or to check the values which are written to the property.

The declaration is as follows:



Note that you cannot have a field in the accessor (read/write) specifiers, it must be a function or procedure.

The following is an example:

```
{$mode objfpc}
unit testprop;

Interface

Function GetMyInt : Integer;
Procedure SetMyInt(Value : Integer);

Property
  MyProp : Integer Read GetMyInt Write SetMyInt;

Implementation

Uses sysutils;

Var
  FMyInt : Integer;

Function GetMyInt : Integer;

begin
  Result:=FMyInt;
end;

Procedure SetMyInt(Value : Integer);

begin
  If ((Value mod 2)=1) then
    Raise Exception.Create('MyProp can only contain even value');
  FMyInt:=Value;
end;

end.
```

The read/write specifiers must be in the same unit as the unit which declared the property.

More information about properties can be found in chapter 6, page 92.

Chapter 5

Objects

5.1 Declaration

Free Pascal supports object oriented programming. In fact, most of the compiler is written using objects. Here we present some technical questions regarding object oriented programming in Free Pascal.

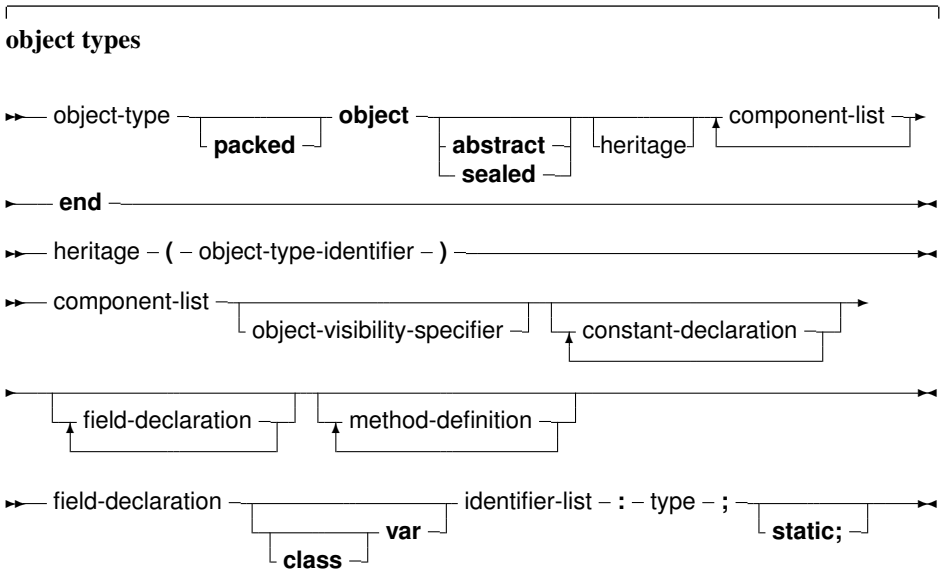
Objects should be treated as a special kind of record. The record contains all the fields that are declared in the objects definition, and pointers to the methods that are associated to the objects' type.

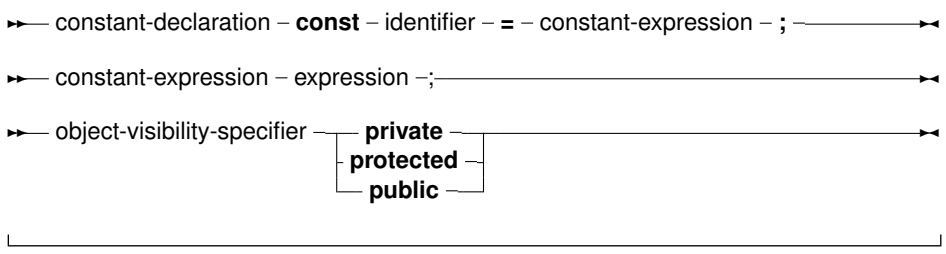
An object is declared just as a record would be declared; except that now, procedures and functions can be declared as if they were part of the record. Objects can "inherit" fields and methods from "parent" objects. This means that these fields and methods can be used as if they were included in the objects declared as a "child" object.

Furthermore, a concept of visibility is introduced: fields, procedures and functions can be declared as `public`, `protected` or `private`. By default, fields and methods are `public`, and are exported outside the current unit.

Fields or methods that are declared `private` are only accessible in the current unit: their scope is limited to the implementation of the current unit.

The prototype declaration of an object is as follows:





As can be seen, any visibility block can be specified multiple times, as often as needed.

The following is a valid definition of an object:

```
Type
  TObj = object
  Private
    Caption : ShortString;
  Public
    Constructor init;
    Destructor done;
    Procedure SetCaption (AValue : String);
    Function GetCaption : String;
  end;
```

It contains a constructor/destructor pair, and a method to get and set a caption. The `Caption` field is private to the object: it cannot be accessed outside the unit in which `TObj` is declared.

Remark In MacPas mode, the `Object` keyword is replaced by the `class` keyword for compatibility with other pascal compilers available on the Mac. That means that objects cannot be used in MacPas mode.

Remark Free Pascal also supports the packed object. This is the same as an object, only the elements (fields) of the object are byte-aligned, just as in the packed record. The declaration of a packed object is similar to the declaration of a packed record:

```
Type
  TObj = packed object
   Constructor init;
   ...
   end;
  Pobj = ^TObj;
Var PP : Pobj;
```

Similarly, the `{$PackRecords }` directive acts on objects as well.

5.2 Abstract and sealed objects

An object can be declared as sealed. In that case, it is not possible to declare a descendent object. The compiler will return an error if it encounters a declaration of a descendent:

```
Type
  TMyObject = object Sealed
    x : integer;
  end;
```

```
  TMyObject2 = object(TMyObject)
    Y : Integer;
  end;

begin
end.
```

This will result in the following error:

```
Error: Cannot create a descendant of the sealed class "TMyObject"
```

The 'class' is not an typo, the compiler actually uses this word.

An abstract object is an object that cannot be used directly. Instead, a descendent object must always be used. However, for Delphi compatibility, the compiler ignores this directive.

5.3 Fields

Object Fields are like record fields. They are accessed in the same way as a record field would be accessed: by using a qualified identifier. Given the following declaration:

```
Type TAnObject = Object
       AField : Longint;
       Procedure AMethod;
       end;
Var AnObject : TAnObject;
```

then the following would be a valid assignment:

```
  AnObject.AField := 0;
```

Inside methods, fields can be accessed using the short identifier:

```
Procedure TAnObject.AMethod;
begin
  ...
  AField := 0;
  ...
end;
```

Or, one can use the `self` identifier. The `self` identifier refers to the current instance of the object:

```
Procedure TAnObject.AMethod;
begin
  ...
  Self.AField := 0;
  ...
end;
```

One cannot access fields that are in a private or protected sections of an object from outside the objects' methods. If this is attempted anyway, the compiler will complain about an unknown identifier.

It is also possible to use the `with` statement with an object instance, just as with a record:

```
With AnObject do
  begin
  Afield := 12;
  AMethod;
  end;
```

In this example, between the `begin` and `end`, it is as if `AnObject` was prepended to the `Afield` and `Amethod` identifiers. More about this in section 13.2.8, page 215.

5.4 Class or Static fields

An object can contain class or static fields: these fields are global to the object type, and act like global variables, but are known only in the scope of the object. The difference between static and class variables is purely the mode in which they work: The `static` keyword will always work, the `class` keyword will need `{$MODE DELPHI}` or `{$MODE OBJFPC}`.

They can be referenced from within the objects methods, but can also be referenced from outside the object by providing the fully qualified name.

For instance, the output of the following program:

```
{$mode objfpc}
type
  cl=object
    l : longint; static;
    class var v : integer;
  end;

var
  cl1,cl2 : cl;

begin
  Writeln('Static');
  cl1.l:=2;
  writeln(cl2.l);
  cl2.l:=3;
  writeln(cl1.l);
  Writeln(cl.l);
  Writeln('Class');
  cl1.v:=4;
  writeln(cl2.v);
  cl2.v:=5;
  writeln(cl1.v);
  Writeln(cl.v);
end.
```

will be the following

```
Static
2
3
3
Class
4
```

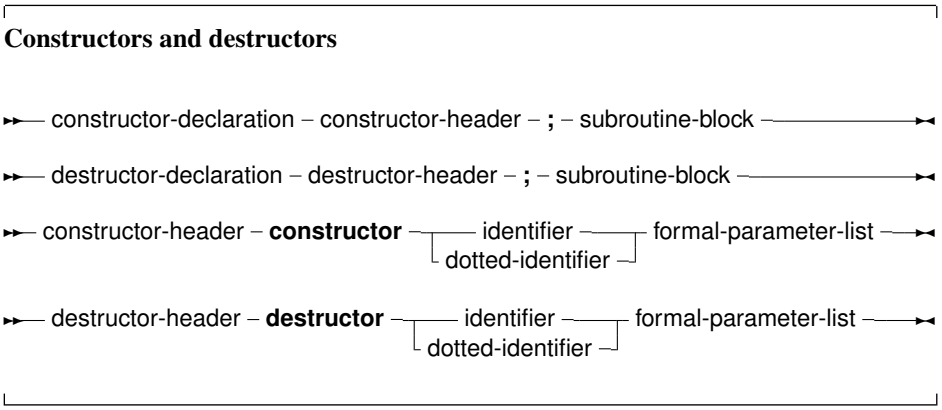
```
5
5
```

Note that the last line of code references the object type itself (`cl`), and not an instance of the object (`cl1` or `cl2`).

5.5 Constructors and destructors

As can be seen in the syntax diagram for an object declaration, Free Pascal supports constructors and destructors. The programmer is responsible for calling the constructor and the destructor explicitly when using objects.

The declaration of a constructor or destructor is as follows:



A constructor/destructor pair is *required* if the object uses virtual methods. The reason is that for an object with virtual methods, some internal housekeeping must be done: this housekeeping is done by the constructor[1].

In the declaration of the object type, a simple identifier should be used for the name of the constructor or destructor. When the constructor or destructor is implemented, the fully qualified identifier should be used for the method, i. e. an identifier of the form `objectidentifier.methodidentifier`.

Free Pascal supports also the extended syntax of the `New` and `Dispose` procedures. In case a dynamic variable of an object type must be allocated the constructor's name can be specified in the call to `New`. The `New` is implemented as a function which returns a pointer to the instantiated object. Consider the following declarations:

```
Type
  TObj = object;
   Constructor init;
   ...
   end;
  Pobj = ^TObj;
Var PP : Pobj;
```

Then the following three calls are equivalent:

```
 pp := new (Pobj,Init);
```

[1] A pointer to the VMT must be set up.

and

```
new(pp,init);
```

and also

```
new (pp);
pp^.init;
```

In the last case, the compiler will issue a warning that the extended syntax of `new` and `dispose` must be used to generate instances of an object. It is possible to ignore this warning, but it's better programming practice to use the extended syntax to create instances of an object. Similarly, the `Dispose` procedure accepts the name of a destructor. The destructor will then be called, before removing the object from the heap.

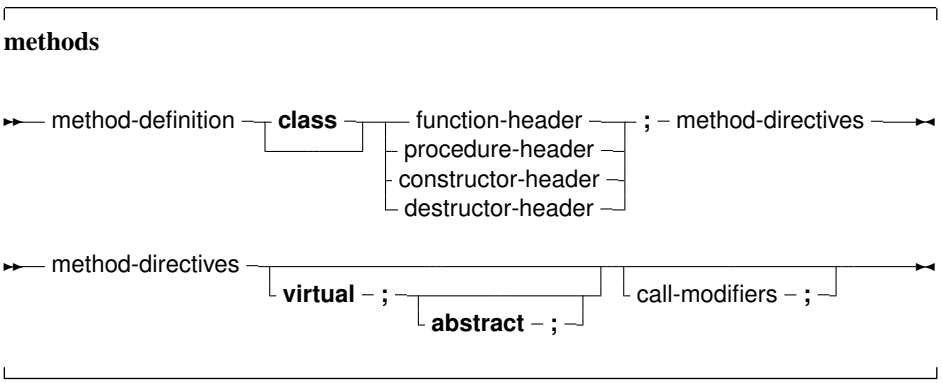
In view of the compiler warning remark, the following chapter presents the Delphi approach to object-oriented programming, and may be considered a more natural way of object-oriented programming.

5.6 Methods

Object methods are just like ordinary procedures or functions, only they have an implicit extra parameter: `self`. Self points to the object with which the method was invoked. When implementing methods, the fully qualified identifier must be given in the function header. When declaring methods, a normal identifier must be given.

5.6.1 Declaration

The declaration of a method is much like a normal function or procedure declaration, with some additional specifiers, as can be seen from the following diagram, which is part of the object declaration:



from the point of view of declarations, `Method definitions` are normal function or procedure declarations. Contrary to TP and Delphi, fields can be declared after methods in the same block, i. e. the following will generate an error when compiling with Delphi or Turbo Pascal, but not with FPC:

```
Type
  MyObj = Object
    Field : Longint;
    Procedure Doit;
  end;
```

5.6.2 Method invocation

Methods are called just as normal procedures are called, only they have an object instance identifier prepended to them (see also chapter 13, page 196). To determine which method is called, it is necessary to know the type of the method. We treat the different types in what follows.

Normal static methods

Normal (static) methods are methods that have been declared without a `abstract` or `virtual` keyword. When calling a static method, the declared (i. e. compile time) method of the object is used. For example, consider the following declarations:

```
Type
  TParent = Object
    ...
    procedure Doit;
    ...
    end;
  PParent = ^TParent;
  TChild = Object(TParent)
    ...
    procedure Doit;
    ...
    end;
  PChild = ^TChild;
```

As it is visible, both the parent and child objects have a method called `Doit`. Consider now the following declarations and calls:

```
Var
  ParentA,ParentB : PParent;
  Child           : PChild;

begin
   ParentA := New(PParent,Init);
   ParentB := New(PChild,Init);
   Child := New(PChild,Init);
   ParentA^.Doit;
   ParentB^.Doit;
   Child^.Doit;
```

Of the three invocations of `Doit`, only the last one will call `TChild.Doit`, the other two calls will call `TParent.Doit`. This is because for static methods, the compiler determines at compile time which method should be called. Since `ParentB` is of type `TParent`, the compiler decides that it must be called with `TParent.Doit`, even though it will be created as a `TChild`. There may be times when the method that is actually called should depend on the actual type of the object at run-time. If so, the method cannot be a static method, but must be a virtual method.

Virtual methods

To remedy the situation in the previous section, `virtual` methods are created. This is simply done by appending the method declaration with the `virtual` modifier. The descendent object can then override the method with a new implementation by re-declaring the method (with the same parameter list) using the `virtual` keyword.

Going back to the previous example, consider the following alternative declaration:

```
Type
  TParent = Object
    ...
    procedure Doit;virtual;
    ...
    end;
  PParent = ^TParent;
  TChild = Object(TParent)
    ...
    procedure Doit;virtual;
    ...
    end;
  PChild = ^TChild;
```

As it is visible, both the parent and child objects have a method called `Doit`. Consider now the following declarations and calls:

```
Var
  ParentA,ParentB : PParent;
  Child           : PChild;

begin
   ParentA := New(PParent,Init);
   ParentB := New(PChild,Init);
   Child := New(PChild,Init);
   ParentA^.Doit;
   ParentB^.Doit;
   Child^.Doit;
```

Now, different methods will be called, depending on the actual run-time type of the object. For `ParentA`, nothing changes, since it is created as a `TParent` instance. For `Child`, the situation also doesn't change: it is again created as an instance of `TChild`.

For `ParentB` however, the situation does change: Even though it was declared as a `TParent`, it is created as an instance of `TChild`. Now, when the program runs, before calling `Doit`, the program checks what the actual type of `ParentB` is, and only then decides which method must be called. Seeing that `ParentB` is of type `TChild`, `TChild.Doit` will be called. The code for this run-time checking of the actual type of an object is inserted by the compiler at compile time.

The `TChild.Doit` is said to *override* the `TParent.Doit`. It is possible to access the `TParent.Doit` from within the varTChild.Doit, with the `inherited` keyword:

```
Procedure TChild.Doit;
begin
  inherited Doit;
  ...
end;
```

In the above example, when `TChild.Doit` is called, the first thing it does is call `TParent.Doit`. The inherited keyword cannot be used in static methods, only on virtual methods.

To be able to do this, the compiler keeps – per object type – a table with virtual methods: the VMT (Virtual Method Table). This is simply a table with pointers to each of the virtual methods: each virtual method has its fixed location in this table (an index). The compiler uses this table to look

up the actual method that must be used. When a descendent object overrides a method, the entry of the parent method is overwritten in the VMT. More information about the VMT can be found in Programmer's Guide.

As remarked earlier, objects that have a VMT must be initialized with a constructor: the object variable must be initialized with a pointer to the VMT of the actual type that it was created with.

Abstract methods

An abstract method is a special kind of virtual method. A method that is declared `abstract` does not have an implementation for this method. It is up to inherited objects to override and implement this method.

From this it follows that a method can not be abstract if it is not virtual (this can be seen from the syntax diagram). A second consequence is that an instance of an object that has an abstract method cannot be created directly.

The reason is obvious: there is no method where the compiler could jump to! A method that is declared `abstract` does not have an implementation for this method. It is up to inherited objects to override and implement this method. Continuing our example, take a look at this:

```
Type
  TParent = Object
    ...
    procedure Doit;virtual;abstract;
    ...
    end;
  PParent=^TParent;
  TChild = Object(TParent)
    ...
    procedure Doit;virtual;
    ...
    end;
  PChild = ^TChild;
```

As it is visible, both the parent and child objects have a method called `Doit`. Consider now the following declarations and calls:

```
Var
  ParentA,ParentB : PParent;
  Child           : PChild;

begin
   ParentA := New(PParent,Init);
   ParentB := New(PChild,Init);
   Child := New(PChild,Init);
   ParentA^.Doit;
   ParentB^.Doit;
   Child^.Doit;
```

First of all, Line 3 will generate a compiler error, stating that one cannot generate instances of objects with abstract methods: The compiler has detected that `PParent` points to an object which has an abstract method. Commenting line 3 would allow compilation of the program.

Remark If an abstract method is overridden, the parent method cannot be called with `inherited`, since there is no parent method; The compiler will detect this, and complain about it, like this:

```
testo.pp(32,3) Error: Abstract methods can't be called directly
```

If, through some mechanism, an abstract method is called at run-time, then a run-time error will occur. (run-time error 211, to be precise)

Class or static methods

Class methods or methods declared with the `static` directive are methods that are global to the object type. When called, the implicit "self" pointer is not available. This means that normal methods cannot be called, and none of the fields of an object can be accessed. Class variables can be used, however.

Class or static methods are regular methods, they can be assigned to a procedural variable.

The following program demonstrates all this. The commented-out statements will not compile.

```
{$APPTYPE CONSOLE}
{$IFDEF FPC}{$MODE DELPHI}{$H+}{$ENDIF}
type
  TTest = object
    const Epsylon = 100;
    var f : integer;
    class var cv1,cv2:integer;
    procedure myproc;
    class procedure testproc;
    class procedure testproc2;static;
    procedure testproc3; static;
  end;

  procedure TTest.myproc;
  begin
    cv1:=0;
    f:=1;
  end;

  class procedure TTest.Testproc;
  begin
    cv1:=1;
    // f:=1;
  end;

  class procedure TTest.Testproc2;
  begin
    cv1:=2;
    // f:=1;
  end;

  procedure TTest.Testproc3;
  begin
    cv1:=3;
    // f:=1;
  end;


Var
```

```
  P : Procedure;

begin
  P:=@TTest.Myproc;
  P:=@TTest.Testproc;
  P:=@TTest.Testproc2;
  P:=@TTest.Testproc3;
end.
```

Uncommenting one of the commented statements and trying to compile the resulting code will result in a compiler error:

```
ocv.pp(32,6) Error: Only class methods, class properties and
  class variables can be accessed in class methods
```

5.7 Visibility

For objects, three visibility specifiers exist: `private`, `protected` and `public`. If a visibility specifier is not specified, `public` is assumed. Both methods and fields can be hidden from a programmer by putting them in a `private` section. The exact visibility rule is as follows:

Private All fields and methods that are in a `private` block, can only be accessed in the module (i. e. unit or program) that contains the object definition. They can be accessed from inside the object's methods or from outside them e. g. from other objects' methods, or global functions.

Protected Is the same as `Private`, except that the members of a `Protected` section are also accessible to descendent types, even if they are implemented in other modules.

Public fields and methods are always accessible, from everywhere. Fields and methods in a `public` section behave as though they were part of an ordinary `record` type.

Chapter 6

Classes

In the Delphi approach to Object Oriented Programming, everything revolves around the concept of "Classes". A class can be seen as a pointer to an object, or a pointer to a record, with methods associated with it.

The difference between objects and classes is mainly that an object is allocated on the stack, as an ordinary record would be, and that classes are always allocated on the heap. In the following example:

```
Var
  A : TSomeObject; // an Object
  B : TSomeClass;  // a Class
```

The main difference is that the variable `A` will take up as much space on the stack as the size of the object (`TSomeObject`). The variable `B`, on the other hand, will always take just the size of a pointer on the stack. The actual class data is on the heap.

From this, a second difference follows: a class must *always* be initialized through its constructor, whereas for an object, this is not necessary. Calling the constructor allocates the necessary memory on the heap for the class instance data.

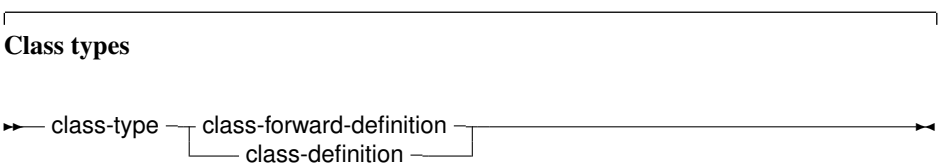
Remark In earlier versions of Free Pascal it was necessary, in order to use classes, to put the objpas unit in the uses clause of a unit or program. *This is no longer needed* as of version 0.99.12. As of this version, the unit will be loaded automatically when the `-MObjfpc` or `-MDelphi` options are specified, or their corresponding directives are used:

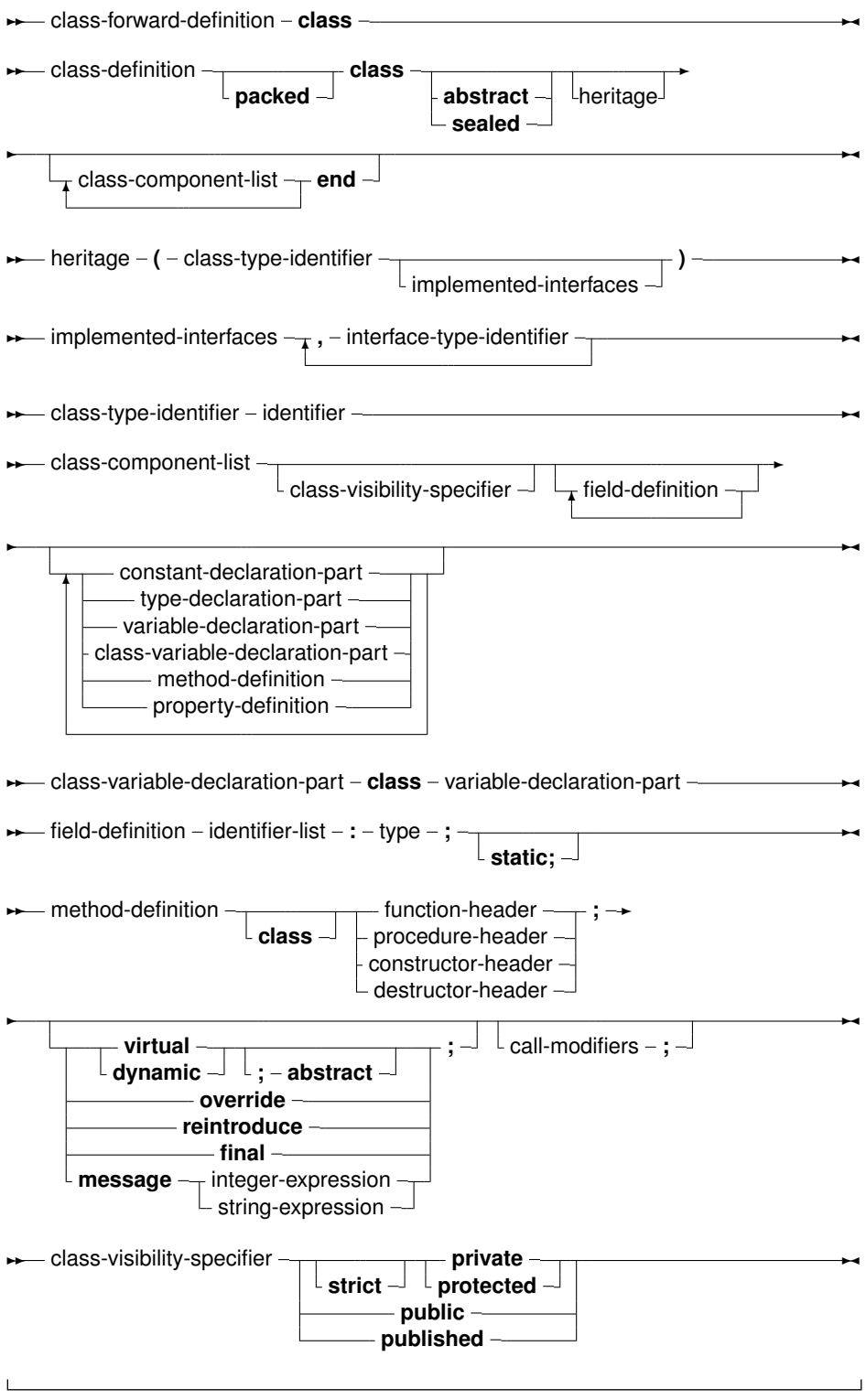
```
{$mode objfpc}
{$mode delphi}
```

In fact, the compiler will give a warning if it encounters the objpas unit in a uses clause.

6.1 Class definitions

The prototype declaration of a class is as follows:





Remark In MacPas mode, the `Object` keyword is replaced by the `class` keyword for compatibility with other pascal compilers available on the Mac. That means that in MacPas mode, the reserved word "class" in the above diagram may be replaced by the reserved word "object".

In a class declaration, as many visibility blocks as needed can be used: the various blocks can be repeated, and there is no special order in which they must appear.

Methods are normal function or procedure declarations. As can be seen, the declaration of a class is almost identical to the declaration of an object. The real difference between objects and classes is in the way they are created (see further in this chapter).

The visibility of the different sections is as follows:

Private All fields and methods that are in a `private` block, can only be accessed in the module (i. e. unit) that contains the class definition. They can be accessed from inside the classes' methods or from outside them (e. g. from other classes' methods)

Strict Private All fields and methods that are in a `strict private` block, can only be accessed from methods of the class itself. Other classes or descendent classes (even in the same unit) cannot access strict private members.

Protected Is the same as `Private`, except that the members of a `Protected` section are also accessible to descendent types, even if they are implemented in other modules. It follows that code outside a descendant in the module that defines a descendant also will have access to these protected methods (since the descendant inherits and exposes the protected methods with the same visibility).

Strict Protected Is the same as `Protected`, except that the members of a `Protected` section are also accessible to other classes implemented in the same unit. `Strict protected` members are only visible to descendent classes, not to other classes in the same unit.

Public sections are always accessible.

Published From a language perspective, this is the same as a `Public` section, but the compiler generates also type information that is needed for automatic streaming of these classes if the compiler is in the `{$M+}` state. Fields defined in a `published` section must be of class type. Array properties cannot be in a `published` section.

In the syntax diagram, it can be seen that a class can list implemented interfaces. This feature will be discussed in the next chapter.

Classes can contain `Class` methods: these are functions that do not require an instance. The `Self` identifier is valid in such methods, but refers to the class pointer (the VMT).

Remark Like with functions and pointer types, sometimes a forward definition of a class is needed. A class forward definition is simply the name of the class, with the keyword `Class`, as in the following example:

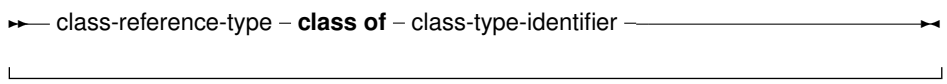
```
Type
  TClassB = Class;
  TClassA = Class
    B : TClassB;
  end;

  TClassB = Class
   A : TClassA;
  end;
```

When using a class forward definition, the class must be defined in the same unit, in the same section (interface/implementation). It must not necessarily be defined in the same type section.

It is also possible to define class reference types:

Class reference type



Class reference types are used to create instances of a certain class, which is not yet known at compile time, but which is specified at run time. Essentially, a variable of a class reference type contains a pointer to the definition of the specified class. This can be used to construct an instance of the class corresponding to the definition, or to check inheritance. The following example shows how it works:

```
Type
  TComponentClass = Class of TComponent;

Function CreateComponent(AClass: TComponentClass;
                         AOwner: TComponent): TComponent;

begin
  // ...
  Result:=AClass.Create(AOwner);
  // ...
end;
```

This function can be passed a class reference of any class that descends from `TComponent`. The following is a valid call:

```
Var
  C : TComponent;

begin
  C:=CreateComponent(TEdit,Form1);
end;
```

On return of the `CreateComponent` function, `C` will contain an instance of the class `TEdit`. Note that the following call will fail to compile:

```
Var
  C : TComponent;

begin
  C:=CreateComponent(TStream,Form1);
end;
```

because `TStream` does not descend from `TComponent`, and `AClass` refers to a `TComponent` class. The compiler can (and will) check this at compile time, and will produce an error.

References to classes can also be used to check inheritance:

```
  TMinClass = Class of TMyClass;
  TMaxClass = Class of TMyClassChild;

Function CheckObjectBetween(Instance : TObject) : boolean;

begin
  If not (Instance is TMinClass)
     or ((Instance is TMaxClass)
          and (Instance.ClassType<>TMaxClass)) then
    Raise Exception.Create(SomeError)
end;
```

The above example will raise an exception if the passed instance is not a descendent of `TMinClass` or a descendent of `TMaxClass`.

More about instantiating a class can be found in section 6.4, page 99.

6.2 Abstract and sealed classes

A class can be declared as `sealed`. In that case, it is not possible to declare a descendent class. The compiler will return an error if it encounters a declaration of a descendent:

```
{$mode objfpc}
{$h+}

Type
  TMyClass = Class Sealed
    x : integer;
  end;

  TMyClass2 = Class(TMyClass)
    Y : Integer;
  end;

begin
end.
```

This will result in the following error:

```
Error: Cannot create a descendant of the sealed class "TMyClass"
```

An abstract class is a class that cannot be instantiated directly. Instead, a descendent class must always be instantiated. However, for Delphi compatibility, the compiler ignores this directive.

6.3 Normal and static fields

Classes can have fields. Depending on how they are defined, fields hold data specific to an instance of a class or to the class as a whole. Whatever the way they were defined, fields observe the rules of visibility just like any other member of the class.

6.3.1 Normal fields/variables

There are two ways to declare a normal field. The first one is the classical way, similar to a definition in an object:

```
{$mode objfpc}
type
  cl=class
    l : longint;
  end;
var
  cl1,cl2 : cl;
begin
```

```
  cl1:=cl.create;
  cl2:=cl.create;
  cl1.l:=2;
  writeln(cl1.l);
  writeln(cl2.l);
end.
```

will be the following

```
2
0
```

The example demonstrates that values of fields are initialized with zero (or the equivalent of zero for non ordinal types: empty string, empty array and so on).

The second way to declare a field (only available in more recent versions of Free Pascal) is using a `var` block:

```
{$mode objfpc}
type
  cl=class
  var
    l : longint;
  end;
```

This definition is completely equivalent to the previous definition.

Remark As of version 3.0 of the compiler, the compiler can re-order the fields in memory if this leads to better alignment and smaller instances. That means that in an instance, the fields do not necessarily appear in the same order as in the declaration. RTTI generated for a class will reflect this change.

6.3.2 Class fields/variables

Similar to objects, a class can contain static fields or class variables: these fields or variables are global to the class, and act like global variables, but are known only as part of the class. They can be referenced from within the classes' methods, but can also be referenced from outside the class by providing the fully qualified name.

Again, there are two ways to define class variables. The first one is equivalent to the way it is done in objects, using a static modifier:

For instance, the output of the following program is the same as the output for the version using an object:

```
{$mode objfpc}
type
  cl=class
    l : longint;static;
  end;
var
  cl1,cl2 : cl;
begin
  cl1:=cl.create;
  cl2:=cl.create;
  cl1.l:=2;
  writeln(cl2.l);
```

```
  cl2.l:=3;
  writeln(cl1.l);
  Writeln(cl.l);
end.
```

The output of this will be the following:

```
2
3
3
```

Note that the last line of code references the class type itself (`cl`), and not an instance of the class (`cl1` or `cl2`).

In addition to the static field approach, in classes, a `Class Var` can be used. Similar to the way a field can be defined in a variable block, a class variable can be declared in a class var block:

```
{$mode objfpc}
type
  cl=class
  class var
    l : longint;
  end;
```

This definition is equivalent to the previous one.

Note that a class variable is tied to a specific class. Descendent classes will refer to the same instance, unless the variable is redeclared. The following program demonstrates this:

```
{$mode objfpc}
type
  TA = class // base type
    class var CODE: integer;
  end;
  TB = class(TA);
  TC = class(TA);

begin
  TA.Code:=0;
  TB.Code:=1;
  TC.Code:=2;
  Writeln(Ta.Code:2,Tb.Code:2,Tc.code:2);
end.
```

The output of this program is:

```
 2 2 2
```

Because it is tied to a class, it can be overridden in Delphi mode:

```
{$mode delphi}
type
  TA = class // base type
    class var CODE: integer;
  end;
```

```
  TB = class(TA)
    Class var code : integer;
  end;
  TC = class(TA)
    Class var code : integer;
  end;

begin
  TA.Code:=0;
  TB.Code:=1;
  TC.Code:=2;
  Writeln(Ta.Code:2,Tb.Code:2,Tc.code:2);
end.
```

It will print the following:

```
 0 1 2
```

However, in OBJFPC mode it will not compile, and will give a duplicate identifier error.

6.4 Class instantiation: Creating a class instance

Class instances must be created using one of their constructors (there can be multiple constructors). Remember that a class is a pointer to an object on the heap. When a variable of some class is declared, the compiler just allocates room for this pointer, not the entire object. The constructor of a class returns a pointer to an initialized instance of the class on the heap. So, to initialize an instance of some class, one would do the following:

```
  ClassVar := ClassType.ConstructorName;
```

The extended syntax of `new` and `dispose` can *not* be used to instantiate and destroy class instances: That construct is reserved for use with objects only. Calling the constructor will provoke a call to the virtual class method `NewInstance`, which, in its default implementation, calls `GetMem`, to allocate enough space to hold the class instance data, and then zeroes out the memory.

After that, the constructor's code is executed. The constructor has a pointer to its data, in `Self`.

Remark

- The `{$PackRecords }` directive also affects classes, i. e. the alignment in memory of the different fields depends on the value of the `{$PackRecords }` directive.
- Just as for objects and records, a packed class can be declared. This has the same effect as on an object, or record, namely that the elements are aligned on 1-byte boundaries, i. e. as close as possible.
- `SizeOf(class)` will return the same as `SizeOf(Pointer)`, since a class is a pointer to an object. To get the size of the class instance data, use the `TObject.InstanceSize` method.
- If an unhandled exception happens during an the execution of a constructor, the destructor will be called automatically. In this case, the variable to which the class instance is to be assigned does not change.

6.5 Class destruction : Disposing of a class instance

Class instances must be destroyed using the destructor. In difference with the constructor, there is no choice in destructors: the destructor *must* have the name `Destroy`, it *must* override the `Destroy` destructor declared in `TObject`, cannot have arguments, and the inherited destructor must always be called.

`Destroy` will call `FreeInstance`, which, in its default implementation, calls `FreeMem` to release the memory occupied by the instance.

To avoid calling a destructor on a `Nil` instance, it is best to call the `Free` method of `TObject`. This method will check if `Self` is not `Nil`, and if so, then it calls `Destroy`. If `Self` equals `Nil`, it will just exit.

Destroying an instance does not remove or Nil a reference to an instance:

```
uses
  Classes;
var
  A: TComponent;
begin
  A:=TComponent.Create(Nil);
  A.Name:='MyComponent';
  A.Free;
  Writeln('A is still assigned: ',Assigned(A));
end.
```

After the call to `Free`, the variable A will not be `Nil`, the output of this program will be:

```
A is still assigned: TRUE
```

To make sure that the variable `A` is cleared after the destructor was called, the function `FreeAndNil` from the SysUtils unit can be used. It will call `Free` and will then write `Nil` in the object pointer (`A` in the above example):

```
uses
  Classes, SysUtils;
var
  A: TComponent;
begin
  A:=TComponent.Create(Nil);
  A.Name:='MyComponent';
  FreeAndNil(A);
  Writeln('A is still assigned: ',Assigned(A));
end.
```

After the call to `FreeAndNil`, the variable `A` will contain `Nil`, the output of this program will be:

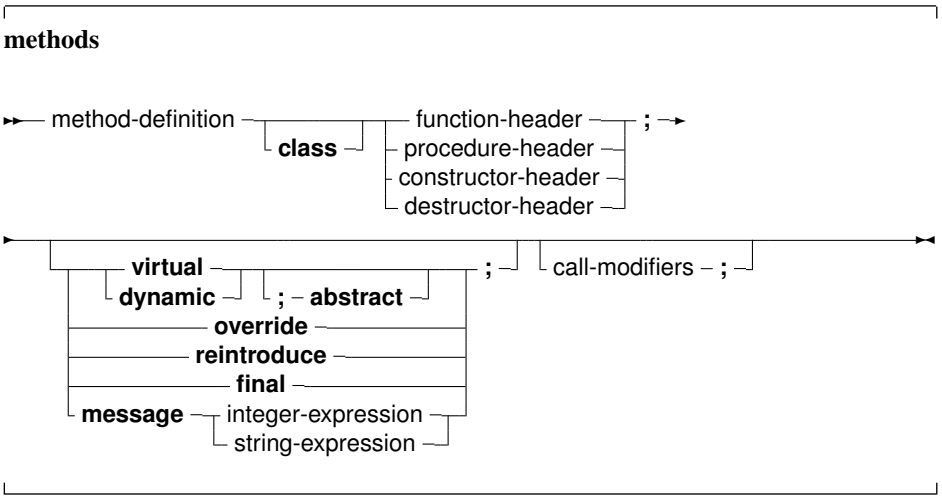
```
A is still assigned: FALSE
```

Remark if an unhandled exception happens during an the execution of a constructor, the destructor will be called automatically. It must be prepared for the fact that not all fields might have been initialized.

6.6 Methods

6.6.1 Declaration

Declaration of methods in classes follows the same rules as method declarations in objects:



The only differences are the `override`, `reintroduce`, `final` and `message` directives.

6.6.2 Implementation

Method implementations for classes ar no different than for objects: the implementation is specified using the `ClassName.MethodName` identifier:

```
Type
   TMyClass  = class
     Procedure DoIt;
   end;

procedure TMyClass.DoIt;

begin
  // Do something nice
end;
```

Note: Because the compiler considers a type alias to be the same type as the aliased type, type aliases can currently be used when writing the implementation of a method:

```
Interface

Type
   TSomeReallyAnnoyingIncrediblyLongClassName = Class
     Procedure DoSomething;
   end;

implementation

Type
```

```
   T1 = TSomeReallyAnnoyingUltraLongClassName;

procedure T1.DoSomething;

begin
end;
```

6.6.3 Invocation

Method invocation for classes is no different than for objects. The following is a valid method invocation:

```
Var A : TMyClass;
begin
  A := TMyClass.Create;
  A.MyProc;
```

6.6.4 Virtual methods

Classes have virtual methods, just as objects do. There is however a difference between the two. For objects, it is sufficient to redeclare the same method in a descendent object with the keyword `virtual` to override it. For classes, the situation is different: virtual methods *must* be overridden with the `override` keyword. Failing to do so, will start a *new* batch of virtual methods, hiding the previous one. The `Inherited` keyword will not jump to the inherited method, if `Virtual` was used.

The following code is *wrong*:

```
Type
  TParent = Class
    Procedure MyProc; virtual;
  end;
  TChild  = Class(TParent)
    Procedure MyProc; virtual;
  end;
```

The compiler will produce a warning:

```
Warning: An inherited method is hidden by "MyProc;"
```

The compiler will compile it, but using `Inherited` can produce strange effects.

The correct declaration is as follows:

```
Type
  TParent = Class
    Procedure MyProc; virtual;
  end;
  TChild  = Class(TParent)
    Procedure MyProc; override;
  end;
```

This will compile and run without warnings or errors.

If the virtual method should really be replaced with a method with the same name, then the `reintroduce` keyword can be used:

```
Type
  TParent = Class
    Procedure MyProc; virtual;
  end;
  TChild  = Class(TParent)
    Procedure MyProc; reintroduce;
  end;
```

This new method is no longer virtual.

To be able to do this, the compiler keeps – per class type – a table with virtual methods: the VMT (Virtual Method Table). This is simply a table with pointers to each of the virtual methods: each virtual method has its fixed location in this table (an index). The compiler uses this table to look up the actual method that must be used at runtime. When a descendent object overrides a method, the entry of the parent method is overwritten in the VMT. More information about the VMT can be found in Programmer's Guide.

Remark The keyword "virtual" can be replaced with the "dynamic" keyword: dynamic methods behave the same as virtual methods. Unlike in Delphi, in FPC the implementation of dynamic methods is equal to the implementation of virtual methods.

6.6.5 Final methods

The `final` method can be used on a virtual or dynamic (or overridden) method to indicate it can no longer be overridden.

Given the following declarations:

```
type
  TParent = class
    procedure Test; virtual;
    procedure TestFinal; virtual; final;
  end;

  TChild1 = class(TParent)
    procedure test; override; final;
  end;
```

The following declarations will result in compiler errors:

```
  TChild2 = class(TChild1)
    // Error: There is no method in an ancestor class to be overridden: "Test;"
    procedure test; override;
  end;

  TChild3 = class(TParent)
    procedure TestProcedure; override;
    // Error: There is no method in an ancestor class to be overridden: "TestFinal;"
    procedure TestFinal; override;
  end;
```

6.6.6 Class methods

Class methods are identified by the keyword `Class` in front of the procedure or function declaration, as in the following example:

```
  Class Function ClassName : String;
```

Class methods are methods that do not have an instance (i. e. Self does not point to a class instance) but which follow the scoping and inheritance rules of a class. They can be used to return information about the current class, for instance for registration or use in a class factory. Since no instance is available, no information available in instances can be used.

Class methods can be called from inside a regular method, but can also be called using a class identifier:

```
Var
  AClass : TClass; // AClass is of type "type of class"

begin
  ..
  if CompareText(AClass.ClassName,'TCOMPONENT')=0 then
  ...
```

But calling them from an instance is also possible:

```
Var
  MyClass : TObject;

begin
  ..
  if MyClass.ClassNameis('TCOMPONENT') then
  ...
```

The reverse is not possible: Inside a class method, the `Self` identifier points to the VMT table of the class. No fields, properties or regular methods are available inside a class method. Accessing a regular property or method will result in a compiler error.

Note that class methods can be virtual, and can be overridden.

Class methods can be used as read or write specifiers for a regular property, but they still won't be able to read or write instance fields.

6.6.7 Class type initialization

A class constructor or destructor can also be created. They serve to instantiate some class variables or class properties which must be initialized before a class can be used. These constructors are called automatically at program startup: The constructor is called before the initialization section of the unit it is declared in, the destructor is called after the finalisation section of the unit it is declared in.

There are some caveats when using class destructors/constructors:

- There may be only one constructor per class. The name is arbitrary, but it can not have parameters.
- There may be only one destructor per class. The name is arbitrary, but it can not have parameters.
- Neither constructor nor destructor can be virtual.
- The class constructor/destructor is called irrespective of the use of the class: even if a class is never used, the constructor and destructor are called anyway.

- There is no guaranteed order in which the class constructors or destructors are called. For nested classes, the only guaranteed order is that the constructors of nested classes are called after the constructor of the encompassing class is called, and for the destructors the opposite order is used.

The following program:

```
{$mode objfpc}
{$h+}

Type
  TA = Class(TObject)
  Private
    Function GetA : Integer;
    Procedure SetA(AValue : integer);

  public
    Class Constructor create;
    Class Destructor destroy;
    Property A : Integer Read GetA Write SetA;
  end;

{Class} Function TA.GetA : Integer;

begin
  Result:=-1;
end;

{Class} Procedure TA.SetA(AValue : integer);

begin
  //
end;

Class Constructor TA.Create;

begin
  Writeln('Class constructor TA');
end;

Class Destructor TA.Destroy;

begin
  Writeln('Class destructor TA');

end;

Var
  A : TA;

begin
end.
```

Will, when run, output the following:

```
Class constructor TA
Class destructor TA
```

6.6.8 Static class methods

FPC knows static class methods in classes: these are class methods that have the `Static` keyword at the end. These methods behave completely like regular procedures or functions. This means that:

- They do not have a `Self` parameter. As a result, they cannot access properties or fields or regular methods.
- They cannot be virtual.
- They can be assigned to regular procedural variables.

Their use is mainly to include the method in the namespace of the class as opposed to having the procedure in the namespace of the unit. Note that they do have access to all class variables, types etc, meaning something like this is possible:

```
 {$mode objfpc}
{$h+}

Type
  TA = Class(TObject)
  Private
    class var myprivatea : integer;
  public
    class Function GetA : Integer;  static;
    class Procedure SetA(AValue : Integer); static;
  end;

Class Function TA.GetA : Integer;

begin
  Result:=myprivateA;
end;

Class Procedure TA.SetA(AValue : integer);

begin
  myprivateA:=AValue;
end;


begin
  TA.SetA(123);
  Writeln(TA.MyPrivateA);
end.
```

Which will output `123`, when run.

In the implementation of a static class method, the `Self` identifier is not available. The method behaves as if `Self` is hardcoded to the declared class, not the actual class with which it was called. In regular class methods, `Self` contains the Actual class for which the method was called. The following example makes this clear:

```
Type
  TA = Class
    Class procedure DoIt; virtual;
    Class Procedure DoitStatic; static;
  end;

  TB = CLass(TA)
    Class procedure DoIt; override;
  end;


Class procedure TA.DOit;

begin
  Writeln('TA.Doit : ',Self.ClassName);
end;

Class procedure TA.DOitStatic;

begin
  Doit;
  Writeln('TA.DoitStatic : ',ClassName);
end;

Class procedure TB.DoIt;

begin
  Inherited;
  Writeln('TB.Doit : ',Self.ClassName);
end;

begin
  Writeln('Through static method:');
  TB.DoItStatic;
  Writeln('Through class method:');
  TB.Doit;
end.
```

When run, this example will print:

```
Through static method:
TA.Doit : TA
TA.DoitStatic : TA
Through class method:
TA.Doit : TB
TB.Doit : TB
```

For the static class method, even though it was called using `TB`, the class (`Self`, if it were available) is set to `TA`, the class in which the static method was defined. For the class method, the class is set to the actual class used to call the method (`TB`).

6.6.9 Covariant return type

When overriding a function that has a class typed result, it is possible to change the exact class to a descendant of the result type of the parent method.

This means that the following will compile:

```
Program Project1;
{$mode objfpc}
type
  TClass1 = class
  public
    function A: TClass1; virtual; abstract;
  end;

  TClass2 = class(TClass1)
  public
    function A: TClass2; override; // result differs: TClass2 as descendant of TClas
  end;

{ TClass2 }

function TClass2.A: TClass2;
begin
  Result := Self;
end;

begin
end.
```

The covariant return type can be used in ObjFPC and Delphi modes.

6.6.10 Message methods

New in classes are `message` methods. Pointers to message methods are stored in a special table, together with the integer or string constant that they were declared with. They are primarily intended to ease programming of callback functions in several `GUI` toolkits, such as `Win32` or `GTK`. In difference with Delphi, Free Pascal also accepts strings as message identifiers. Message methods are always virtual.

As can be seen in the class declaration diagram, message methods are declared with a `Message` keyword, followed by an integer constant expression.

Additionally, they can take only one var argument (typed or not):

```
 Procedure TMyObject.MyHandler(Var Msg); Message 1;
```

The method implementation of a message function is not different from an ordinary method. It is also possible to call a message method directly, but this should not be done. Instead, the `TObject.Dispatch` method should be used. Message methods are automatically virtual, i. e. they can be overridden in descendent classes.

The `TObject.Dispatch` method can be used to call a `message` handler. It is declared in the system unit and will accept a var parameter which must have at the first position a cardinal with the message ID that should be called. For example:

```
Type
```

```
  TMsg = Record
    MSGID : Cardinal;
    Data : Pointer;
Var
  Msg : TMSg;

MyObject.Dispatch (Msg);
```

In this example, the `Dispatch` method will look at the object and all its ancestors (starting at the object, and searching up the inheritance class tree), to see if a message method with message `MSGID` has been declared. If such a method is found, it is called, and passed the `Msg` parameter.

If no such method is found, `DefaultHandler` is called. `DefaultHandler` is a virtual method of `TObject` that doesn't do anything, but which can be overridden to provide any processing that might be needed. `DefaultHandler` is declared as follows:

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

In addition to the message method with a `Integer` identifier, Free Pascal also supports a message method with a string identifier:

```
 Procedure TMyObject.MyStrHandler(Var Msg); Message 'OnClick';
```

The working of the string message handler is the same as the ordinary integer message handler:

The `TObject.DispatchStr` method can be used to call a `message` handler. It is declared in the system unit and will accept one parameter which must have at the first position a short string with the message ID that should be called. For example:

```
Type
  TMsg = Record
    MsgStr : String[10]; // Arbitrary length up to 255 characters.
    Data : Pointer;
Var
  Msg : TMSg;

MyObject.DispatchStr (Msg);
```

In this example, the `DispatchStr` method will look at the object and all its ancestors (starting at the object, and searching up the inheritance class tree), to see if a message method with message `MsgStr` has been declared. If such a method is found, it is called, and passed the `Msg` parameter.

If no such method is found, `DefaultHandlerStr` is called. `DefaultHandlerStr` is a virtual method of `TObject` that doesn't do anything, but which can be overridden to provide any processing that might be needed. `DefaultHandlerStr` is declared as follows:

```
   procedure DefaultHandlerStr(var message);virtual;
```

6.6.11 Using inherited

In an overridden virtual method, it is often necessary to call the parent class' implementation of the virtual method. This can be done with the `inherited` keyword. Likewise, the `inherited` keyword can be used to call any method of the parent class.

The first case is the simplest:

```
Type
  TMyClass = Class(TComponent)
    Constructor Create(AOwner : TComponent); override;
  end;

Constructor TMyClass.Create(AOwner : TComponent);

begin
  Inherited;
  // Do more things
end;
```

In the above example, the `Inherited` statement will call `Create` of `TComponent`, passing it `AOwner` as a parameter: the same parameters that were passed to the current method will be passed to the parent's method. They must not be specified again: if none are specified, the compiler will pass the same arguments as the ones received.

If no inherited method with the same name exists, the `Inherited` will have no effect in this case. The presence of `Inherited` in this form can thus be interpreted as "call the overridden method if it exists".

The second case is slightly more complicated:

```
Type
  TMyClass = Class(TComponent)
    Constructor Create(AOwner : TComponent); override;
    Constructor CreateNew(AOwner : TComponent; DoExtra : Boolean);
  end;

Constructor TMyClass.Create(AOwner : TComponent);
begin
  Inherited;
end;

Constructor TMyClass.CreateNew(AOwner : TComponent; DoExtra : Boolean);
begin
  Inherited Create(AOwner);
  // Do stuff
end;
```

The `CreateNew` method will first call `TComponent.Create` and will pass it `AOwner` as a parameter. It will not call `TMyClass.Create`.

If no method with the given name exists in parent classes, the compiler will give an error.

Although the examples were given using constructors, the use of `inherited` is not restricted to constructors, it can be used for any procedure or function or destructor as well.

6.7 Method overloading

Similar to function overloading, method overloading simply means that the same method is defined more than once in the same class, but each time with a different formal parameter list.

The parameter lists must differ at least in one of its element types. When the compiler encounters a method call, it will look at the method's parameters to decide which one of the defined methods it should call. This can be useful when the same method must be defined for different types.

For example, in the `TStrings` class of the `Classes` unit, the `AddStrings` method has many overloads

```
procedure AddStrings(TheStrings: TStrings); overload; virtual;
procedure AddStrings(TheStrings: TStrings; ClearFirst : Boolean); overload;
procedure AddStrings(const TheStrings: array of string); overload; virtual;
procedure AddStrings(const TheStrings: array of string; ClearFirst : Boolean); overl
```

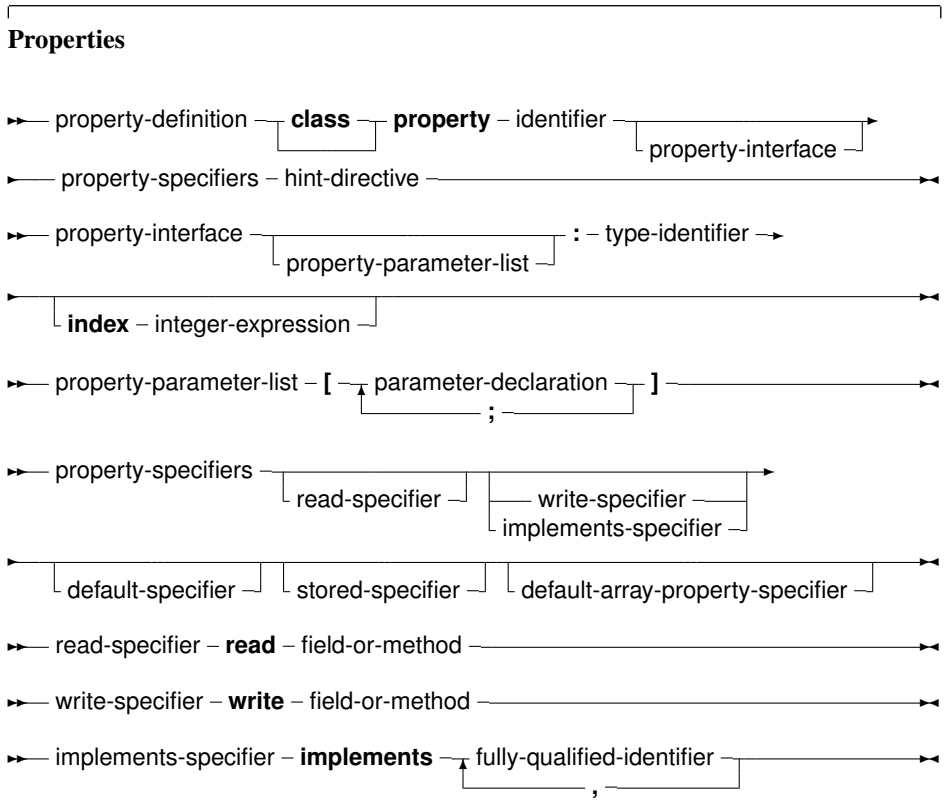
When the compiler encounters a call to the `AddStrings` method, it will first search which method it should use. It therefore checks the parameters in a method call, and looks if there is a method definition which matches the specified parameter list. If the compiler finds such a method, a call is inserted to that function. If no such method is found, a compiler error is generated.

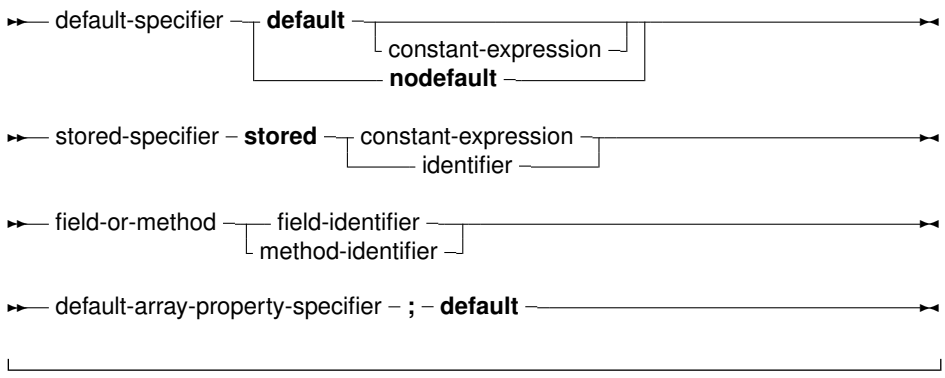
If the `overload` keyword is not present, then all overloaded versions of the method must reside in the same class, i. e. the compiler will not look for overloaded methods in parent classes if the `overload` keyword was not specified.

6.8 Properties

6.8.1 Definition

Classes can contain properties as part of their fields list. A property acts like a normal field, i. e. its value can be retrieved or set, but it allows to redirect the access of the field through functions and procedures. They provide a means to associate an action with an assignment of or a reading from a class "field". This allows e. g. checking that a value is valid when assigning, or, when reading, it allows to construct the value on the fly. Moreover, properties can be read-only or write only. The prototype declaration of a property is as follows:





A `read-specifier` is either the name of a field that contains the property, or the name of a method function that has the same return type as the property type. In the case of a simple type, this function must not accept an argument. In case of an array property, the function must accept a single argument of the same type as the index. In case of an indexed property, it must accept a integer as an argument.

A `read-specifier` is optional, making the property write-only. Note that class methods cannot be used as read specifiers for regular properties, they can only be used for class properties.

A `write-specifier` is optional: If there is no `write-specifier`, the property is read-only. A write specifier is either the name of a field, or the name of a method procedure that accepts as a sole argument a variable of the same type as the property. In case of an array property, the procedure must accept two arguments: the first argument must have the same type as the index, the second argument must be of the same type as the property. Similarly, in case of an indexed property, the first parameter must be an integer.

The section (`private`, `published`) in which the specified function or procedure resides is irrelevant. Usually, however, this will be a protected or private method.

For example, given the following declaration:

```
Type
  MyClass = Class
    Private
    Field1 : Longint;
    Field2 : Longint;
    Field3 : Longint;
    Procedure  Sety (value : Longint);
    Function Gety : Longint;
    Function Getz : Longint;
    Public
    Property X : Longint Read Field1 write Field2;
    Property Y : Longint Read GetY Write Sety;
    Property Z : Longint Read GetZ;
    end;

Var
  MyClass : TMyClass;
```

The following are valid statements:

```
WriteLn ('X : ',MyClass.X);
WriteLn ('Y : ',MyClass.Y);
WriteLn ('Z : ',MyClass.Z);
```

```
MyClass.X := 0;
MyClass.Y := 0;
```

But the following would generate an error:

```
MyClass.Z := 0;
```

because Z is a read-only property.

What happens in the above statements is that when a value needs to be read, the compiler inserts a call to the various `getNNN` methods of the object, and the result of this call is used. When an assignment is made, the compiler passes the value that must be assigned as a parameter to the various `setNNN` methods.

Because of this mechanism, properties cannot be passed as var arguments to a function or procedure, since there is no known address of the property (at least, not always).

As for many identifiers, properties can have hint directives (section 1.5, page 17):

```
Property MyProp : Integer read FMyProp; deprecated;
```

6.8.2 Indexed properties

If the property definition contains an index, then the read and write specifiers must be a function and a procedure. Moreover, these functions require an additional parameter : An integer parameter. This allows to read or write several properties with the same function. For this, the properties must have the same type. The following is an example of a property with an index:

```
{$mode objfpc}
Type
  TPoint = Class(TObject)
  Private
    FX,FY : Longint;
    Function GetCoord (Index : Integer): Longint;
    Procedure SetCoord (Index : Integer; Value : longint);
  Public
    Property X : Longint index 1 read GetCoord Write SetCoord;
    Property Y : Longint index 2 read GetCoord Write SetCoord;
    Property Coords[Index : Integer]:Longint Read GetCoord;
  end;

Procedure TPoint.SetCoord (Index : Integer; Value : Longint);
begin
  Case Index of
   1 : FX := Value;
   2 : FY := Value;
  end;
end;

Function TPoint.GetCoord (INdex : Integer) : Longint;
begin
  Case Index of
   1 : Result := FX;
   2 : Result := FY;
  end;
```

```
end;

Var
  P : TPoint;

begin
  P := TPoint.create;
  P.X := 2;
  P.Y := 3;
  With P do
    WriteLn ('X=',X,' Y=',Y);
end.
```

When the compiler encounters an assignment to `X`, then `SetCoord` is called with as first parameter the index (1 in the above case) and with as a second parameter the value to be set. Conversely, when reading the value of `X`, the compiler calls `GetCoord` and passes it index 1. Indexes can only be integer values.

6.8.3 Array properties

Array properties also exist. These are properties that accept an index, just as an array does. The index can be one-dimensional, or multi-dimensional. In difference with normal (static or dynamic) arrays, the index of an array property doesn't have to be an ordinal type, but can be any type.

A `read specifier` for an array property is the name method function that has the same return type as the property type. The function must accept as a sole argument a variable of the same type as the index type. For an array property, one cannot specify fields as `read specifiers`.

A `write specifier` for an array property is the name of a method procedure that accepts two arguments: the first argument has the same type as the index, and the second argument is a parameter of the same type as the property type. As an example, see the following declaration:

```
Type
  TIntList = Class
  Private
    Function GetInt (I : Longint) : longint;
    Function GetAsString (A : String) : String;
    Procedure SetInt (I : Longint; Value : Longint;);
    Procedure SetAsString (A : String; Value : String);
  Public
    Property Items [i : Longint] : Longint Read GetInt
                                           Write SetInt;
    Property StrItems [S : String] : String Read GetAsString
                                            Write SetAsstring;
  end;

Var
  AIntList : TIntList;
```

Then the following statements would be valid:

```
AIntList.Items[26] := 1;
AIntList.StrItems['twenty-five'] := 'zero';
WriteLn ('Item 26 : ',AIntList.Items[26]);
WriteLn ('Item 25 : ',AIntList.StrItems['twenty-five']);
```

While the following statements would generate errors:

```
AIntList.Items['twenty-five'] := 1;
AIntList.StrItems[26] := 'zero';
```

Because the index types are wrong.

Array properties can be multi-dimensional:

```
Type
  TGrid = Class
  Private
    Function GetCell (I,J : Longint) : String;
    Procedure SetCell (I,J : Longint; Value : String);
  Public
    Property Cellcs [Row,Col : Longint] : String Read GetCell
                                            Write SetCell;
  end;
```

If there are N dimensions, then the types of the first N arguments of the getter and setter must correspond to the types of the N index specifiers in the array property definition.

6.8.4 Default properties

Array properties can be declared as `default` properties. This means that it is not necessary to specify the property name when assigning or reading it. In the previous example, if the definition of the items property would have been

```
 Property Items[i : Longint]: Longint Read GetInt
                                      Write SetInt; Default;
```

Then the assignment

```
AIntList.Items[26] := 1;
```

Would be equivalent to the following abbreviation.

```
AIntList[26] := 1;
```

Only one default property per class is allowed, but descendent classes can redeclare the default property.

6.8.5 Published properties

Classes compiled in the `{$M+}` state (such as `TPersistent` from the classes unit) can have a published section. For methods, fields and properties in the `Published` section, the compiler generates RTTI information (Run Time Type Information), which can be used to query the defined methods, fields and properties in the published section(s). The `typinfo` unit contains the necessary routines to query this information, and this unit is used in the streaming system in FPC in the classes unit.

The RTTI is generated regardless of what the read and write specifiers are: fields, functions/procedures or indexed functions/procedures.

Only class-typed fields can be published. For properties, any simple property whose size is less than or equal to a pointer, can be declared published: floats, integers, sets (with less than 32 distinct elements), enumerated, classes or dynamic arrays (not array properties).

Although run-time type information is available for other types, these types cannot be used for a property or field definition in a published section. The information is present to describe for example arguments of procedures or functions.

6.8.6 Storage information

The compiler offers two specifiers to control whether a property is streamed using a streaming mechanism, such as the one implemented in the classes unit. These specifiers write extra information to the generated RTTI, which is later used in the streaming system:

1. The *stored* specifier.
2. The *default* specifier and its counterpart *nodefault*.

These two specifiers are both evaluated to decide whether a property must be streamed.

The argument to the `stored` specifier should be either a boolean constant, a boolean field of the class, or a parameterless function which returns a boolean result. If the argument of the stored specifier evaluates to `False`, the property will not be streamed.

Remark Absence of a `stored` specifier is equivalent to `stored True`.

If the evaluation of `Stored` resulted in `True`, the default for a property is considered:

The `default` specifier can be specified for ordinal types and sets. Properties that have as value this default value, will not be written to the stream by the streaming system, even if `Stored` is `True`. The default value is stored in the RTTI that is generated for the class.

String, floating-point and pointer properties have implicit `default` value of empty string, 0 or nil, respectively. Ordinal and set properties have no implicit `default` value.

The `nodefault` specifier must be used to indicate that a property has no default value. The effect is that the value of this property is always written to the stream when streaming the property, if stored is `True`.

Remark

1. When the class is instantiated, the default value is not automatically applied to the property, it is the responsibility of the programmer to do this in the constructor of the class.
2. The value -2147483648 cannot be used as a default value, as it is used internally to denote `nodefault`.
3. It is not possible to specify a default for array properties.
4. It is not possible to specify the `Stored` directive for array properties.
5. All storage specifiers can be overridden in descendent classes.

   ```
   property Test stored False;
   ```

6. The `nodefault` specifier can be used to override the implicit `default` specifier of string, floating-point and pointer properties.
7. The streaming mechanism described here is the one implemented in the classes unit (see Unit Reference). Other streaming mechanisms can be implemented, and they can use the RTTI information in a different way.

6.8.7 Overriding and redeclaring properties

Properties can be both overridden and redeclared in descendent classes.

Property *redeclaration* takes action if the property type is declared, otherwise it is property *override*. The only difference is that property *override* replaces or extends the inherited modifiers with the new modifiers, whereas property *redeclaration* hides all inherited modifiers that are not present in the redeclaration. The type of the redeclared property does not have to be the same as the parent"s class property type.

The example below demonstrates the difference between property *override* and *redeclaration*.

```
type
  TAncestor = class
  private
    FP1 : Integer;
  public
    property P: integer Read FP1 write FP1;
  end;

  TP1 = class(TAncestor)
  public
    // property override
    property P default 1;
  end;

  TPReadOnly = class(TAncestor)
  public
    // property redeclaration
    property P: integer Read FP1;
  end;
```

`TP1` extends `property P` with a default value, `TPReadOnly` redeclares `property P` as read-only.

Remark `TP1` should set the default value of `P` to `1` in its constructor.

In case of both property *redeclaration* and property *override*, the access to the getter and setter is always static. I.e. property *override* acts only on the RTTI of the object and is not to be confused with method override.

The keyword "inherited" can be used to refer to the parent definition of the property. For example consider the following code:

```
type
  TAncestor = class
  private
    FP1 : Integer;
  public
    property P: integer Read FP1 write FP1;
  end;

  TClassA = class(TAncestor)
  private
    procedure SetP(const AValue: char);
    function getP : Char;
  public
    constructor Create;
```

```
    property P: char Read GetP write SetP;
  end;

procedure TClassA.SetP(const AValue: char);

begin
  Inherited P:=Ord(AValue);
end;

function TClassA.GetP : char;

begin
  Result:=Char((Inherited P) and $FF);
end;
```

TClassA redefines `P` as a character property instead of an integer property, but uses the parent"s `P` property to store the value.

Care must be taken when using virtual get/set routines for a property: setting the inherited property still observes the normal rules of inheritance for methods. Consider the following example:

```
type
  TAncestor = class
  private
    procedure SetP1(const AValue: integer); virtual;
  public
    property P: integer write SetP1;
  end;

  TClassA = class(TAncestor)
  private
    procedure SetP1(const AValue: integer); override;
    procedure SetP2(const AValue: char);
  public
    constructor Create;
    property P: char write SetP2;
  end;

constructor TClassA.Create;
begin
  inherited P:=3;
end;
```

In this case, when setting the inherited property `P`, the implementation `TClassA.SetP1` will be called, because the `SetP1` method is overridden.

If the parent class implementation of `SetP1` must be called, then this must be called explicitly:

```
constructor TClassA.Create;
begin
  inherited SetP1(3);
end;
```

The redeclared ancestor properties are also available from inside and outside the descendant object with a direct cast to the ancestor:

```
function GetP(const AClassA: TClassA): Integer;
begin
  Result := TAncestor(AClassA).P;
end;
```

6.9 Class properties

Class properties are very much like global property definitions. They are associated with the class, not with an instance of the class.

A consequence of this is that the storage for the property value must be a class var, not a regular field or variable of the class: normal fields or variables are stored in an instance of the class.

Class properties can have a getter and setter method like regular properties, but these must be static methods of the class.

That means that the following contains a valid class property definition:

```
TA = Class(TObject)
Private
  class var myprivatea : integer;
  class Function GetB : Integer;  static;
  class Procedure SetA(AValue : Integer); static;
  class Procedure SetB(AValue : Integer); static;
public
  Class property MyA : Integer Read MyPrivateA Write SetA;
  Class property MyA : Integer Read GetB Write SetB;
end;
```

The reason for the requirement is that a class property is associated to the particular class in which it is defined, but not to descendent classes. Since class methods can be virtual, this would allow descendent classes to override the method, making them unsuitable for class property access.

6.10 Nested types, constants and variables

A class definition can contain a type section, const section and a variable section. The type and constant sections act as a regular type section as found in a unit or method/function/procedure implementation. The variables act as regular fields of the class, unless they are in a `class var` section, in which case they act as if they were defined at the unit level, within the namespace of the class (section 6.3, page 96).

However, the visibility of these sections does play a role: private and protected (strict or not) constants, types and variables can only be used as far as their visibility allows.

Public types can be used outside the class, by their full name:

```
type
  TA = Class(TObject)
  Public
    Type TEnum = (a,b,c);
    Class Function DoSomething : TEnum;
  end;

Class Function TA.DoSomething : TEnum;
```

```
begin
  Result:=a;
end;

var
  E : TA.TEnum;

begin
  E:=TA.DoSomething;
end.
```

Whereas

```
type
  TA = Class(TObject)
  Strict Private
    Type TEnum = (a,b,c);
  Public
    Class Function DoSomething : TEnum;
  end;

Class Function TA.DoSomething : TEnum;

begin
  Result:=a;
end;

var
  E : TA.TEnum;

begin
  E:=TA.DoSomething;
end.
```

Will not compile and will return an error:

```
tt.pp(20,10) Error: identifier idents no member "TEnum"
```

Note that for writeable constants, the same rules apply as for class variables with regard to scope and overriding in descendents:

```
{$mode delphi}{$J+}
type
  TA = class // base type
    const CODE: integer = 99;
  end;
  TB = class(TA);
  TC = class(TA);

begin
  TA.Code:=0;
  TB.Code:=1;
  TC.Code:=2;
```

```
  Writeln(Ta.Code:2,Tb.Code:2,Tc.code:2);
end.
```

Will write

```
 2 2 2
```

But

```
{$mode delphi}{$J+}
type
  TA = class // base type
    const CODE: integer = -99;
  end;
  TB = class(TA)
    const code : integer = -98;
  end;
  TC = class(TA)
    Const code : integer = -97;
  end;

begin
  TA.Code:=0;
  TB.Code:=1;
  TC.Code:=2;
  Writeln(Ta.Code:2,Tb.Code:2,Tc.code:2);
end.
```

Will write

```
 0 1 2
```

Chapter 7

Interfaces

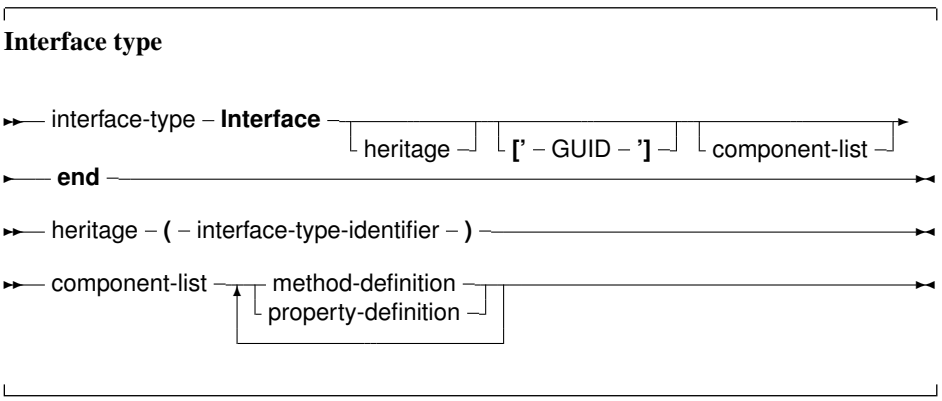
7.1 Definition

As of version 1.1, FPC supports interfaces. Interfaces are an alternative to multiple inheritance (where a class can have multiple parent classes) as implemented for instance in C++. An interface is basically a named set of methods and properties: a class that *implements* the interface provides *all* the methods as they are enumerated in the Interface definition. It is not possible for a class to implement only part of the interface: it is all or nothing.

Interfaces can also be ordered in a hierarchy, exactly as classes: an interface definition that inherits from another interface definition contains all the methods from the parent interface, as well as the methods explicitly named in the interface definition. A class implementing an interface must then implement all members of the interface as well as the methods of the parent interface(s).

An interface can be uniquely identified by a GUID. GUID is an acronym for Globally Unique Identifier, a 128-bit integer guaranteed always to be unique[1]. Especially on Windows systems, the GUID of an interface can and must be used when using COM.

The definition of an Interface has the following form:



Along with this definition the following must be noted:

- Interfaces can only be used in `DELPHI` mode or in `OBJFPC` mode.
- There are no visibility specifiers. All members are public (indeed, it would make little sense to make them private or protected).

[1] In theory, of course.

- The properties declared in an interface can only have methods as read and write specifiers.
- There are no constructors or destructors. Instances of interfaces cannot be created directly: instead, an instance of a class implementing the interface must be created.
- Only calling convention modifiers may be present in the definition of a method. Modifiers as `virtual`, `abstract` or `dynamic`, and hence also `override` cannot be present in the interface definition.

The following are examples of interfaces:

```
IUnknown = interface ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const iid : tguid;out obj) : longint;
  function _AddRef : longint;
  function _Release : longint;
end;
IInterface = IUnknown;

IMyInterface = Interface
  Function MyFunc : Integer;
  Function MySecondFunc : Integer;
end;
```

As can be seen, the GUID identifying the interface is optional.

7.2 Interface identification: A GUID

An interface can be identified by a GUID. This is a 128-bit number, which is represented in a text representation (a string literal):

```
['{HHHHHHHH-HHHH-HHHH-HHHH-HHHHHHHHHHHH}']
```

Each `H` character represents a hexadecimal number (0–9, A–F). The format contains 8-4-4-4-12 numbers. A GUID can also be represented by the following record, defined in the objpas unit (included automatically when in `DELPHI` or `OBJFPC` mode):

```
PGuid = ^TGuid;
TGuid = packed record
   case integer of
      1 : (
           Data1 : DWord;
           Data2 : word;
           Data3 : word;
           Data4 : array[0..7] of byte;
          );
      2 : (
           D1 : DWord;
           D2 : word;
           D3 : word;
           D4 : array[0..7] of byte;
          );
      3 : ( { uuid fields according to RFC4122 }
           time_low : dword;
```

```
            time_mid : word;
            time_hi_and_version : word;
            clock_seq_hi_and_reserved : byte;
            clock_seq_low : byte;
            node : array[0..5] of byte;
            );
end;
```

A constant of type TGUID can be specified using a string literal:

```
{$mode objfpc}
program testuid;

Const
  MyGUID : TGUID = '{10101010-1010-0101-1001-110110110110}';

begin
end.
```

Normally, the GUIDs are only used in Windows, when using COM interfaces. More on this in the next section.

7.3 Interface implementations

When a class implements an interface, it should implement all methods of the interface. If a method of an interface is not implemented, then the compiler will give an error. For example:

```
Type
  IMyInterface = Interface
    Function MyFunc : Integer;
    Function MySecondFunc : Integer;
  end;

  TMyClass = Class(TInterfacedObject,IMyInterface)
    Function MyFunc : Integer;
    Function MyOtherFunc : Integer;
  end;

Function TMyClass.MyFunc : Integer;

begin
  Result:=23;
end;

Function TMyClass.MyOtherFunc : Integer;

begin
  Result:=24;
end;
```

will result in a compiler error:

```
Error: No matching implementation for interface method
"IMyInterface.MySecondFunc:LongInt" found
```

Normally, the names of the methods that implement an interface, must equal the names of the methods in the interface definition. The compiler will look for matching methods in all visible methods: the methods of the class, and in parent classes methods with visibility protected or higher.

However, it is possible to provide aliases for methods that make up an interface: that is, the compiler can be told that a method of an interface is implemented by an existing method with a different name. This is done as follows:

```
Type
  IMyInterface = Interface
    Function MyFunc : Integer;
  end;

  TMyClass = Class(TInterfacedObject,IMyInterface)
    Function MyOtherFunction : Integer;
    Function IMyInterface.MyFunc = MyOtherFunction;
  end;
```

This declaration tells the compiler that the `MyFunc` method of the `IMyInterface` interface is implemented in the `MyOtherFunction` method of the `TMyClass` class.

7.4 Interface inheritance

It is possible to let one interface be a descendent from another interface:

```
IParentInterface = interface
  ['{0F78D56E-85A6-4024-98D7-720D7C7B9573}']
  procedure Foo;
end;

IChildInterface = interface(IParentInterface)
  ['{1AB2EB85-6843-462E-8CE4-32ECC065011E}']
  procedure Bar;
end;
```

`IChildInterface` will have two methods: `foo` and `bar`. Any class implementing this interface will therefore need to implement both interfaces:

```
TImplementor = class(TInterfacedObject, IChildInterface)
public
  procedure Foo;
  procedure Bar;
end;

procedure TImplementor.Foo;
begin

end;

procedure TImplementor.Bar;
begin

end;
```

Note that when a class declares a child interface, it can be assigned to a variable with the child interface. Given the above declarations, the following will compile:

```
var
  Child: IChildInterface;

begin
  Child := TImplementor.Create;
```

But this does not imply that it automatically also is assignment compatible with a variable with the type of parent interface. The following will not compile:

```
var
  Parent: IParentInterface;

begin
  Parent := TImplementor.Create;
```

To make this compile, it is necessary to declare the class as:

```
TImplementor = class(TInterfacedObject,
                     IParentInterface,
                     IChildInterface)
public
  procedure Foo;
  procedure Bar;
end;
```

The reason for this is that although the class actually implements the methods of `IParentInterface`, the compiler checks only actually declared interfaces when checking assignment compatibility: all declared interfaces are put in a table and only the contents of this table is checked.

The same check is performed at runtime: the compiler generates a table of all interfaces a class declares, and this table is checked at runtime. That means that although the following will compile if only `IChildInterface` is declared:

```
 ParentImplementorInstance := (TImplementor.Create as IParentInterface);
```

it will still fail with a run-time error:

```
home:~> ./ti
An unhandled exception occurred at $0000000000411A27:
EInvalidCast: Invalid type cast
$0000000000411A27
```

7.5 Interface delegation

Sometimes, the methods of an interface are implemented by a helper (or delegate) object, or the class instance has obtained an interface pointer for this interface and that should be used. This can be for instance when an interface must be added to a series of totally unrelated classes: the needed interface functionality is added to a separate class, and each of these classes uses an instance of the helper class to implement the functionality.

In such a case, it is possible to instruct the compiler that the interface is not implemented by the object itself, but actually resides in a helper class or interface. This can be done with the `implements` property modifier.

If the class has a pointer to the desired interface, the following will instruct the compiler that when the `IMyInterface` interface is requested, it should use the reference in the field:

```
type
  IMyInterface = interface
    procedure P1;
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
  private
    FMyInterface: IMyInterface; // interface type
  public
    property MyInterface: IMyInterface
       read FMyInterface implements IMyInterface;
  end;
```

The interface should not necessarily be in a field, any read identifier can be used.

If the interface is implemented by a delegate object, (a helper object that actually implements the interface) then it can be used as well with the `implements` keyword:

```
{$interfaces corba}
type
  IMyInterface = interface
    procedure P1;
  end;

  // NOTE: Interface must be specified here
  TDelegateClass = class(TObject, IMyInterface)
  private
    procedure P1;
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
  private
    FMyInterface: TDelegateClass; // class type
    property MyInterface: TDelegateClass
      read FMyInterface implements IMyInterface;
  end;
```

Note that in difference with Delphi, the delegate class must explicitly specify the interface: the compiler will not search for the methods in the delegate class, it will simply check if the delegate class implements the specified interface.

It is possible to implement multiple interfaces using a single delegated object:

```
{$interfaces corba}
type
  IMyInterface = interface
    procedure P1;
  end;
  IMyInterface1 = interface
```

```
    procedure P2;
  end;

  // NOTE: Interface must be specified here
  TDelegateClass = class(TObject, IMyInterface,IMyInterface1)
  private
    procedure P1;
    procedure P2;
  end;

  TMyClass = class(TInterfacedObject, IMyInterface, IMyInterface1)
  private
    FMyInterface: TDelegateClass; // class type
    property MyInterface: TDelegateClass
      read FMyInterface implements IMyInterface,IMyInterface1;
  end;
```

It is not possible to mix method resolution and interface delegation. That means, it is not possible to implement part of an interface through method resolution and implement part of the interface through delegation. The following attempts to implement `IMyInterface` partly through method resolution (P1), and partly through delegation. The compiler will not accept the following code:

```
{$interfaces corba}
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;


  TMyClass = class(TInterfacedObject, IMyInterface)
    FI : IMyInterface;
  protected
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
    property MyInterface: IMyInterface  read FI implements IMyInterface;
  end;
```

The compiler will throw an error:

```
Error: Interface "IMyInterface" can't be delegated by "TMyClass",
it already has method resolutions
```

However, it is possible to implement one interface through method resolution, and another through delegation:

```
{$interfaces corba}
type
  IMyInterface = interface
    procedure P1;
  end;

  IMyInterface2 = interface
    procedure P2;
```

```
  end;

  TMyClass = class(TInterfacedObject,
                   IMyInterface, IMyInterface2)
    FI2 : IMyInterface2;
  protected
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
  public
    property MyInterface: IMyInterface2
       read FI2 implements IMyInterface2;
  end;
```

Note that interface delegation can be used to specify that a class implements parent interfaces:

```
IGMGetFileName = interface(IUnknown)
  ['{D3ECCB42-A563-4cc4-B375-79931031ECBA}']
  function GetFileName: String; stdcall;
  property FileName: String read GetFileName;
end;

IGMGetSetFileName = Interface(IGMGetFileName)
  ['{ECFB879F-86F6-41a3-A685-0C899A2B5BCA}']
  procedure SetFileName(const Value: String); stdcall;
  property FileName: String read GetFileName write SetFileName;
end;

TIntfDelegator = class(TInterfacedObject, IGMGetFileName, IGMGetSetFileName)
 protected
  FGetSetFileName: IGMGetSetFileName;
 public
  constructor Create;
  destructor Destroy; override;
  property Implementor: IGMGetSetFileName read FGetSetFileName
    implements IGMGetFileName, IGMGetSetFileName;
end;
```

7.6 Interfaces and COM

When using interfaces on Windows which should be available to the COM subsystem, the calling convention should be `stdcall` – this is not the default Free Pascal calling convention, so it should be specified explicitly.

COM does not know properties. It only knows methods. So when specifying property definitions as part of an interface definition, be aware that the properties will only be known in the Free Pascal compiled program: other Windows programs will not be aware of the property definitions.

7.7 Dispatch Interfaces

Instead of the `Interface` keyword, you can also use the `dispinterface` keywords. This is specially for COM interfaces that implement the `IDispatch` interface: This means the compiler will not attempt to locate a method address in the interface table, but instead will perform the call

using the `Invoke` method offered by `IDispatch`. To this end, you must specify the dispatch ID for each method, as follows: This will compile on any platform but will work only on Windows, as only windows has the COM framework.

```
type
  IADsCollectionDisp = dispinterface
    ['{72B945E0-253B-11CF-A988-00AA006BC149}']
    property _NewEnum: IUnknown readonly dispid -4;
    procedure Add(const bstrName: WideString; vItem: OleVariant); dispid 4;
    procedure Remove(const bstrItemToBeRemoved: WideString); dispid 5;
    function  GetObject(const bstrName: WideString): OleVariant; dispid 6;
  end;
\end{verbatul}
Only external COM interface definitions can be dispatch interfaces, classes cannot i

Note that the parameter and result types must be automatable, meaning they can be an

You can store a dispatch interface in an OleVariant, and then call the methods of th
\begin{verbatim}
var
   WordFileName, PDFFileName: String;
   WordApplication, WordFile: OleVariant;
begin
   WordApplication := Null;
   WordFile := Null;
   WordFileName := 'mydoc.docx';
   PDFFileName := ChangeFileExt(WordFileName, '.pdf');
   WordApplication := CreateOleObject('Word.Application');
   If VarIsNull(WordApplication) then
     exit;
   WordApplication.Visible := False;
   WordApplication.DisplayAlerts := False;
   WordFile := WordApplication.Documents.Open(WordFileName);
   If VarIsNull(WordFile) then
     exit;
   WordFile.SaveAs2(PDFFileName, 17); //wdFormatPDF = 17
   WordFile.Close;
   WordApplication.Quit;
end;
```

The compiler will insert the necessary code to translate the above calls to IDispatch method calls.

7.8 CORBA and other Interfaces

COM is not the only architecture where interfaces are used. CORBA knows interfaces, UNO (the OpenOffice API) uses interfaces, and Java as well. These languages do not know the `IUnknown` interface used as the basis of all interfaces in COM. It would therefore be a bad idea if an interface automatically descended from `IUnknown` if no parent interface was specified. Therefore, a directive `{$INTERFACES}` was introduced in Free Pascal: it specifies what the parent interface is of an interface, declared without parent. More information about this directive can be found in the Programmer's Guide.

Note that COM interfaces are by default reference counted, because they descend from `IUnknown`.

Corba interfaces are identified by a simple string so they are assignment compatible with strings and not with `TGUID`. The compiler does not do any automatic reference counting for the CORBA interfaces, so the programmer is responsible for any reference bookkeeping.

7.9 Reference counting

All COM interfaces use reference counting. This means that whenever an interface is assigned to a variable, its reference count is updated. Whenever the variable goes out of scope, the reference count is automatically decreased. When the reference count reaches zero, usually the instance of the class that implements the interface, is freed.

Care must be taken with this mechanism. The compiler may or may not create temporary variables when evaluating expressions, and assign the interface to a temporary variable, and only then assign the temporary variable to the actual result variable. No assumptions should be made about the number of temporary variables or the time when they are finalized – this may (and indeed does) differ from the way other compilers (e. g. Delphi) handle expressions with interfaces. E. g. a type cast is also an expression:

```
Var
  B : AClass;

begin
  // ...
  AInterface(B.Intf).testproc;
  // ...
end;
```

Assume the interface `intf` is reference counted. When the compiler evaluates `B.Intf`, it creates a temporary variable. This variable may be released only when the procedure exits: it is therefore invalid to e. g. free the instance `B` prior to the exit of the procedure, since when the temporary variable is finalized, it will attempt to free `B` again.

Additionally, function results may point to a non-nil valid COM interface on entry: this is because the function result is treated as a var parameter.

Chapter 8

Generics

8.1 Introduction

Generics are templates for generating other types. These can be classes, objects, interfaces and even functions, arrays, records. It is a concept that comes from C++, where it is deeply integrated in the language. As of version 2.2, Free Pascal also officially has support for templates or Generics. They are implemented as a kind of macro which is stored in the unit files that the compiler generates, and which is replayed as soon as a generic class is specialized.

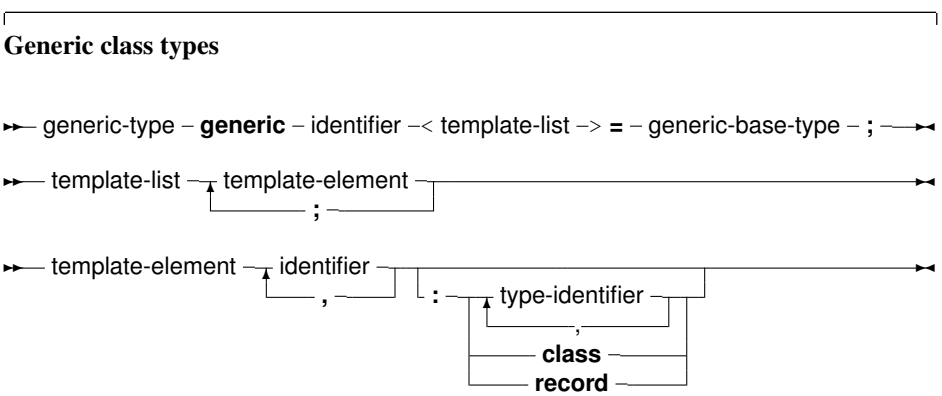
Creating and using generics is a 2-phase process.

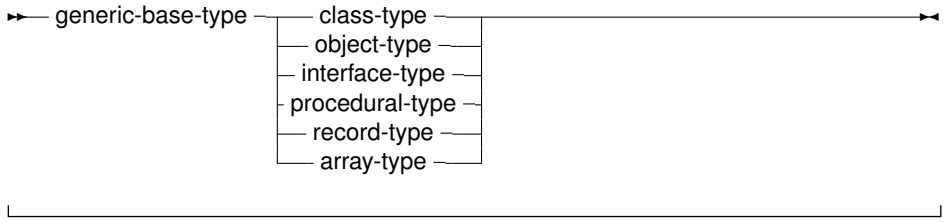
1. The definition of the generic is defined as a new type: this is a code template, a macro which can be replayed by the compiler at a later stage.
2. A generic type is specialized: this defines a second type, which is a specific implementation of the generic type: the compiler replays the macro which was stored when the generic type was defined.

There are several units distributed with free pascal that implement generic containers and classes. For example the fgl unit.

8.2 Generic type definition

A generic type definition is much like a type definition, with the exception that it contains a list of placeholders for types, as can be seen in the following syntax diagram:





In Delphi mode, the keyword `generic` is not needed. The presence of the template list is sufficient to designate a generic type.

For classes, objects, procedural types and extended records, the generic type declaration should be followed by a type implementation.

It is the same as a normal class implementation with a single exception, namely that any identifier with the same name as one of the template identifiers must be a type identifier.

So, the generic type declaration is much like a normal type declaration, except that there is an as yet unknown type present. The unknown types are listed in the placeholder list, and they are unknown until the class is specialized.

The following is a valid generic class definition:

```
Type
  generic TList<_T>=class(TObject)
  Public
    type
       TCompareFunc = function(const Item1, Item2: _T): Integer;
    var
      data : _T;
    procedure Add(item: _T);
    procedure Sort(compare: TCompareFunc);
  end;
```

This class could be followed by an implementation as follows:

```
procedure TList.Add(item: _T);
begin
  data:=item;
end;

procedure TList.Sort(compare: TCompareFunc);
begin
  if compare(data, 20) <= 0 then
    halt(1);
end;
```

There are some noteworthy things about this declaration and implementation:

1. There is a single placeholder `_T`. It will be substituted by a type identifier when the generic class is specialized. The identifier `_T` may not be used for anything else than a type placeholder. This means that the following would be invalid:

   ```
   procedure TList.Sort(compare: TCompareFunc);

   Var
     _t : integer;
   ```

```
begin
  // do something.
end;
```

2. The local type block contains a single type `TCompareFunc`. Note that the actual type is not yet known inside the generic class definition: the definition contains a reference to the placeholder `_T`. All other identifier references must be known when the generic class is defined, *not* when the generic class is specialized.

3. The local variable block is equivalent to the following:

```
generic TList<_T>=class(TObject)
Public
  type
     TCompareFunc = function(const Item1, Item2: _T): Integer;
Public
  data : _T;
  procedure Add(item: _T);
  procedure Sort(compare: TCompareFunc);
end;
```

4. Note that the type parameter names in declaration and implementation must be the same.

Not only generic classes can be defined, but also other types:

```
{$mode objfpc}
{$INTERFACES CORBA}
type

  generic PlanarCoordinate<t> = record
    x,y : t;
  end;

  TScreenCoordinate = specialize PLanarCoordinate<word>;
  TDiscreteCoordinate = specialize PlanarCoordinate<integer>;
  TRealCoordinate = specialize PlanarCoordinate<extended>;

  generic TDistanceFunction<t> = function (x,y : t) : Extended of object;

  TScreenDistance = specialize TDistanceFunction<word>;
  TDiscreteDistance = specialize TDistanceFunction<integer>;
  TRealDistance = specialize TDistanceFunction<Extended>;

  generic TArray<t> = array of t;

  TMyIntegerArray = specialize TArray<integer>;

  generic IList<_T> = Interface
    Function GetItem(AIndex : Integer) : _T;
    Procedure SetItem(AIndex : Integer; AValue : _T);
    Function GetCount : Integer;
    Property Items [AIndex : Integer] : _T Read GetItem Write SetItem;
    Property Count : Integer Read GetCount;
```

```
end;

generic TList<_T>=class(TObject, specialize IList<_T>)
public type
  TCompareFunc = function(const Item1, Item2: _T): Integer;
  Function GetItem(AIndex : Integer) : _T;
  Procedure SetItem(AIndex : Integer; AValue : _T);
  Function GetCount : Integer;
Public
   data : _T;
   procedure Add(item: _T);
   procedure Sort(compare: TCompareFunc);
end;

generic TPointSet<t> = array of specialize PlanarCoordinate<t>;

TScreenPointSet = specialize TPointSet<word>;
TDiscretePointSet =  specialize TPointSet<integer>;
TRealPointSet = specialize TPointSet<extended>;
```

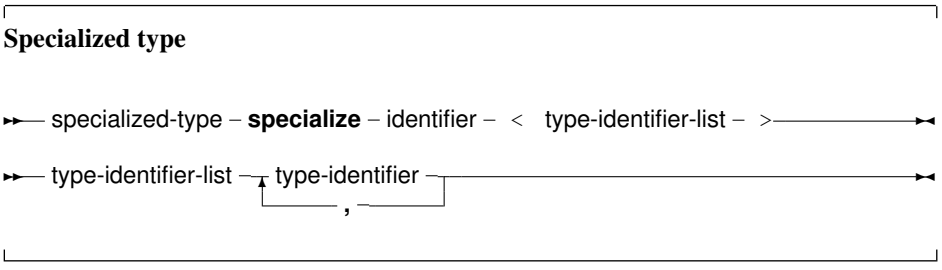
Remark A word on visibility: the template types `T` or `_T` are available as strict private types. That means that the types are not available in the descendent classes, unless they are made available through some protected or private mechanism, as in the following example:

```
generic TList<_T>=class(TObject)
public type
  TItemType = _T;
end;
```

8.3 Generic type specialization

Once a generic type is defined, it can be used to generate other types: this is like replaying the definition of the types, with the template placeholders filled in with actual type definitions.

This can be done in any `Type` definition block. The specialized type looks as follows:



In Delphi mode, the `specialize` keyword must not be used.

The generic specialization is a very simple definition and can be used wherever a type definition would be used.

Given the declaration of `TList` in the previous section, the following would be a valid type definition:

```
Type
```

```
  TPointerList = specialize TList<Pointer>;
  TIntegerList = specialize TList<Integer>;
```

As of version 3.0 of Free Pascal, the specialize keyword can also be used in a variable declaration:

```
Var
  P : specialize TList<Pointer>;
```

The `specialize` keyword is part of the specialized type, so when using fully qualified names, the specialize keyword must be after the unit name and parent type names.

The type in the specialize statement must be known, except in another generic type definition. Given the two generic class definitions:

```
type
  Generic TMyFirstType<T1> = Class(TMyObject);
  Generic TMySecondType<T2> = Class(TMyOtherObject);
```

Then the following specialization is not valid:

```
type
  TMySpecialType = specialize TMySecondType<TMyFirstType>;
```

because the type `TMyFirstType` is a generic type, and thus not fully defined. The compiler will complain:

```
Error: Generics cannot be used as parameters when specializing generics
```

However, the following is allowed:

```
type
  TA = specialize TMyFirstType<Atype>;
  TB = specialize TMySecondType<TA>;
```

because `TA` is already fully defined when `TB` is specialized.

However, the `specialize` keyword can be used in another generic type definition as shown in the example above:

```
  generic TList<_T>=class(TObject, specialize IList<_T>)
```

and

```
  generic TPointSet<t> = array of specialize PlanarCoordinate<t>;
```

In these definitions, the specialization is only performed when the generic type itself is specialized, and at that time, the types are known.

Remark It is not possible to make a forward definition of a class which is a specialization of a generic, i.e. the following will not compile:

```
  TMyClass = Class;

  // Other declarations

  TMyClass = specialize TList<T>;
```

8.4 Generic type restrictions

The diagram in section 8.2, page 132 shows that the type template list can have extra specifiers for the types. This is especially useful for object types: if the template type must descend from a certain class, then this can be specified in the template list:

```
{$mode objfpc}
{$h+}
uses sysutils, classes;

Type
  generic TList<_T : TComponent> = class(TObject)
  public
    Type TCompareFunc = function(const Item1, Item2: _T): Integer;
  Public
    data : _T;
    procedure Add(item: _T);
    procedure Sort(compare: TCompareFunc);
 end;
```

Given the above definition, the following will compile:

```
TPersistentList = specialize TList<TComponent>;
```

But this will not compile

```
TPersistentList = specialize TList<TPersistent>;
```

The compiler will return an error:

```
Error: Incompatible types: got "TPersistent" expected "TComponent"
```

Multiple types can be grouped together:

```
Type
  generic TList<Key1,Key2 : TComponent; Value1 : TObject> = class(TObject)
```

Additionally, it is possible to specify more than one type identifier for class and interface type restrictions. If a class is specified, then the type used for the template must be equal to or descend from the indicated type:

```
Type
  generic TList<T: TComponent, IEnumerable> = class(TObject)
```

A class used to specialize `T` must descend from `TComponent` and must implement `IEnumerable`.

If an interface is specified, then the template type must implement at least this interface, but it can also be a descendent interface from this interface:

```
Type
  generic TGenList<T: IEnumerable> = class(TObject)

  IMyEnum = Interface (IEnumerable)
    Procedure DoMy;
```

```
  end;

  TList = specialize TGenList<IMyEnum>;
  TSomeList = Specialize TGenList<TList>;
```

Multiple interfaces can be specified, in that case the class type must implement all listed interfaces: It is possible to mix one class name with several interface names.

If no type restrictions are in effect, the compiler will assume that template types are *not* assignment compatible.

This is specially important when the generic class contains overloaded methods. Given the following generic type declaration:

```
type
  generic TTest<T1, T2> = class
    procedure Test(aArg: LongInt);
    procedure Test(aArg: T1);
    procedure Test(aArg: T2);
  end;
```

Specializing the above will compile if `T1` and `T2` are of two different types and neither is also `LongInt`. The following will compile:

```
T1 = specialize TTest<String, TObject>;
```

But the following two will not compile:

```
T2 = specialize TTest<String, String>;
```

or

```
T2 = specialize TTest<String, Longint>;
```

8.5 Generic procedures and functions

As can be seen in the syntax diagrams for generic procedures and functions, it is possible to define generic procedures and functions or generic methods. Where in a generic class type all methods of the class can contain the template, it is possible to have a non-generic class with a generic method:

```
uses typinfo;

type
  TMyEnum = (one,two,three);

  TTypeHelper = class(TObject)
    generic procedure PrintTypeName<T>(a : T);
  end;

generic procedure TTypeHelper.PrintTypeName<T>(a : T);
begin
  Writeln(PTypeInfo(typeInfo(T)) ^.name);
end;
```

```
var
  Helper : TTypeHelper;

begin
  Helper.specialize PrintTypeName<TMyEnum>(one);
end.
```

or, in Delphi syntax:

```
uses typinfo;

type
  TMyEnum = (one,two,three);

  TTypeHelper = class(TObject)
    procedure PrintTypeName<T>(a : T);
  end;

procedure TTypeHelper.PrintTypeName<T>(a : T);
begin
  Writeln(PTypeInfo(typeInfo(T)) ^.name);
end;

var
  Helper : TTypeHelper;

begin
  Helper.PrintTypeName<TMyEnum>(one);
end.
```

Both global procedures or functions can also be generic:

```
ses typinfo;

type
  TMyEnum = (one,two,three);

generic function GetTypeName<T>(a : T) : string;

begin
  Result:=PTypeInfo(typeInfo(T))^.name;
end;

begin
  Writeln(specialize GetTypeName<TMyEnum>(one));
end.
```

It is even possible to use them in Delphi mode:

```
uses typinfo;

type
  TMyEnum = (one,two,three);
```

```
function GetTypeName<T>(a : T) : string;

begin
  Result:=PTypeInfo(typeInfo(T))^.name;
end;

begin
  Writeln(GetTypeName<TMyEnum>(one));
end.
```

But this code is not compilable with Delphi. In delphi, the closest thing to this feature is to use a record or class with generic class methods:

```
uses typinfo;

type
  TMyEnum = (one,two,three);

  TTypeHelper = class(TObject)
    class procedure PrintTypeName<T>(a : T);
  end;

class procedure TTypeHelper.PrintTypeName<T>(a : T);
begin
  Writeln(PTypeInfo(typeInfo(T)) ^.name);
end;

begin
  TTypeHelper.PrintTypeName<TMyEnum>(one);
end.
```

8.6 Delphi compatibility

FPC's generics support is implemented somewhat different from Delphi. In this section the main differences are highlighted.

8.6.1 Syntax elements

The syntax shown in the syntax diagrams is the syntax required in the ObjFPC mode of the compiler. However, in Delphi mode, the `specialize` and `generic` keywords must not be used, as shown in the following example:

```
Type
  TTest<T> = Class(TObject)
  Private
    FObj : T;
  Public
    Property Obj : T Read FObj Write FObj;
  end;

  TIntegerTest = TTest<Integer>;
```

In difference with Mode Objfpc, the template type names must be repeated in method definitions.

```
Type
  TTest<T> = Class(TObject)
  Private
    FObj : T;
  Public
    Procedure DoIt;
    Property Obj : T Read FObj Write FObj;
  end;

Procedure TTest<T>.DoIt;

begin
end;
```

This requirement is directly related to the generic type overload capability mentioned in the next section.

8.6.2 Record type restrictions

In Delphi mode, the `record` type restrictions will also allow the use of simple types:

```
Type
  TList<_T : record> = class(TObject)
  public
    Type TCompareFunc = function(const Item1, Item2: _T): Integer;
  Public
    data : _T;
    procedure Add(item: _T);
    procedure Sort(compare: TCompareFunc);
 end;

TIntList = TList<Integer>;
```

The restriction is enforced when specializing the type. That means that the mode active when specializing a type determines whether a simple type can be used or not: if the restriction to record was compiled using ObjFPC mode, code written in Delphi mode can specialize it with a simple type anyway.

for example:

```
unit tg;

interface

{$mode objfpc}

Type
  generic TList<_T : record> = class(TObject)
  public
    Type TCompareFunc = function(const Item1, Item2: _T): Integer;
  Public
    data : _T;
    procedure Add(item: _T);
    procedure Sort(compare: TCompareFunc);
```

```
 end;

implementation

generic procedure TList<_T>.Add(item: _T);

begin

end;

generic  procedure TList<_T>.Sort(compare: TCompareFunc);

begin
end;

end.
```

can be used in {$MODE Delphi} for:

```
{$mode delphi}
uses tg;

Type
  TIntList = TList<Integer>;
begin
end.
```

8.6.3 Type overloads

Delphi mode allows generic type overloads. This means that it is possible to declare the same generic class with different template type lists. The following declarations are therefore possible:

```
Type
  TTest<T> = Class(TObject)
  Private
    FObj : T;
  Public
    Property Obj : T Read FObj Write FObj;
  end;

  TTest<T,S> = Class(TObject)
  Private
    FObj1 : T;
    FObj2 : S;
  Public
    Property Obj1 : T Read FObj1 Write FObj1;
    Property Obj2 : S Read FObj2 Write FObj2;
  end;
```

8.6.4 Name space considerations

In Delphi mode, the generics do not interfere with the namespace for variables, this means that the following will also compile:

```
Type
  TTest<T> = Class(TObject)
  Private
    FObj : T;
  Public
    Property Obj : T Read FObj Write FObj;
  end;

Var
  TTest : Integer;
```

However, this does not work for constants and functions.

8.7 Type compatibility

Whenever a generic class is specialized with a different set of template types, this results in a new, distinct type.

The types are assignment compatible if the same template types are used.

If the same types are used, and compiler can reuse a previously generated generic, it will reuse this previously generated generic, but this is not guaranteed to be possible: you may still have 2 separate generic types, but they will be assignment compatible.

Take the following generic definition:

```
{$mode objfpc}
unit ua;

interface

type
  Generic TMyClass<T> = Class(TObject)
    Procedure DoSomething(A : T; B : INteger);
  end;

Implementation

Procedure TMyClass.DoSomething(A : T; B : Integer);

begin
  // Some code.
end;

end.
```

And the following specializations:

```
{$mode objfpc}
unit ub;

interface

uses ua;
```

```
Type
  TB = Specialize TMyClass<string>;

implementation

end.
```

the following specializations is identical, but appears in a different unit:

```
{$mode objfpc}
unit uc;

interface

uses ua;

Type
  TB = Specialize TMyClass<string>;

implementation

end.
```

The following will then compile:

```
{$mode objfpc}
unit ud;

interface

uses ua,ub,uc;

Var
  B : ub.TB;
  C : uc.TB;

implementation

begin
  B:=C;
end.
```

The types ub.TB and uc.TB are assignment compatible. It does not matter that the types are defined in different units. They could be defined in the same unit as well:

```
{$mode objfpc}
unit ue;

interface

uses ua;

Type
  TB = Specialize TMyClass<string>;
```

```
  TC = Specialize TMyClass<string>;


Var
  B : TB;
  C : TC;

implementation

begin
  B:=C;
end.
```

Each specialization of a generic class with the same types as parameters is a new, distinct type, but these types are assignment compatible if the template types used to specialize them are equal.

If the specialization is with a different template type, the types are still distinct, but no longer assignment compatible. i. e. the following will not compile:

```
{$mode objfpc}
unit uf;

interface

uses ua;

Type
  TB = Specialize TMyClass<string>;
  TC = Specialize TMyClass<integer>;


Var
  B : TB;
  C : TC;

implementation

begin
  B:=C;
end.
```

When compiling, an error will result:

```
Error: Incompatible types: got "TMyClass<System.LongInt>"
                           expected "TMyClass<System.ShortString>"
```

8.8 Using the default intrinsic

When writing generic routines, sometimes a variable must be initialized whose type is not known during the declaration of the generic. This is where the `Default` intrinsic (section 4.5, page 77) also comes into play. Given the following generic declaration:

```
type
```

```
  generic TTest<T> = class
    procedure Test;
  end;
```

The following code will correctly initialize the variable `myt` during specialization:

```
procedure TTest.Test;
var
  myt: T;
begin
  // will have the correct Default if class is specialized
  myt := Default(T);
end;
```

8.9 A word about scope

Beside the template type, all other types used in the generic declaration must be known when the declaration is parsed.

This means that a type identifier with the same name must exist. The following unit will produce an error:

```
{$mode objfpc}
unit myunit;

interface

type
  Generic TMyClass<T> = Class(TObject)
    Procedure DoSomething(A : T; B : TSomeType);
  end;

Type
  TSomeType = Integer;
  TSomeTypeClass = specialize TMyClass<TSomeType>;

Implementation

Procedure TMyClass.DoSomething(A : T; B : TSomeType);

begin
  // Some code.
end;

end.
```

The above code will result in an error, because the type `TSomeType` is not known when the declaration is parsed:

```
home: >fpc myunit.pp
myunit.pp(8,47) Error: Identifier not found "TSomeType"
myunit.pp(11,1) Fatal: There were 1 errors compiling module, stopping
```

A second way in which this is visible, is the following. Assume a unit

```
{$mode objfpc}
unit mya;

interface

type
  Generic TMyClass<T> = Class(TObject)
    Procedure DoSomething(A : T);
  end;


Implementation

Procedure DoLocalThings;

begin
  Writeln('mya.DoLocalThings');
end;


Procedure TMyClass.DoSomething(A : T);

begin
  DoLocalThings;
end;

end.
```

The compiler will not allow to compile this unit, since the `DoLocalThings` function will not be visible when the generic type is specialized:

```
Error: Global Generic template references static symtable
```

Now, if the unit is modified, and the `DoLocalThings` function is moved to the interface section, the unit will compile. When using this generic in a program:

```
{$mode objfpc}
program myb;

uses mya;

procedure DoLocalThings;

begin
  Writeln('myb.DoLocalThings');
end;

Type
  TB = specialize TMyClass<Integer>;

Var
  B : TB;

begin
```

```
  B:=TB.Create;
  B.DoSomething(1);
end.
```

Despite the fact that generics act as a macro which is replayed at specialization time, the reference to `DoLocalThings` is resolved when `TMyClass` is defined, not when TB is defined. This means that the output of the program is:

```
home: >fpc -S2 myb.pp
home: >myb
mya.DoLocalThings
```

This behavior is dictated by safety and necessity:

1. A programmer specializing a class has no way of knowing which local procedures are used, so he cannot accidentally "override" it.
2. A programmer specializing a class has no way of knowing which local procedures are used, so he cannot implement it either, since he does not know the parameters.
3. If implementation procedures are used as in the example above, they cannot be referenced from outside the unit. They could be in another unit altogether, and the programmer has no way of knowing he should include them before specializing his class.

8.10 Operator overloading and generics

Operator overloading (chapter 15, page 246) and generics are closely related. Imagine a generic class that has the following definition:

```
{$mode objfpc}
unit mya;

interface

type
  Generic TMyClass<T> = Class(TObject)
    Function Add(A,B : T) : T;
  end;

Implementation

Function TMyClass.Add(A,B : T) : T;

begin
  Result:=A+B;
end;

end.
```

When the compiler replays the generics macro, the addition must be possible. For a specialization like this:

```
TMyIntegerClass = specialize TMyClass<integer>;
```

This is not a problem, as the `Add` method would become:

```
Procedure TMyIntegerClass.Add(A,B : Integer) : Integer;

begin
  Result:=A+B;
end;
```

The compiler knows how to add two integers, so this code will compile without problems. But the following code:

```
Type
  TComplex = record
   Re,Im : Double;
  end;

Type
  TMyIntegerClass = specialize TMyClass<TComplex>;
```

Will not compile, unless the addition of two `TComplex` types is defined. This can be done using record operators:

```
{$modeswitch advancedrecords}
uses mya;

Type
  TComplex = record
     Re,Im : Double;
     class operator +(a,b : TComplex) : TComplex;
  end;

class operator TComplex.+ (a,b : TComplex) : TComplex;

begin
  Result.re:=A.re+B.re;
  Result.im:=A.im+B.im;
end;


Type
  TMyComplexClass = specialize TMyClass<TComplex>;

begin
  // Code here
end.
```

Currently, due to an implementation restriction, it will not work using a global operator, i. e. the following does not work yet:

```
uses mya;

Type
  TComplex = record
     Re,Im : Double;
```

```
  end;

operator + (a,b : TComplex) : TComplex;

begin
  Result.re:=A.re+B.re;
  Result.im:=A.im+B.im;
end;

Type
  TMyComplexClass = specialize TMyClass<TComplex>;

begin
  // Code here
end.
```

Support for this construct is expected in a future version of Free Pascal.

Chapter 9

Extended records

9.1 Definition

Extended records are in many ways equivalent to objects and to a lesser extent to classes: they are records which have methods associated with them, and properties. They can also have local constant and type definitions.

Like objects, when defined as a variable, the extended records are allocated on the stack. They do not need to have a constructor.

Extended records have limitations over objects and classes in that they do not allow inheritance and polymorphism. It is impossible to create a descendant record of a record[1].

Why then introduce extended records ? They were introduced by Delphi 2005 to support one of the features introduced by .NET. Delphi deprecated the old TP style of objects, and re-introduced the features of .NET as extended records. Free Pascal aims to be Delphi compatible, so extended records are allowed in Free Pascal as well, but only in Delphi mode.

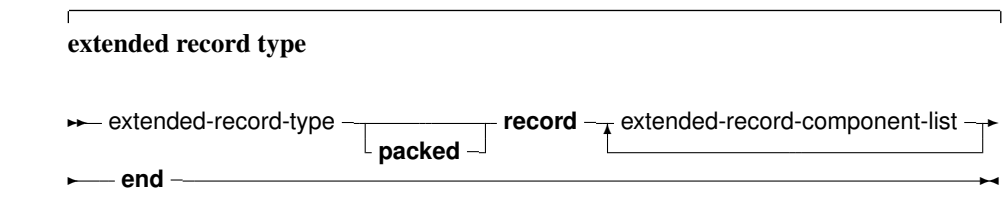
If extended records are desired in ObjFPC mode, then a mode switch must be used:

```
{$mode objfpc}
{$modeswitch advancedrecords}
```

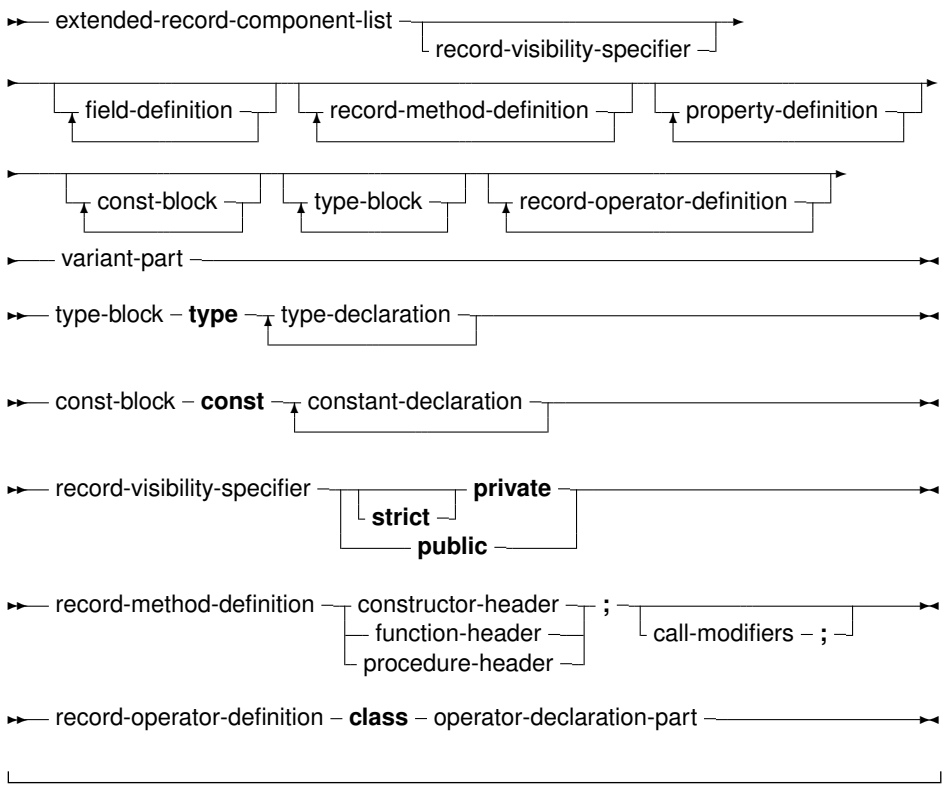
Compatibility is not the only reason for introducing extended records. There are some practical reasons for using methods or properties in records:

1. It is more in line with an object-oriented approach to programming: the type also contains any methods that work on it.
2. In contrast with a procedural approach, putting all operations that work on a record in the record itself, allows an IDE to show the available methods on the record when it is displaying code completion options.

Defining an extended record is much like defining an object or class:



[1] although it can be enhanced using record helpers, more about this in the chapter on record helpers.



Some of the restrictions when compared to classes or objects are obvious from the syntax diagram:

- No inheritance of records.
- No published and protected section exists.
- Class methods (if one can name them so) require the `static` keyword.
- Methods cannot be virtual or abstract – this is a consequence of the fact that there is no inheritance.

Other than that the definition much resembles that of a class or object, except that operators can be defined in an extended record.

As of version 3.2.4 of Free Pascal, extended records can have constructors, on the condition that they have parameters. A parameterless constructor is not allowed.

If no visibility is specified for a record, it is `public` by default, which is in line with 'classic' records in which all fields are always visible.

Remark In the case of a variant record, the `Case` keyword implicitly starts a `var` section, meaning class variables or methods are not allowed in the variant part of the record.

The following are few examples of valid extended record definitions:

```
TTest1 = record
  a : integer;
  function Test(aRecurse: Boolean): Integer;
end;

TTest2 = record
private
```

```
  A,b : integer;
public
  procedure setA(AValue : integer);
  property SafeA : Integer Read A Write SetA;
end;

TTest3 = packed record
private
  fA,fb : byte;
  procedure setA(AValue : Integer);
  function geta : integer;
public
  property A : Integer Read GetA Write SetA;
end;

TTest4 = record
 private
   a : Integer;
   function getp : integer;
 public
   b : string;
   procedure setp (aValue : integer);
   property p : integer read Getp Write SetP;
 public
 case x : integer of
   1 : (Q : string[10]);
   2 : (S : String[10]);
 end;
```

Note that it is possible to specify a visibility for the members of the record. This is particularly useful for example when creating an interface to a C library: the actual fields can be declared hidden, and more "pascal" like properties can be exposed which act as the actual fields. The `TTest3` record definition shows that the `packed` directive can be used in extended records. Extended records have the same memory layout as their regular counterparts: the methods and properties are not part of the record structure in memory.

The `TTest4` record definition in the above examples shows that the extended record still has the ability to define a variant part. As with the regular record, the variant part must come last. It cannot contain methods.

9.2 Extended record enumerators

Extended records can have an enumerator. To this end, a function returning an enumerator record must be defined in the extended record:

```
{$mode objfpc}
{$modeswitch advancedrecords}
type
  TIntArray = array[0..3] of Integer;

  TEnumerator = record
  private
    FIndex: Integer;
```

```
    FArray: TIntArray;
    function GetCurrent: Integer;
  public
    function MoveNext: Boolean;
    property Current: Integer read GetCurrent;
  end;

  TMyArray = record
    F: array[0..3] of Integer;
    function GetEnumerator: TEnumerator;
  end;

function TEnumerator.MoveNext: Boolean;
begin
  inc(FIndex);
  Result := FIndex < Length(FArray);
end;

function TEnumerator.GetCurrent: Integer;
begin
  Result := FArray[FIndex];
end;

function TMyArray.GetEnumerator: TEnumerator;
begin
  Result.FArray := F;
  Result.FIndex := -1;
end;
```

After these definitions, the following code will compile and enumerate all elements in F:

```
var
  Arr: TMyArray;
  I: Integer;
begin
  for I in Arr do
    WriteLn(I);
end.
```

The same effect can be achieved with the enumerator operator:

```
{$mode objfpc}
{$modeswitch advancedrecords}
type
  TIntArray = array[0..3] of Integer;

  TEnumerator = record
  private
    FIndex: Integer;
    FArray: TIntArray;
    function GetCurrent: Integer;
  public
    function MoveNext: Boolean;
    property Current: Integer read GetCurrent;
```

```
  end;

  TMyArray = record
    F: array[0..3] of Integer;
  end;

function TEnumerator.MoveNext: Boolean;
begin
  inc(FIndex);
  Result := FIndex < Length(FArray);
end;

function TEnumerator.GetCurrent: Integer;
begin
  Result := FArray[FIndex];
end;

operator Enumerator(const A: TMyArray): TEnumerator;
begin
  Result.FArray := A.F;
  Result.FIndex := -1;
end;
```

This will allow the code to run as well.

9.3 Record operators

Operator overloading is treated in detail in the chapter on operator overloading chapter 15, page 246. However, since Delphi implemented operator overloading as part of advanced records, a few words need to be said about it here.

As can be seen in the syntax diagram for extended records, FPC supports the Delphi syntax for operators on extended records. This syntax is available in both `ObjFPC` and `Delphi` mode.

In `objfpc` mode, the operators must be defined with their symbolic names, just as regular operator overloads:

```
{$mode objfpc}
{$modeswitch advancedrecords}

Type
  TComplex = record
    Re,Im : Double;
    class operator +(a,b : TComplex) : TComplex;
  end;

class operator TComplex.+ (a,b : TComplex) : TComplex;
begin
  Result.re:=A.re+B.re;
  Result.im:=A.im+B.im;
end;
```

The operators work just as they would if they were defined using the regular FPC syntax for operators, but this syntax is of course limited to record types. Note that they have the `class` keyword prefixed,

and have the record type name prepended in the implementation.

As indicated above, in `ObjFPC` mode, the operators must be indicated with their symbol names. By contrast, in `Delphi` mode, the names of the operators can also be used, similar to the syntax in Delphi:

```
{$mode delphi}
Type
  TComplex = record
    Re,Im : Double;
    class operator add(a,b : TComplex) : TComplex;
  end;

class operator TComplex.add (a,b : TComplex) : TComplex;
begin
  Result.re:=A.re+B.re;
  Result.im:=A.im+B.im;
end;
```

This is of course because the syntax must be compatible with Delphi.

Below is a table that links the symbolic operator names to the plain-text name. Note that some FPC operators do not have an equivalent usin a Delphi name.

Table 9.1: Operator names

Symbol	Name
+	add or positive
-	subtract or negative
*	multiply
/	divide
**	(no equivalent)
><	(no equivalent)
=	equal
<	lessthan
<=	lessthanorequal
>	greaterthan
>=	greaterthanorequal
<>	notequal
:=	implicit
in	in
shr	rightshift
shl	leftshift
div	intdivide
mod	modulus
and	bitwiseand or logicaland
or	bitwiseor or logicalor
xor	bitwisexor
not	logicalnot

For example, the power operator (**) can be used in `Delphi` mode using its symbolic name:

```
{$mode delphi}
```

```
{$modeswitch advancedrecords}

Type
  TComplex = record
    Re,Im : Double;
    class operator **(a,b : TComplex) : TComplex;
  end;

class operator TComplex.** (a,b : TComplex) : TComplex;
begin
  Result.re:=-B.re;
  Result.im:=-B.im;
end;

Var
  a,b : TComplex;
begin
  a:=a**b;
end.
```

More information on operator overloading can be found in the chapter on operator overloading chapter 15, page 246.

Chapter 10

Class, Record and Type helpers

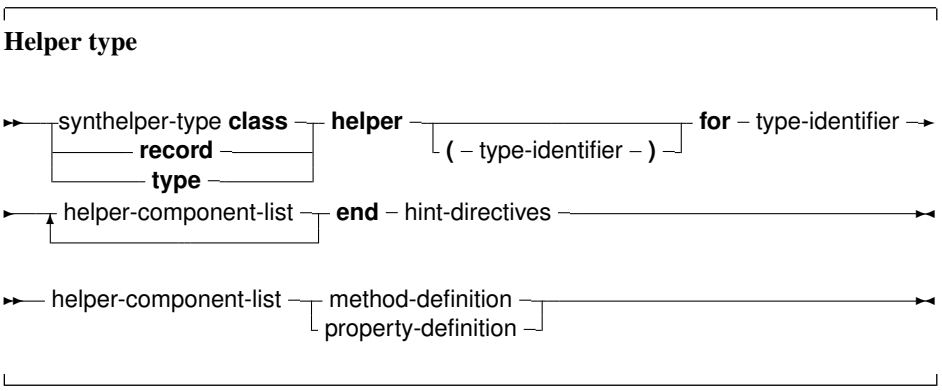
10.1 Definition

Class, record and type helpers can be used to add methods to an existing class, record or simple type, without making a derivation of the class or re-declaring the record.

For a record or a simple type, the type helper acts as if record or the simple type is a class, and methods are declared for it. Inside the methods, `Self` will refer to the value of the record or simple type.

For classes, the effect is like inserting a method in the method table of the class. If the helper declaration is in the current scope of the code, then the methods and properties of the helper can be used as if they were part of the class declaration for the class or record that the helper extends.

The syntax diagram for a class, record or type helper is presented below:



The diagram shows that a helper definition looks very much like a regular class definition. It simply declares some extra constructors, methods, properties and fields for a class: the class, record or simple type for which the helper is an extension is indicated after the `for` keyword.

Since an enumerator for a class is obtained through a regular method, class helpers can also be used to override the enumerators for the class.

As can be seen from the syntax diagram, it is possible to create descendants of helpers: the helpers can form a hierarchy of their own, allowing to override methods of a parent helper. They also have visibility specifiers, just like records and classes.

As in an instance of the class, the `Self` identifier in a method of a class helper refers to the class instance (not the helper instance). For a record, it refers to the record.

The following is a simple class helper for the `TObject` class, which provides an alternate version of the standard `ToString` method.

```
TObjectHelper = class helper for TObject
  function AsString(const aFormat: String): String;
end;

function TObjectHelper.AsString(const aFormat: String): String;
begin
  Result := Format(aFormat, [ToString]);
end;

var
  o: TObject;
begin
  Writeln(o.AsString('The object''s name is %s'));
end.
```

Remark The `helper` modifier is only a modifier just after the `class` or `record` keywords. That means that the first member of a class or record cannot be named `helper`. A member of a class or record can be called `helper`, it just cannot be the first one, unless it is escaped with an `&`, as for all identifiers that match a keyword.

Remark Support for type helpers requires the use of Delphi mode or the use of a `typehelpers` modeswitch in other modes:

```
{$MODESWITCH TYPEHELPERS}
```

Remark Support for record helpers requires the use of Delphi mode or the use of a `advancedrecords` modeswitch in other modes:

```
{$MODESWITCH ADVANCEDRECORDS}
```

10.2 Restrictions on class helpers

It is not possible to extend a class with any method or property. There are some restrictions on the possibilities:

- Destructors or class destructors are not allowed.
- Class constructors are not allowed.
- Class helpers cannot descend from record helpers, and cannot extend record types.
- Field definitions are not allowed. Neither are class fields.
- Properties that refer to a field are not allowed. This is in fact a consequence of the previous item.
- Abstract methods are not allowed.
- Virtual methods of the class cannot be overridden. They can be hidden by giving them the same name or they can be overloaded using the `overload` directive.
- Unlike for regular procedures or methods, the `overload` specifier must be explicitly used when overloading methods from a class in a class helper. If overload is not used, the extended type's method is hidden by the helper method (as for regular classes).

The following modifies the previous example by overloading the `ToString` method:

```
TObjectHelper = class helper for TObject
  function ToString(const aFormat: String): String; overload;
end;

function TObjectHelper.ToString(const aFormat: String): String;
begin
  Result := Format(aFormat, [ToString]);
end;

var
  o: TObject;
begin
  Writeln(o.ToString('The object''s name is %s'));
end.
```

10.3 Restrictions on record helpers

Records do not offer the same possibilities as classes do. This reflects on the possibilities when creating record helpers. Below the restrictions on record helpers are enumerated:

- A record helper cannot be used to extend a class. The following will fail:

  ```
  TTestHelper = record helper for TObject
  end;
  ```

- Inside a helper's declaration the methods/fields of the extended record can't be accessed in e. g. a property definition. They can be accessed in the implementation, of course. This means that the following will not compile:

  ```
  TTest = record
    Test: Integer;
  end;

  TTestHelper = record helper for TTest
    property AccessTest: Integer read Test;
  end;
  ```

- Record helpers can only access public fields (in case an extended record with visibility specifiers is used).
- Inheritance of record helpers is only allowed in ObjFPC mode; In Delphi mode, it is not allowed.
- Record helpers can only descend from other record helpers, not from class helpers.
- Unlike class helpers, a descendent record helper must extend the same record type.
- In Delphi mode, it is not possible to call the extended record's method using `inherited`. It is possible to do so in ObjFPC mode. The following code needs ObjFPC mode to compile:

  ```
  type
    TTest = record
  ```

```
    function Test(aRecurse: Boolean): Integer;
  end;

  TTestHelper = record helper for TTest
    function Test(aRecurse: Boolean): Integer;
  end;

function TTest.Test(aRecurse: Boolean): Integer;
begin
  Result := 1;
end;

function TTestHelper.Test(aRecurse: Boolean): Integer;
begin
  if aRecurse then
    Result := inherited Test(False)
  else
    Result := 2;
end;
```

10.4 Considerations for (simple) type helpers

For simple types, the rules are pretty much the same as for records, plus there are some extra requirements:

- Support for type helpers needs to be activated using the modeswitch typehelpers:

  ```
  {$modeswitch typehelpers}
  ```

  This modeswitch is enabled by default only in mode Delphi and DelphiUnicode.

- In Delphi (and DelphiUnicode) mode, for stricter Delphi compatibility, the record helpers must be used instead of a type helper.

- The modes ObjFPC and MacPas use type helper, but the modeswitch TypeHelpers must be used.

- The following types are not supported:

  - All file types (`Text`, `file of ...`)
  - Procedural variables
  - Types like records, classes, Objective C classes, C++ classes, objects and interfaces are forbidden as well, the class helper must be used for classes. That means that for instance, the following will fail:

    ```
    TTestHelper = type helper for TObject
    end;
    ```

  This of course means that all other simple types are supported.

- Type helpers can implement constructors.

- Inheritance of record helpers is only allowed in ObjFPC mode; In Delphi mode, it is not allowed.

- Type helpers can only descend from other type helpers, not from class or record helpers.
- A descendent type helper must extend the same type.

The following gives an idea of the possibilities:

```
{$mode objfpc}
{$modeswitch typehelpers}

type
 TLongIntHelper = type helper for LongInt
   constructor create(AValue: LongInt);
   class procedure Test; static;
   procedure DoPrint;
 end;

constructor TLongIntHelper.create(AValue: LongInt);

begin
  Self:=Avalue;
  DoPrint;
end;

class procedure TLongIntHelper.Test;

begin
   Writeln('Test');
end;

procedure TLongIntHelper.DoPrint;

begin
   Writeln('Value :',Self);
end;

var
  i: LongInt;
begin
  I:=123;
  i.Test;
  $12345678.Test;
  LongInt.Test;
  I:=123;
  i.DoPrint;
  $12345678.DoPrint;
end.
```

10.5 A note on scope and lifetime for record and type helpers

For classes, the lifetime of an instance of a class is explicitly managed by the programmer. It is therefore clear what the `Self` parameter means and when it is valid.

Records and other simple types are allocated on the stack, which means they go out of scope when the function, procedure or method in which they are used, exits.

Combined with the fact that helper methods are type compatible to class methods, and can therefore be used as event handlers, this can lead to surprising situations: The data pointer in a helper method is set to the address of the variable.

Consider the following example:

```
{$mode objfpc}
{$modeswitch typehelpers}
uses
  Classes;

type
  TInt32Helper = type helper for Int32
    procedure Foo(Sender: TObject);
  end;

procedure TInt32Helper.Foo(Sender: TObject);
begin
  Writeln(Self);
end;

var
  i: Int32 = 10;
  m: TNotifyEvent;
begin
  m := @i.Foo;
  WriteLn('Data : ',PtrUInt(TMethod(m).Data));
  m(nil);
end.
```

This will print something like (the actual value for data may differ):

```
Data : 6848896
10
```

The variable `i` is still in scope when `m` is called.

But changing the code to

```
{$mode objfpc}
{$modeswitch typehelpers}
uses
  Classes;

type
  TInt32Helper = type helper for Int32
    procedure Foo(Sender: TObject);
  end;

procedure TInt32Helper.Foo(Sender: TObject);
begin
  Writeln(Self);
end;

Function GetHandler  :TNotifyEvent;
```

```
var
  i: Int32 = 10;

begin
  Result:=@i.foo;
end;

Var
  m: TNotifyEvent;
begin
  m := GetHandler;
  WriteLn(PtrUInt(TMethod(m).Data));
  m(nil);
end.
```

The output will be:

```
140727246638796
0
```

The actual output will depend on the architecture, but the point is that `i` is no longer in scope, making the output of its value meaningless, and possibly even leading to access violations and program crashes.

10.6 Inheritance

As noted in the previous section, it is possible to create descendants of helper classes. Since only the last helper class in the current scope can be used, it is necessary to descend a helper class from another one if methods of both helpers must be used. More on this in a subsequent section.

A descendent of a class helper can extend a different class than its parent. The following is a valid class helper for `TMyObject`:

```
TObjectHelper = class helper for TObject
  procedure SomeMethod;
end;

TMyObject = class(TObject)
end;

TMyObjectHelper = class helper(TObjectHelper) for TMyObject
  procedure SomeOtherMethod;
end;
```

The `TMyObjectHelper` extends `TObjectHelper`, but does not extend the `TObject` class, it only extends the `TMyObject` class.

Since records know no inheritance, it is obvious that descendants of record helpers can only extend the same record.

Remark For maximum delphi compatibility, it is impossible to create descendants of record helpers in Delphi mode.

10.7 Usage

Once a helper class is defined, its methods can be used whenever the helper class is in scope. This means that if it is defined in a separate unit, then this unit should be in the uses clause wherever the methods of the helper class are used.

Consider the following unit:

```
{$mode objfpc}
{$h+}
unit oha;

interface

Type
  TObjectHelper = class helper for TObject
    function AsString(const aFormat: String): String;
  end;

implementation

uses sysutils;

function TObjectHelper.AsString(const aFormat: String): String;

begin
  Result := Format(aFormat, [ToString]);
end;

end.
```

Then the following will compile:

```
Program Example113;

uses oha;

{ Program to demonstrate the class helper scope. }

Var
  o : TObject;

begin
  O:=TObject.Create;
  Writeln(O.AsString('O as a string : %s'));
end.
```

But, if a second unit (ohb) is created:

```
{$mode objfpc}
{$h+}
unit ohb;

interface
```

```
Type
  TAObjectHelper = class helper for TObject
    function MemoryLocation: String;
  end;

implementation

uses sysutils;

function TAObjectHelper.MemoryLocation: String;

begin
  Result := format('%p',[pointer(Self)]);
end;

end.
```

And is added after the first unit in the uses clause:

```
Program Example113;

uses oha,ohb;

{ Program to demonstrate the class helper scope. }

Var
  o : TObject;

begin
  O:=TObject.Create;
  Writeln(O.AsString('O as a string : %s'));
  Writeln(O.MemoryLocation);
end.
```

Then the compiler will complain that it does not know the method “AsString”. This is because the compiler stops looking for class helpers as soon as the first class helper is encountered. Since the `ohb` unit comes last in the uses clause, the compiler will only use `TAObjectHelper` as the class helper.

The solution is to re-implement unit ohb:

```
{$mode objfpc}
{$h+}
unit ohc;

interface

uses oha;

Type
  TAObjectHelper = class helper(TObjectHelper) for TObject
    function MemoryLocation: String;
  end;

implementation
```

```
uses sysutils;

function TAObjectHelper.MemoryLocation: String;

begin
  Result := format('%p',[pointer(Self)]);
end;

end.
```

And after replacing unit `ohb` with `ohc`, the example program will compile and function as expected.

Note that it is not enough to include a unit with a class helper once in a project; The unit must be included whenever the class helper is needed.

Chapter 11

Objective-Pascal Classes

11.1 Introduction

The preferred programming language to access Mac OS X system frameworks is Objective-C. In order to fully realize the potential offered by system interfaces written in that language, a variant of Object Pascal exists in the Free Pascal compiler that tries to offer the same functionality as Objective-C. This variant is called Objective-Pascal.

The compiler has mode switches to enable the use of these Objective-C-related constructs. There are two kinds of Objective-C language features, discerned by a version number: Objective-C 1.0 and Objective-C 2.0.

The Objective-C 1.0 language features can be enabled by adding a modeswitch to the source file:

```
{$modeswitch objectivec1}
```

or by using the `-Mobjectivec1` command line switch of the compiler.

The Objective-C 2.0 language features can be enabled using a similar modewitch:

```
{$modeswitch objectivec2}
```

or the command-line option `-Mobjectivec2`.

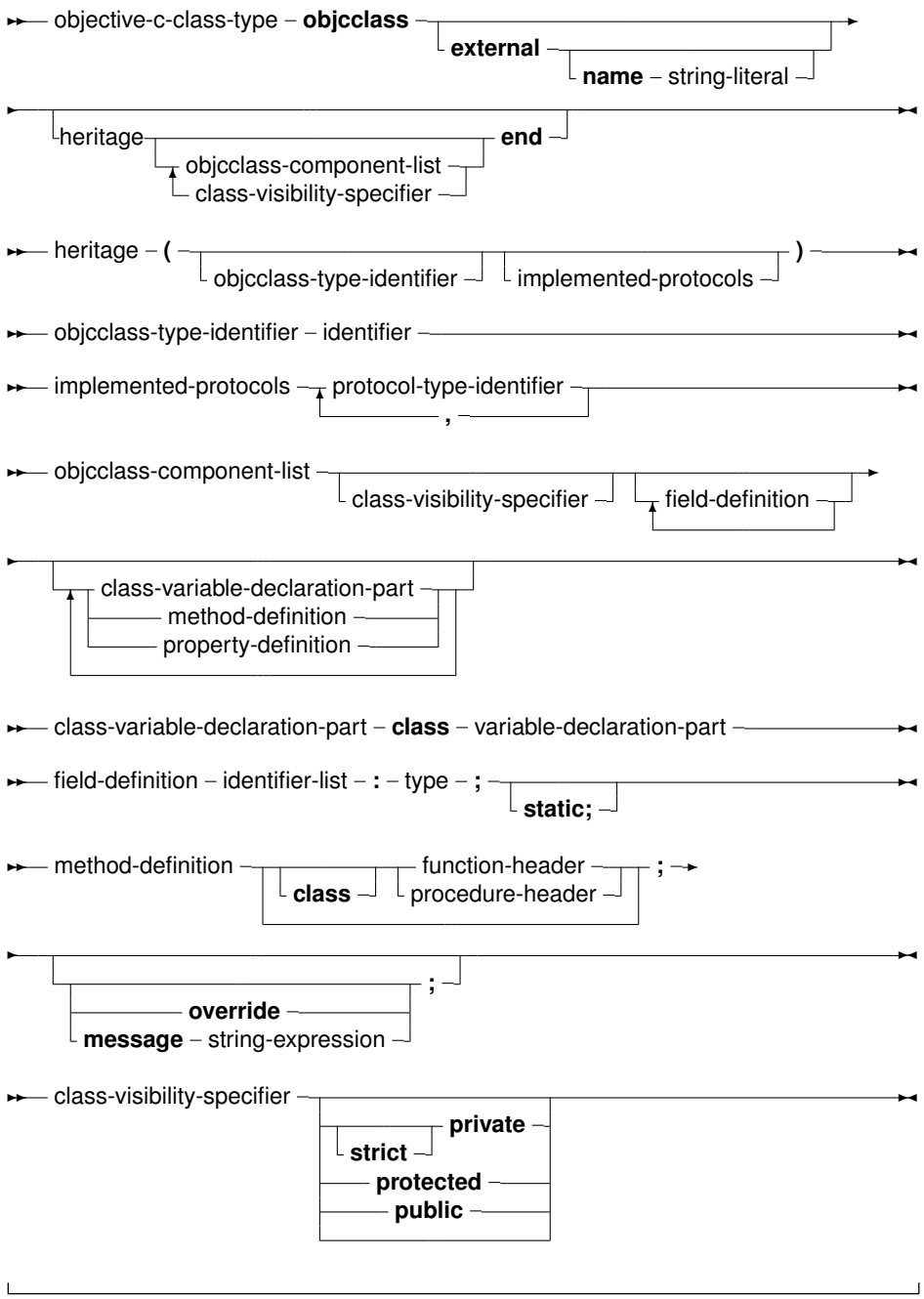
The Objective-C 2.0 language features are a superset of the Objective-C 1.0 language features, and therefore the latter switch automatically implies the former. Programs using Objective-C 2.0 language features will only work on Mac OS X 10.5 and later.

The fact that objective-C features are enabled using mode switches rather than actual syntax modes, means they can be used in combination with every general syntax mode (fpc, objfpc, tp, delphi, macpas). Note that a `{$Mode }` directive switch will reset the mode switches, so the `{$modeswitch }` statement should be located after it.

11.2 Objective-Pascal class declarations

Objective-C or -Pascal classes are declared much as Object Pascal classes are declared, but they use the `objcclass` keyword:

Objective C Class types



As can be seen, the syntax is roughly equivalent to Object Pascal syntax, with some extensions.

In order to use Objective-C classes, an external modifier exists: this indicates to the compiler that the class is implemented in an external object file or library, and that the definition is meant for import purposes. The following is an example of an external Objective-C class definition:

```
NSView = objcclass external(NSResponder)
private
  _subview  : id;
public
  function initWithFrame(rect : NSRect): id;
     message 'initWithFrame:';
```

```
  procedure addSubview(aview: NSView);
     message 'addSubview:';
  procedure setAutoresizingMask(mask: NSUInteger);
     message 'setAutoresizingMask:';
  procedure setAutoresizesSubviews(flag: LongBool);
     message 'setAutoresizesSubviews:';
  procedure drawRect(dirtyRect: NSRect);
     message 'drawRect:';
end;
```

As can be seen, the class definition is not so different from an Object Pascal class definition; Only the message directive is more prominently present: each Objective-C or Objective-Pascal method must have a message name associated with it. In the above example, no external name was specified for the class definition, meaning that the Pascal identifier is used as the name for the Objective-C class. However, since Objective-C is not so strict in its naming conventions, sometimes an alias must be created for an Objective-C class name that doesn't obey the Pascal identifier rules.

The following example defines an Objective-C class which is implemented in Pascal:

```
MyView = objcclass(NSView)
public
  data : Integer;
  procedure customMessage(dirtyRect: NSRect);
    message 'customMessage';
  procedure drawRect(dirtyRect: NSRect); override;
end;
```

The absence of the `external` keyword tells the compiler that the methods must be implemented later in the source file: it will be treated much like a regular object pascal class. Note the presence of the `override` directive: in Objective-C, all methods are virtual. In Object Pascal, overriding a virtual method must be done through the `override` directive. This has been extended to `Objective-C` classes: it allows the compiler to verify the correctness of the definition.

Unless the class is implementing the method of a protocol (more about this in a subsequent section), one of `message` or `override` is expected: all methods are virtual, and either a new method is started (or re-introduced), or an existing is overridden. Only in the case of a method that is part of a protocol, the method can be defined without `message` or `override`.

Note that the Objective-C class declaration may or may not specify a parent class. In Object Pascal, omitting a parent class will automatically make the new class a descendant of `TObject`. In Objective-C, this is not the case: the new class will be a new root class. However, Objective-C does have a class which fulfills the function of generic root class: `NSObject`, which can be considered the equivalent of `TObject` in Object Pascal. It has other root classes, but in general, Objective-Pascal classes should descend from `NSObject`. If a new root class is constructed anyway, it must implement the `NSObjectProtocol` – just as the `NSObject` class itself does.

Finally, Objective-Pascal classes can have properties, but these properties are only usable in Pascal code: the compiler currently does not export the properties in a way that makes them usable from Objective-C.

11.3 Formal declaration

Object Pascal has the concept of Forward declarations. Objective-C takes this concept a bit further: it allows to declare a class which is defined in another unit. This has been dubbed "Formal declaration" in Objective-Pascal. Looking at the syntax diagram, the following is a valid declaration:

```
MyExternalClass = objcclass external;
```

This is a formal declaration. It tells the compiler that `MyExternalClass` is an Objective-C class type, but that there is no declaration of the class members. The type can be used in the remainder of the unit, but its use is restricted to storage allocation (in a field or method parameter definition) and assignment (much like a pointer).

As soon as the class definition is encountered, the compiler can enforce type compatibility.

The following unit uses a formal declaration:

```
unit ContainerClass;

{$mode objfpc}
{$modeswitch objectivec1}

interface

type
  MyItemClass = objcclass external;

  MyContainerClass = objcclass
    private
     item: MyItemClass;
    public
     function getItem: MyItemClass; message 'getItem';
  end;

implementation

function MyContainerClass.getItem: MyItemClass;
begin
  result:=item; // Assignment is OK.
end;

end.
```

A second unit can contain the actual class declaration:

```
unit ItemClass;

{$mode objfpc}
{$modeswitch objectivec1}

interface

type
  MyItemClass = objcclass(NSObject)
  private
    content : longint;
  public
    function initWithContent(c: longint): MyItemClass;
       message 'initWithContent:';
     function getContent: longint;
       message 'getContent';
```

```
  end;

implementation

function MyItemClass.initWithContent(c: longint):
   MyItemClass;
begin
  content:=c;
  result:=self;
end;

function MyItemClass.getContent: longint;
begin
  result:=content;
end;

end.
```

If both units are used in a program, the compiler knows what the class is and can verify the correctness of some assignments:

```
Program test;

{$mode objfpc}
{$modeswitch objectivec1}

uses
  ItemClass, ContainerClass;

var
  c: MyContainerClass;
  l: longint;
begin
  c:=MyContainerClass.alloc.init;
  l:=c.getItem.getContent;
end.
```

11.4 Allocating and de-allocating Instances

The syntax diagram of Objective-C classes shows that the notion of constructor and destructor is not supported in Objective-C. New instances are created in a 2-step process:

1. Call the "alloc" method (send an "alloc" message): This is a class method of `NSObject`, and returns a pointer to memory for the new instance. The use of `alloc` is a convention in Objective-C.
2. Send an "initXXX" message. By convention, all classes have one or more "InitXXX" methods that initializes all fields in the instance. This method will return the final instance pointer, which may be `Nil`.

The following code demonstrates this:

```
var
```

```
  obj: NSObject;
begin
  // First allocate the memory.
  obj:=NSObject.alloc;
  // Next, initialise.
  obj:=obj.init;
  // Always check the result !!
  if (Obj=Nil) then
    // Some error;
```

By convention, the `initXXX` method will return `Nil` if initialization of some fields failed, so it is imperative that the result of the function is tested.

Similarly, no privileged destructor exists; By convention, the `dealloc` method fulfills the cleanup of the instances. This method can be overridden to perform any cleanup necessary. Like `Destroy`, it should never be called directly, instead, the `release` method should be called instead: All instances in Objective-C are reference counted, and `release` will only call `dealloc` if the reference count reaches zero.

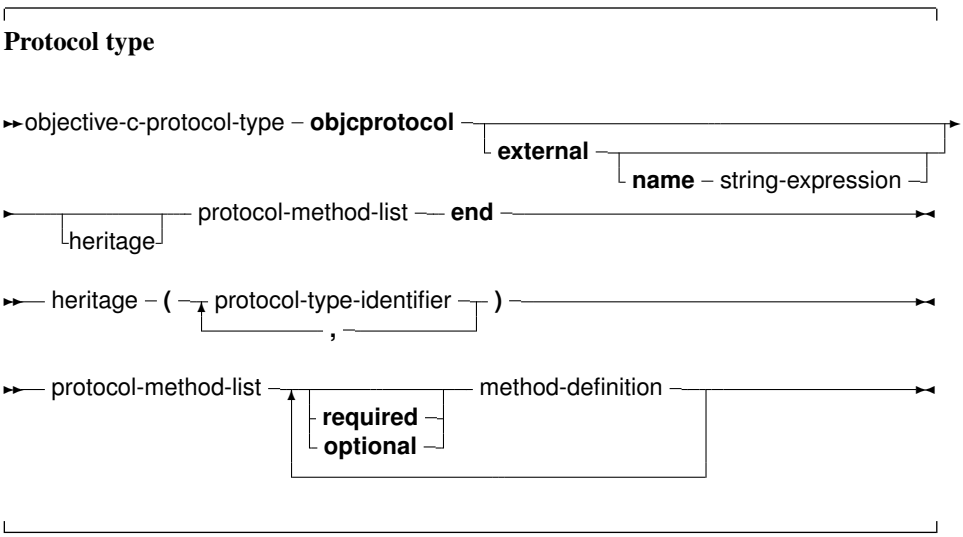
11.5 Protocol definitions

In Objective-C, protocols play the role that interfaces play in Object Pascal, but there are some differences:

- Protocol methods can be marked optional, i. e. the class implementing the protocol can decide not to implement these methods.
- Protocols can inherit from multiple other protocols.

Objective-C classes can indicate which protocols they implement in the class definition, as could be seen in the syntax diagram for Objective-C classes.

The following diagram shows how to declare a protocol. It starts with the `objcprotocol` keyword:



As in the case of Objective-Pascal classes, the `external` specifier tells the compiler that the declaration is an import of a protocol defined elsewhere. For methods, almost the same rules apply as for

methods in the Objective-Pascal class declarations. The exception is that message specifiers must be present.

The `required` and `optional` specifiers before a series of method declarations are optional. If none is specified, `required` is assumed. The following is a definition of a protocol:

```
type
  MyProtocol = objccprotocol
    // default is required
    procedure aRequiredMethod;
      message 'aRequiredMethod';
  optional
    procedure anOptionalMethodWithPara(para: longint);
      message 'anOptionalMethodWithPara:';
    procedure anotherOptionalMethod;
      message 'anotherOptionalMethod';
  required
    function aSecondRequiredMethod: longint;
      message 'aSecondRequiredMethod';
  end;

  MyClassImplementingProtocol = objcclass(NSObject,MyProtocol)
    procedure aRequiredMethod;
    procedure anOptionalMethodWithPara(para: longint);
    function aSecondRequiredMethod: longint;
  end;
```

Note that in the class declaration, the message specifier was omitted. The compiler (and runtime) can deduce it from the protocol definition.

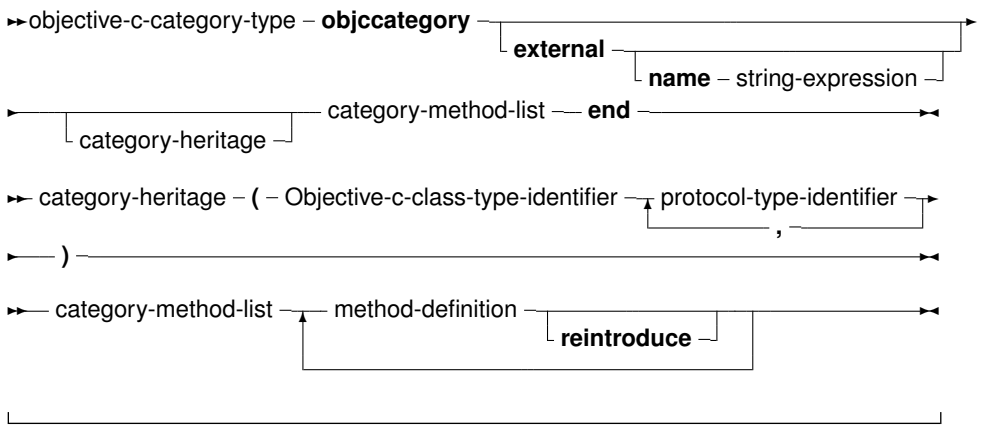
11.6 Categories

Similar to class helpers in Object Pascal, Objective-C has Categories. Categories allow to extend classes without actually creating a descendant of these classes. However, Objective-C categories provide more functionality than a class helper:

1. In Object Pascal, only one helper class can be in scope (the last one). In Objective-C, multiple categories can be in scope at the same time for a particular class.
2. In Object Pascal, a helper method cannot change an existing method present in the original class (but it can hide a method). In Objective-C, a category can also replace existing methods in another class rather than only add new ones. Since all methods are virtual in Objective-C, this also means that this method changes for all classes that inherit from the class in which the method was replaced (unless they override it).
3. Object Pascal helpers cannot be used to add interfaces to existing classes. By contrast, an Objective-C category can also implement protocols.

The definition of an objective-C class closely resembles a protocol definition, and is started with the `objccategory` keyword:

Category type



Note again the possibility of an alias for externally defined categories: Objective-C 2.0 allows an empty category name. Note that the `reintroduce` modifier must be used if an existing method is being replaced rather than that a new method is being added.

When replacing a method, calling "inherited" will not call the original method of the class, but instead will call the parent class' implementation of the method.

The following is an example of a category definition:

```
MyProtocol = objcprotocol
  procedure protocolmethod; message 'protocolmethod';
end;

MyCategory = objccategory(NSObject,MyProtocol)
  function hash: cuint; reintroduce;
  procedure protocolmethod; // from MyProtocol.
  class procedure newmethod; message 'newmethod';
end;
```

Note that this declaration replaces the `Hash` method of every class that descends from `NSObject` (unless it specifically overrides it).

11.7 Name scope and Identifiers

In Object Pascal, each identifier must be unique in its namespace: the unit. In Objective-C, this need not be the case and each type identifier must be unique among its kind: classes, protocols, categories, fields or methods. This is shown in the definitions of the basic protocol and class of Objective-C: Both protocol and class are called `NSObject`.

When importing Objective-C classes and protocols, the Objective-Pascal names of these types must conform to the Object Pascal rules, and therefore must have distinct names. Likewise, names that are valid identifiers in Objective-C may be reserved words in Object Pascal. They also must be renamed when imported.

To make this possible, the `External` and "message" modifiers allow to specify a name: this is the name of the type or method as it exists in Objective-C:

```
NSObjectProtocol = objcprotocol external name 'NSObject'
  function _class: pobjc_class; message name 'class';
end;
```

```
NSObject = objcclass external (NSObjectProtocol)
  function _class: pobjc_class;
  class function classClass: pobjc_class; message 'class';
end;
```

11.8 Selectors

A Selector in Objective-C can be seen as an equivalent to a procedural type in Object Pascal.

In difference with the procedural type, Objective-C has only one selector type: `SEL`. It is defined in the `objc` unit – which is automatically included in the uses clause of any unit compiled with the `objectivec1` modeswitch.

To assign a value to a variable of type `SEL`, the `objcselector` method must be used:

```
{$modeswitch objectivec1}
var
  a: SEL;
begin
  a:=objcselector('initiWithWidth:andHeight:');
  a:=objcselector('myMethod');
end.
```

The `objc` unit contains methods to manipulate and use the selector.

11.9 The `id` type

The `id` type is special in Objective-C/Pascal. It is much like the pointer type in Object Pascal, except that it is a real class. It is assignment-compatible with instances of every `objcclass` and `objcprotocol` type, in two directions:

1. variables of any `objcclass`/`objcprotocol` type can be assigned to a variable of the type `id`.
2. variables of type `id` can be assigned to variables of any particular `objcclass`/`objcprotocol` type.

No explicit typecast is required for either of these assignments.

Additionally, any Objective-C method declared in an `objcclass` or `objccategory` that is in scope can be called when using an `id`-typed variable.

If, at run time, the actual `objcclass` instance stored in the `id`-typed variable does not respond to the sent message, the program will terminate with a run time error: much like the dispatch mechanism for variants under MS-Windows.

When there are multiple methods with the same Pascal identifier, the compiler will use the standard overload resolution logic to pick the most appropriate method. In this process, it will behave as if all `objcclass`/`objccategory` methods in scope have been declared as global procedures/functions with the `overload` specifier. Likewise, the compiler will print an error if it cannot determine which overloaded method to call.

In such cases, a list of all methods that could be used to implement the call will be printed as a hint.

To resolve the error, an explicit type cast must be used to tell the compiler which objcclass type contains the needed method.

11.10 Enumeration in Objective-C classes

Fast enumeration in Objective-C is a construct which allows to enumerate the elements in a Cocoa container class in a generic way. It is implemented using a `for-in` loop in Objective-C.

This has been translated to Objective-Pascal using the existing `for-in` loop mechanism. Therefore, the feature behaves identically in both languages. Note that it requires the Objective-C 2.0 mode switch to be activated.

The following is an example of the use of for-in:

```
{$mode delphi}
{$modeswitch objectivec2}

uses
  CocoaAll;

var
  arr: NSMutableArray;
  element: NSString;
  pool: NSAutoreleasePool;
  i: longint;
begin
  pool:=NSAutoreleasePool.alloc.init;
  arr:=NSMutableArray.arrayWithObjects(
    NSSTR('One'),
    NSSTR('Two'),
    NSSTR('Three'),
    NSSTR('Four'),
    NSSTR('Five'),
    NSSTR('Six'),
    NSSTR('Seven'),
    nil);

  i:=0;
  for element in arr do
    begin
      inc(i);
      if i=2 then
        continue;
      if i=5 then
        break;
      if i in [2,5..10] then
        halt(1);
      NSLog(NSSTR('element: %@'),element);
    end;
  pool.release;
end.
```

Chapter 12

Expressions

Expressions occur in assignment statements or in tests.

Expressions are built of two components: operators and their operands. Most operators are binary, i. e. require two operands. Binary operators occur always between the operands (as in `X/Y`). Few operators are unary, i. e. require only one operand. A unary operator occurs always before the operand, as in `-X`.

An expression resolves into a value of a certain type. The resulting type is determined by the types of the values in the expression and the operators in the expression.

When using multiple operands in an expression, the precedence rules of table (12.1) are used.

Table 12.1: Precedence of operators

Operator	Precedence	Category
`Not`, unary `+`, unary `-`, `@`, `**`	Highest (first)	Unary operators, power
`*`, `/`, `div`, `mod`, `and`, `shl`, `shr`, `as`, `<<`, `>>`	Second	Multiplying operators
`+`, `-`, `or`, `xor`, `><`	Third	Adding operators
`=`, `<>`, `<`, `>`, `<=`, `>=`, `in`, `is`	Lowest (Last)	relational operators

When determining the precedence, the compiler uses the following rules:

1. In operations with unequal precedences the operands belong to the operator with the highest precedence. For example, in `5*3+7`, the multiplication is higher in precedence than the addition, so it is ievaluated first. The result would be 22.
2. If parentheses are used in an expression, their contents is evaluated first. Thus, `5*(3+7)` would result in 50.
3. Otherwise, binary operators of the same precedence are left-associative. `5 * 3 div 7` will evaluate to 2 and not 0.

Remark The order in which expressions of the same precedence are evaluated is not guaranteed to be left-to-right. In general, no assumptions on which subexpression is evaluated first should be made in such a case.

The compiler will decide which subexpression to evaluate first based on optimization rules. Thus, in the following expression:

```
a := g(3) + f(2);
```

`f(2)` may be executed before `g(3)`. This behavior is distinctly different from Delphi or Turbo Pascal.

If one expression *must* be executed before the other, it is necessary to split up the statement using temporary results:

```
e1 := g(3);
a  := e1 + f(2);
```

A notable exception to this behavior is the evaluation of boolean expressions: if short-circuit boolean evaluation is enabled (the default) then the compiler will evaluate from left to right, but will still respect precedence, i. e. in parts with equal precedence the left operand will always be evaluated before the right one. Thus, the following example:

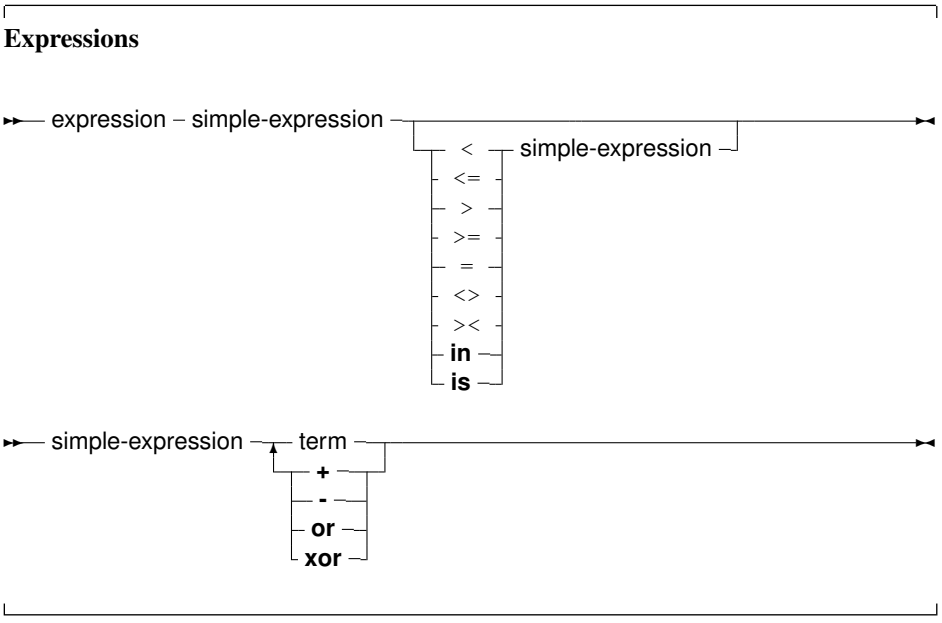
```
True or True and False
```

will evaluate to `True`, because it is equivalent to

```
True or (True and False)
```

12.1 Expression syntax

An expression applies relational operators to simple expressions. Simple expressions are a series of terms (what a term is, is explained below), joined by adding operators.



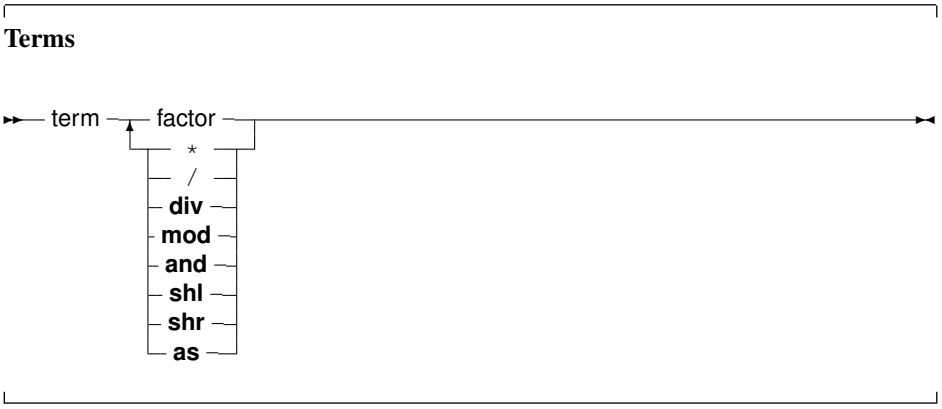
The following are valid expressions:

```
GraphResult<>grError
(DoItToday=Yes) and (DoItTomorrow=No);
Day in Weekend
```

And here are some simple expressions:

```
A + B
-Pi
ToBe or NotToBe
```

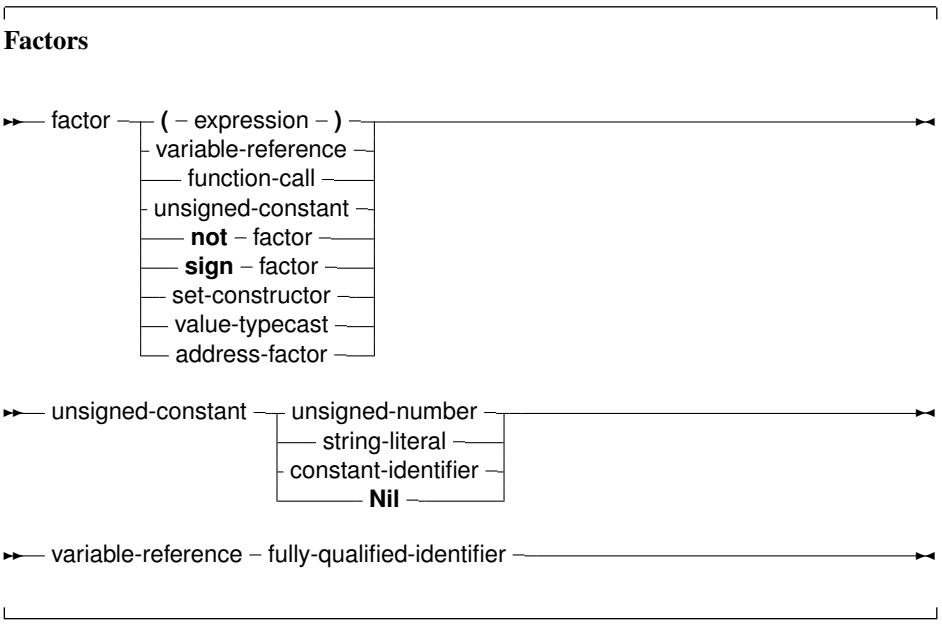
Terms consist of factors, connected by multiplication operators.



Here are some valid terms:

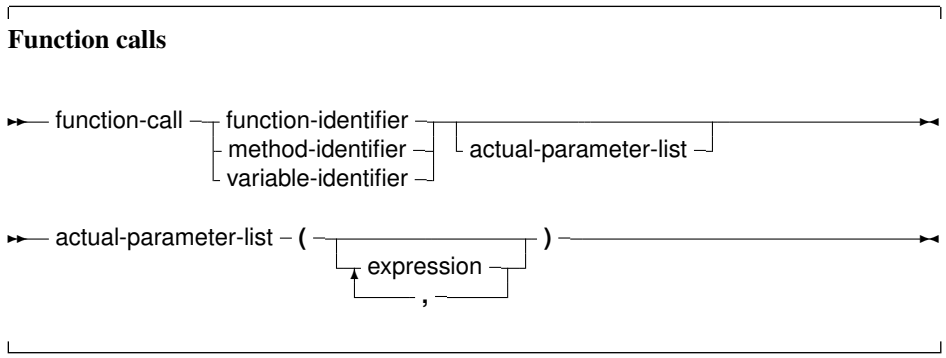
```
2 * Pi
A Div B
(DoItToday=Yes) and (DoItTomorrow=No);
```

Factors are all other constructions:



12.2 Function calls

Function calls are part of expressions (although, using extended syntax, they can be statements too). They are constructed as follows:



The `variable-reference` must be a procedural type variable reference. A method designator can only be used inside the method of an object. A qualified method designator can be used outside object methods too. The function that will get called is the function with a declared parameter list that matches the actual parameter list. This means that

1. The number of actual parameters must equal the number of declared parameters (unless default parameter values are used).
2. The types of the parameters must be compatible. For variable reference parameters, the parameter types must be exactly the same.

If no matching function is found, then the compiler will generate an error. Which error depends – among other things – on whether the function is overloaded or not: i. e. multiple functions with the same name, but different parameter lists.

There are cases when the compiler will not execute the function call in an expression. This is the case when assigning a value to a procedural type variable, as in the following example in Delphi or Turbo Pascal mode:

```
Type
  FuncType = Function: Integer;
Var A : Integer;
Function AddOne : Integer;
begin
  A := A+1;
  AddOne := A;
end;
Var F : FuncType;
    N : Integer;
begin
  A := 0;
  F := AddOne; { Assign AddOne to F, Don't call AddOne}
  N := AddOne; { N := 1 !!}
end.
```

In the above listing, the assignment to `F` will not cause the function `AddOne` to be called. The assignment to `N`, however, will call `AddOne`.

Sometimes, the call is desired, for instance in recursion in that case, the call must be forced. This can be done by adding the parenthesis to the function name:

```
function rd : char;
```

```
var
  c : char;

begin
  read(c);
  if (c='\') then
    c:=rd();
  rd:=c;
end;

var ch : char;

begin
   ch:=rd;
   writeln(ch);
end.
```

The above will read a character and print it. If the input is a backslash, a second character is read.

A problem with this syntax is the following construction:

```
If F = AddOne Then
  DoSomethingHorrible;
```

Should the compiler compare the addresses of `F` and `AddOne`, or should it call both functions, and compare the result? In `fpc` and `objfpc` mode this is solved by considering a procedural variable as equivalent to a pointer. Thus the compiler will give a type mismatch error, since `AddOne` is considered a call to a function with integer result, and `F` is a pointer.

How then, should one check whether `F` points to the function `AddOne`? To do this, one should use the address operator `@`:

```
If F = @AddOne Then
  WriteLn ('Functions are equal');
```

The left hand side of the boolean expression is an address. The right hand side also, and so the compiler compares two addresses. How to compare the values that both functions return ? By adding an empty parameter list:

```
  If F()=Addone then
    WriteLn ('Functions return same values ');
```

Remark that this last behavior is not compatible with Delphi syntax. Switching on `Delphi` mode will allow you to use Delphi syntax.

If a function returns a procedural type (a pointer to a function or procedure), then it can be called immediately (without assigning a pointer to a variable) by specifying additional parentheses for its arguments:

```
  i := GetProc(1, 2)('test');
  // which is equivalent to
  p := GetProc(1, 2);
  i := p('test');
```

If the called functions have no arguments, empty parentheses must be specified so that the compiler can distinguish between a function call and a call to the function's result:

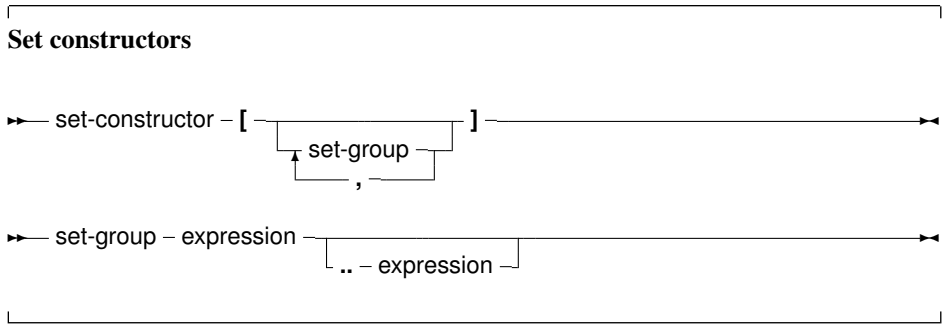
```
  GetProc()();
  // which is equivalent to
  p := GetProc();
  p();
```

The difference is demonstrated in the following example. In `Delphi` mode, all calls will assign the value `123` to the variable `i`. In `fpc` and `objfpc` mode, the example will fail to compile due to the first two lines containing the calls:

```
program Project1;
//{$mode delphi}
type TFunc = function: integer;
  function GetInt: integer;
  begin
    result := 123;
  end;
  function GetProc: TFunc;
  begin
    result := @GetInt;
  end;
var i: integer;
begin
  i := GetProc;     // ERROR - GetProc returns TFunc (optional parentheses omitted)
  i := GetProc();   // ERROR - GetProc returns TFunc (optional parentheses included)
  i := GetProc()(); // OK
end.
```

12.3 Set constructors

When a set-type constant must be entered in an expression, a set constructor must be given. In essence this is the same thing as when a type is defined, only there is no identifier to identify the set with. A set constructor is a comma separated list of expressions, enclosed in square brackets.



All set groups and set elements must be of the same ordinal type. The empty set is denoted by `[]`, and it can be assigned to any type of set. A set group with a range `[A..Z]` makes all values in the range a set element. The following are valid set constructors:

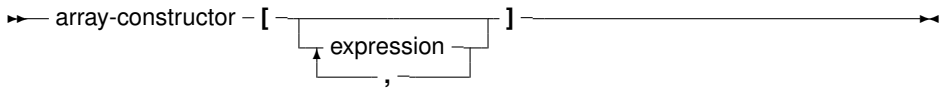
```
[today,tomorrow]
[Monday..Friday,Sunday]
[ 2, 3*2, 6*2, 9*2 ]
['A'..'Z','a'..'z','0'..'9']
```

Remark If the first range specifier has a bigger ordinal value than the second, the resulting set will be empty, e. g., `['Z'..'A']` denotes an empty set. One should be careful when denoting a range.

12.4 Array constructors

When a dynamic array constant must be entered in an expression, an array constructor can be given. An array constructor is a comma separated list of expressions enclosed in square brackets. The result type of each expression in the list must be compatible to the array element type.

Set constructors



The empty array is denoted by `[]`, and it can be assigned to any type of dynamic array.

The following are valid array constructors:

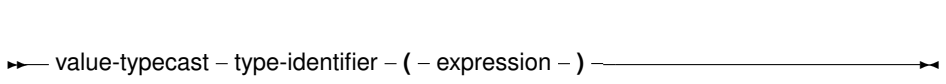
```
['yesterday,'today','tomorrow']
[ 1, 2, 4, 9, 10]
[ 2, 3*2, 6*2, 9*2 ]
```

The second and third expression are potentially sets. The compiler must be able to determine whether the result needs to be a set or an array.

12.5 Value typecasts

Sometimes it is necessary to change the type of an expression, or a part of the expression, to be able to be assignment compatible. This is done through a value typecast. The syntax diagram for a value typecast is as follows:

Typecasts



Value typecasts cannot be used on the left side of assignments, as variable typecasts. Here are some valid typecasts:

```
Byte('A')
Char(48)
boolean(1)
longint(@Buffer)
```

In general, the type size of the expression and the size of the type cast must be the same. However, for ordinal types (byte, char, word, boolean, enumerates) this is not so, they can be used interchangeably. That is, the following will work, although the sizes do not match.

```
Integer('A');
Char(4875);
boolean(100);
Word(@Buffer);
```

This is compatible with Delphi or Turbo Pascal behavior.

Remark Note that if you typecast a value (or variable), the compiler will assume that the result of the typecast is a valid value for the type you typecasted it to. In particular, this means no range checks will be inserted, even when range checks are enabled.

12.6 Variable typecasts

A variable can be considered a single factor in an expression. It can therefore be typecast as well. A variable can be typecast to any type, provided the type has the same size as the original variable.

It is a bad idea to typecast integer types to real types and vice versa. It's better to rely on type assignment compatibility and using some of the standard type changing functions.

Note that variable typecasts can occur on either side of an assignment, i. e. the following are both valid typecasts:

```
Var
  C : Char;
  B : Byte;

begin
  B:=Byte(C);
  Char(B):=C;
end;
```

But not all typecasts can be assigned to. The above will work, but e.g. a typecast to single cannot be assigned to.

Pointer variables can be typecast to procedural types, but not to method pointers.

A typecast is an expression of the given type, which means the typecast can be followed by a qualifier:

```
Type
  TWordRec = Packed Record
    L,H : Byte;
  end;

Var
  P : Pointer;
  W : Word;
  S : String;

begin
  TWordRec(W).L:=$FF;
  TWordRec(W).H:=0;
  S:=TObject(P).ClassName;
```

12.7 Unaligned typecasts

A special typecast is the `Unaligned` typecast of a variable or expression. This is not a real typecast, but is rather a hint for the compiler that the expression may be misaligned (i. e. not on an aligned memory address). Some processors do not allow direct access to misaligned data structures, and therefore must access the data byte per byte.

Typecasting an expression with the unaligned keyword signals the compiler that it should access the data byte per byte.

Note that the compiler assumes that access to all fields/elements of packed data structures is unaligned.

Example:

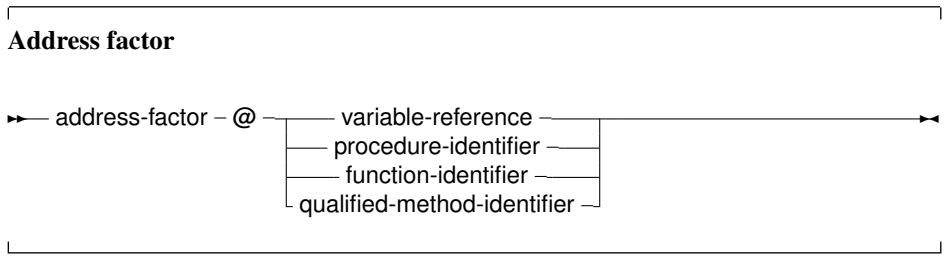
```
program me;

Var
  A : packed Array[1..20] of Byte;
  I : LongInt;

begin
  For I:=1 to 20 do
    A[I]:=I;
  I:=Unaligned(PInteger(@A[13])^);
end.
```

12.8 The @ operator

The address operator `@` returns the address of a variable, procedure or function. It is used as follows:



The `@` operator returns a typed pointer if the `$T` switch is on. If the `$T` switch is off then the address operator returns an untyped pointer, which is assignment compatible with all pointer types. The type of the pointer is `^T`, where `T` is the type of the variable reference. For example, the following will compile

```
Program tcast;
{$T-} { @ returns untyped pointer }

Type art = Array[1..100] of byte;
Var Buffer : longint;
    PLargeBuffer : ^art;

begin
 PLargeBuffer := @Buffer;
end.
```

Changing the `{$T-}` to `{$T+}` will prevent the compiler from compiling this. It will give a type mismatch error.

By default, the address operator returns an untyped pointer: applying the address operator to a function, method, or procedure identifier will give a pointer to the entry point of that function. The result is an untyped pointer.

This means that the following will work:

```
Procedure MyProc;

begin
end;

Var
  P : PChar;

begin
  P:=@MyProc;
end;
```

By default, the address operator must be used if a value must be assigned to a procedural type variable. This behavior can be avoided by using the `-Mtp` or `-MDelphi` switches, which result in a more compatible Delphi or Turbo Pascal syntax.

12.9 Operators

Operators can be classified according to the type of expression they operate on. We will discuss them type by type.

12.9.1 Arithmetic operators

Arithmetic operators occur in arithmetic operations, i. e. in expressions that contain integers or reals. There are two kinds of operators: Binary and unary arithmetic operators. Binary operators are listed in table (12.2), unary operators are listed in table (12.3).

Table 12.2: Binary arithmetic operators

Operator	Operation
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`**`	Exponentiation
`/`	Division
`Div`	Integer division
`Mod`	Remainder

With the exception of `Div` and `Mod`, which accept only integer expressions as operands, all operators accept real and integer expressions as operands.

Remark The exponentiation operator (`**`) is available for overloading (chapter 15, page 246), but is not defined on any of the standard Pascal types (floats and/or integers).

For binary operators, the result type will be integer if both operands are integer type expressions. If one of the operands is a real type expression, then the result is real.

As an exception, division (`/`) results always in real values.

Table 12.3: Unary arithmetic operators

Operator	Operation
+	Sign identity
−	Sign inversion

For unary operators, the result type is always equal to the expression type. The division (`/`) and `Mod` operator will cause run-time errors if the second argument is zero.

The sign of the result of a `Mod` operator is the same as the sign of the left side operand of the `Mod` operator. In fact, the `Mod` operator is equivalent to the following operation:

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

But it executes faster than the right hand side expression.

12.9.2 Logical operators

Logical operators act on the individual bits of ordinal expressions. Logical operators require operands that are of an integer type, and produce an integer type result. The possible logical operators are listed in table (12.4).

Table 12.4: Logical operators

Operator	Operation
`not`	Bitwise negation (unary)
`and`	Bitwise and
`or`	Bitwise or
`xor`	Bitwise xor
`shl`	Bitwise shift to the left
`shr`	Bitwise shift to the right
`<<`	Bitwise shift to the left (same as shl)
`>>`	Bitwise shift to the right (same as shr)

The following are valid logical expressions:

```
A shr 1   { same as A div 2, but faster}
Not 1     { equals -2 }
Not 0     { equals -1 }
Not -1    { equals 0  }
B shl 2   { same as B * 4 for integers }
1 or 2    { equals 3 }
3 xor 1   { equals 2 }
```

When performing logical operations, it should be noted that:

- No overflow checking is done on logical operations. Range errors may of course occur when the result of a logical operation is assigned to a variable.
- Bit shift operations that shift bits with more positions than the `(bit size of operand)-1` are undefined. The result of a negative shift is also undefined.

12.9.3 Boolean operators

Boolean operators can be considered as logical operations on a type with 1 bit size. Therefore the `shl` and `shr` operations have little sense. Boolean operators can only have boolean type operands, and the resulting type is always boolean. The possible operators are listed in table (12.5)

Table 12.5: Boolean operators

Operator	Operation
`not`	logical negation (unary)
`and`	logical and
`or`	logical or
`xor`	logical xor

Remark By default, boolean expressions are evaluated with short-circuit evaluation. This means that from the moment the result of the complete expression is known, evaluation is stopped and the result is returned. For instance, in the following expression:

```
B := True or MaybeTrue;
```

The compiler will never look at the value of `MaybeTrue`, since it is obvious that the expression will always be `True`. As a result of this strategy, if `MaybeTrue` is a function, it will not get called ! (This can have surprising effects when used in conjunction with properties)

12.9.4 String operators

There is only one string operator: `+`. Its action is to concatenate the contents of the two strings (or characters) it acts on. One cannot use `+` to concatenate null-terminated (`PChar`) strings. The following are valid string operations:

```
'This is ' + 'VERY ' + 'easy !'
Dirname+'\'
```

The following is not:

```
Var
  Dirname : PChar;
...
  Dirname := Dirname+'\';
```

Because `Dirname` is a null-terminated string.

Note that if all strings in a string expressions are short strings, the resulting string is also a short string. Thus, a truncation may occur: there is no automatic upscaling to ansistring.

If all strings in a string expression are ansistrings, then the result is an ansistring.

If the expression contains a mix of ansistrings and shortstrings, the result is an ansistring.

The value of the `{$H}` switch can be used to control the type of constant strings; by default, they are short strings (and thus limited to 255 characters).

12.9.5 Dynamic array operators

There is only one dynamic array operator: +. This operator is available in Delphi mode, but must be enabled explicily using the modeswitch `arrayoperators` in objfpc mode:

```
{$mode objfpc}
{$modeswitch arrayoperators}
```

When enabled, its action is similar to the concatenation for strings: to concatenate the contents of the two arrays it acts on. The array element type must of course be identical for both arrays, i.e. the following will work:

```
{$mode objfpc}
{$modeswitch arrayoperators}
var
  a,b, c : array of byte;

begin
  a:=[0,1,2];
  b:=[3,4,5];
  c:=a+b;
  writeln('C has ',length(c),' elements');
```

But the following will not work:

```
{$mode objfpc}
{$modeswitch arrayoperators}

var
  b, c : array of byte;
  a : array of integer;

begin
  a:=[0,1,2];
  b:=[3,4,5];
  c:=a+b;
  writeln('C has ',length(c),' elements');
```

The compiler will give an error when compiling this code.

12.9.6 Set operators

The following operations on sets can be performed with operators: union, difference, symmetric difference, inclusion and intersection. The operators are listed in table (12.6).

Table 12.6: Set operators

Operation	Action
+	Union
-	Difference operator
*	Intersection
><	Symmetric difference
<=	Contains
>=	Left hand side set is a superset of the one on the right
in	check whether an element is in a set

In addition to the above operators, elements can be added or removed from the set with the `Include` or `Exclude` procedures:

include includes an element in the set.

exclude excludes an element from the set.

The set type of the operands or arguments must be the same, or an error will be generated by the compiler.

The following program gives some valid examples of set operations:

```
Type
  Day = (mon,tue,wed,thu,fri,sat,sun);
  Days = set of Day;

Procedure PrintDays(W : Days);
Const
  DayNames : array [Day] of String[3]
           = ('mon','tue','wed','thu',
              'fri','sat','sun');
Var
  D : Day;
  S : String;
begin
  S:='';
  For D:=Mon to Sun do
    if D in W then
      begin
      If (S<>'') then S:=S+',';
      S:=S+DayNames[D];
      end;
  Writeln('[',S,']');
end;

Const
  WorkWeek = [mon,tue,wed,thu,fri];
  WeekEnd = [sat,sun];

Var
  W : Days;
```

```
begin
   W:=[mon,tue]+[wed,thu,fri]; // equals [mon,tue,wed,thu,fri]
   PrintDays(W);
   W:=[mon,tue,wed]-[wed];     // equals [mon,tue]
   PrintDays(W);
   W:=[mon,tue,wed]-[wed,thu];     // also equals [mon,tue]
   PrintDays(W);
   W:=[mon,tue,wed]*[wed,thu,fri]; // equals [wed]
   PrintDays(W);
   W:=[mon,tue,wed]><[wed,thu,fri]; // equals [mon,tue,thu,fri]
   PrintDays(W);
   if [mon,tue]<=WorkWeek then
     Writeln('Must work on monday and tuesday');
   if Weekend>=[sun] then
     Writeln('Can rest on sunday');
end.
```

As can be seen, the union is equivalent to a binary OR, while the intersection is equivalent to a binary AND, and the symmetric difference equals a XOR operation.

The `Include` and `Exclude` operations are equivalent to a union or a difference with a set of one element. Thus,

```
Include(W,wed);
```

is equivalent to

```
W:=W+[wed];
```

and

```
Exclude(W,wed);
```

is equivalent to

```
W:=W-[wed];
```

The `In` operation results in a `True` if the left operand (an element) is included of the right operand (a set), the result will be `False` otherwise.

12.9.7 Relational operators

The relational operators are listed in table (12.7)

Table 12.7: Relational operators

Operator	Action
=	Equal
<>	Not equal
<	Strictly less than
>	Strictly greater than
<=	Less than or equal
>=	Greater than or equal
in	Element of

Normally, left and right operands must be of the same type. There are some notable exceptions, where the compiler can handle mixed expressions:

1. Integer and real types can be mixed in relational expressions.
2. If the operator is overloaded, and an overloaded version exists whose arguments types match the types in the expression.
3. Short-, Ansi- and widestring types can be mixed.

Comparing strings is done on the basis of their character code representation.

When comparing pointers, the addresses to which they point are compared. This also is true for `PChar` type pointers. To compare the strings the `PChar` point to, the `StrComp` function from the strings unit must be used. The `in` returns `True` if the left operand (which must have the same ordinal type as the set type, and which must be in the range 0..255) is an element of the set which is the right operand, otherwise it returns `False`.

12.9.8 Class operators

Class operators are slightly different from the operators above in the sense that they can only be used in class expressions which return a class. There are only two class operators, as can be seen in table (12.8).

Table 12.8: Class operators

Operator	Action
`is`	Checks class type
`as`	Conditional typecast

An expression containing the `is` operator results in a boolean type. The `is` operator can only be used with a class reference or a class instance. The usage of this operator is as follows:

```
Object is Class
```

This expression is completely equivalent to

```
Object.InheritsFrom(Class)
```

If `Object` is `Nil`, `False` will be returned.

The following are examples:

```
Var
  A : TObject;
  B : TClass;

begin
  if A is TComponent then ;
  If A is B then;
end;
```

The `as` operator performs a conditional typecast. It results in an expression that has the type of the class:

```
  Object as Class
```

This is equivalent to the following statements:

```
  If Object=Nil then
    Result:=Nil
  else if Object is Class then
    Result:=Class(Object)
  else
    Raise Exception.Create(SErrInvalidTypeCast);
```

Note that if the object is `nil`, the `as` operator does not generate an exception.

The following are some examples of the use of the `as` operator:

```
Var
  C : TComponent;
  O : TObject;

begin
  (C as TEdit).Text:='Some text';
  C:=O as TComponent;
end;
```

The `as` and `is` operators also work on interfaces (both COM and CORBA). They can be used to check whether an interface also implements another interface as in the following example:

```
{$mode objfpc}

uses
  SysUtils;

type
  IMyInterface1 = interface
    ['{DD70E7BB-51E8-45C3-8CE8-5F5188E19255}']
    procedure Bar;
  end;

  IMyInterface2 = interface
    ['{7E5B86C4-4BC5-40E6-A0DF-D27DBF77BCA0}']
    procedure Foo;
  end;

  TMyObject = class(TInterfacedObject, IMyInterface1, IMyInterface2)
    procedure Bar;
    procedure Foo;
  end;

procedure TMyObject.Bar;
begin

end;

procedure TMyObject.Foo;
begin
```

```
end;

var
  i: IMyInterface1;
begin
  i := TMyObject.Create;
  i.Bar;
  Writeln(BoolToStr(i is IMyInterface2, True)); // prints true
  Writeln(BoolToStr(i is IDispatch, True)); // prints false
  (i as IMyInterface2).Foo;
end.
```

Additionally, the `is` operator can be used to check if a class implements an interface, and the `as` operator can be used to typecast an interface back to the class:

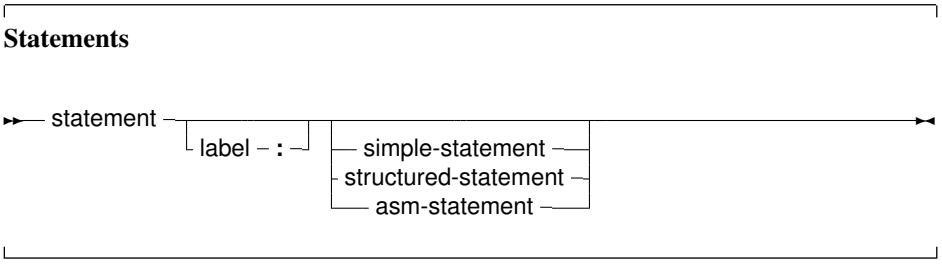
```
{$mode objfpc}
var
  i: IMyInterface;
begin
  i := TMyObject.Create;
  Writeln(BoolToStr(i is TMyObject,True)); // prints true
  Writeln(BoolToStr(i is TObject,True)); // prints true
  Writeln(BoolToStr(i is TAggregatedObject,True)); // prints false
  (i as TMyObject).Foo;
end.
```

Although the interfaces must be COM interfaces, the typecast back to a class will only work if the interface comes from an Object Pascal class. It will not work on interfaces obtained from the system by COM.

Chapter 13

Statements

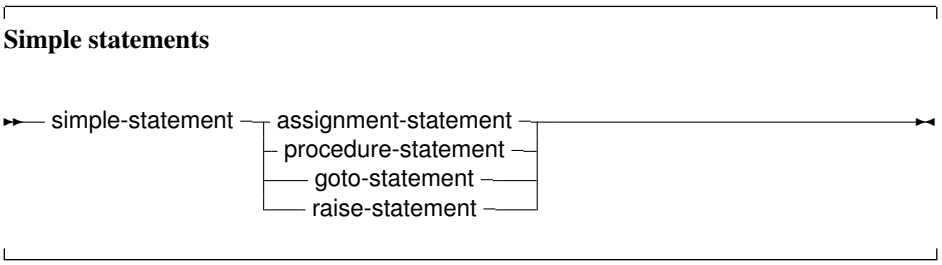
The heart of each algorithm are the actions it takes. These actions are contained in the statements of a program or unit. Each statement can be labeled and jumped to (within certain limits) with `Goto` statements. This can be seen in the following syntax diagram:



A label can be an identifier or an integer digit.

13.1 Simple statements

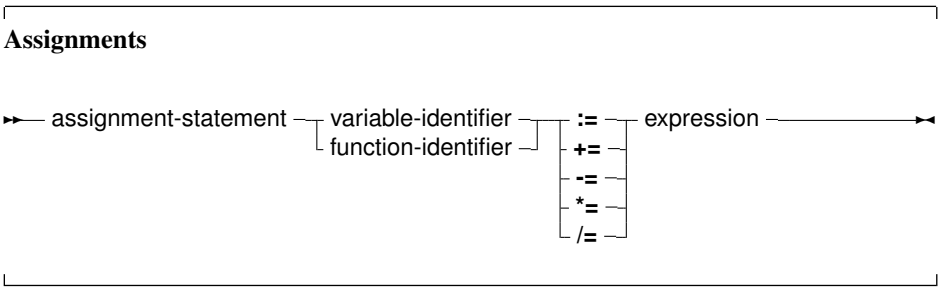
A simple statement cannot be decomposed in separate statements. There are basically four kinds of simple statements:



Of these statements, the *raise statement* will be explained in the chapter on Exceptions (chapter 17, page 272)

13.1.1 Assignments

Assignments give a value to a variable, replacing any previous value the variable might have had:



In addition to the standard Pascal assignment operator (:=), which simply replaces the value of the variable with the value resulting from the expression on the right of the := operator, Free Pascal supports some C-style constructions. All available constructs are listed in table (13.1).

Table 13.1: Allowed C constructs in Free Pascal

Assignment	Result
a += b	Adds b to a, and stores the result in a.
a -= b	Subtracts b from a, and stores the result in a.
a *= b	Multiplies a with b, and stores the result in a.
a /= b	Divides a through b, and stores the result in a (only for floats, not integers).

For C-like assignments, the $COPERATORS directive must be enabled or the -Sc command-line switch must be specified when compiling. These constructions are just for typing convenience and generate the same code.

Remark Here is an example with some valid assignment statements:

```
var
  Done: Boolean;
  S: String;
  X, Y: Double;
begin
  Done:=False or (1=2);
  S:='text';
  X:=X+Y;
  {$COPERATORS ON}
  X+=Y; // same as X:=X+Y
  X/=2; // same as X:=X/2
end.
```

Keeping in mind that the dereferencing of a typed pointer results in a variable of the type the pointer points to, the following are also valid assignments:

```
var
  L : ^Longint;
  P : PPChar;
begin
  L^:=3;
  P^^:='A';
end.
```

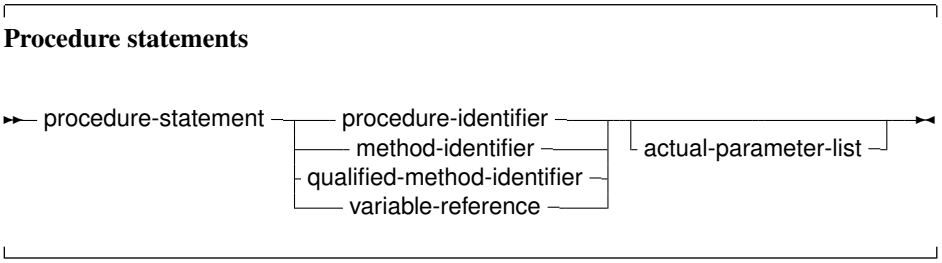
Note the double dereferencing in the second assignment.

13.1.2 Procedure statements

Procedure statements are calls to subroutines. There are different possibilities for procedure calls:

- A normal procedure call.
- An object method call (fully qualified or not).
- Or even a call to a procedural type variable.

All types are present in the following diagram:



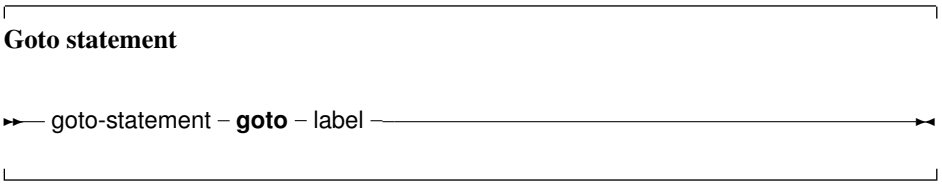
The Free Pascal compiler will look for a procedure with the same name as given in the procedure statement, and with a declared parameter list that matches the actual parameter list. The following are valid procedure statements:

```
Usage;
WriteLn('Pascal is an easy language !');
Doit();
```

Remark When looking for a function that matches the parameter list of the call, the parameter types should be assignment-compatible for value and const parameters, and should match exactly for parameters that are passed by reference.

13.1.3 Goto statements

Free Pascal supports the `goto` jump statement. Its prototype syntax is



When using `goto` statements, the following must be kept in mind:

1. The jump label must be defined in the same block as the `Goto` statement.
2. Jumping from outside a loop to the inside of a loop or vice versa can have strange effects.
3. To be able to use the `Goto` statement, the `-Sg` compiler switch must be used, or `{$GOTO ON}` must be used.

Remark In iso or macpas mode, or with the modeswitch "nonlocalgoto", the compiler will also allow non-local gotos.

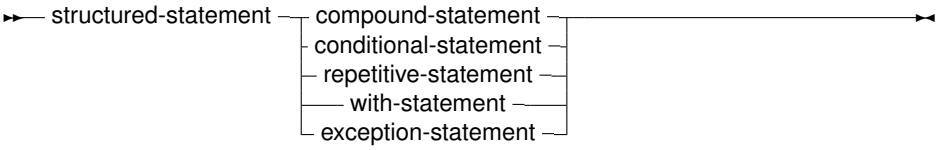
`Goto` statements are considered bad practice and should be avoided as much as possible. It is always possible to replace a `goto` statement by a construction that doesn't need a `goto`, although this construction may not be as clear as a goto statement. For instance, the following is an allowed goto statement:

```
label
  jumpto;
...
Jumpto :
  Statement;
...
Goto jumpto;
...
```

13.2 Structured statements

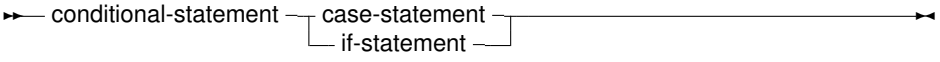
Structured statements can be broken into smaller simple statements, which should be executed repeatedly, conditionally or sequentially:

Structured statements



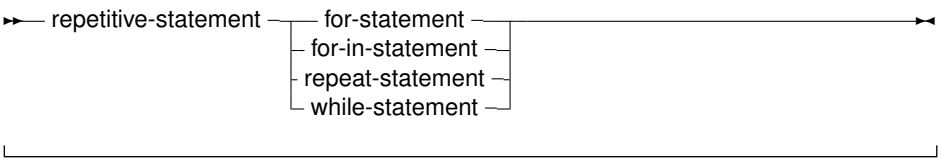
Conditional statements come in two flavors:

Conditional statements



Repetitive statements come in four flavors:

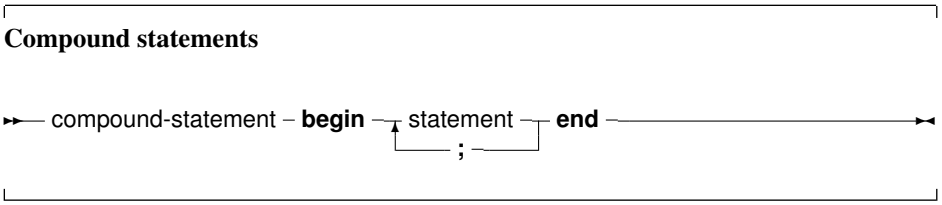
Repetitive statements



The following sections deal with each of these statements.

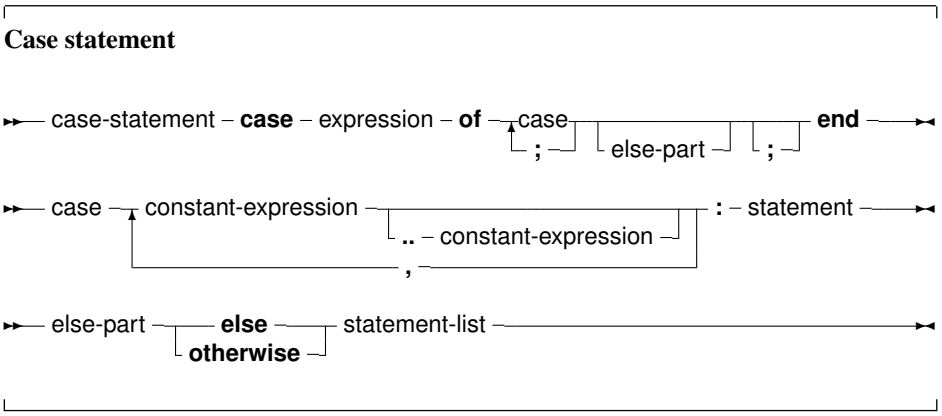
13.2.1 Compound statements

Compound statements are a group of statements, separated by semicolons, that are surrounded by the keywords `Begin` and `End`. The last statement – before the `End` keyword – doesn't need to be followed by a semicolon, although it is allowed. A compound statement is a way of grouping statements together, executing the statements sequentially. They are treated as one statement in cases where Pascal syntax expects one statement, such as in `if...then...else` statements.



13.2.2 The `Case` statement

Free Pascal supports the `case` statement. Its syntax diagram is



The constants appearing in the various case parts must be known at compile-time, and can be of the following types: enumeration types, Ordinal types (including chars) or string types. The case expression must be also of this type, or a compiler error will occur. All case constants must have the same type.

The compiler will evaluate the case expression. If one of the case constants' value matches the value of the expression, the statement that follows this constant is executed. After that, the program continues after the final `end`.

If none of the case constants match the expression value, the statement list after the `else` or `otherwise` keyword is executed. This can be an empty statement list. If no else part is present, and no case constant matches the expression value, program flow continues after the final `end`.

The case statements can be compound statements (i. e. a `Begin..End` block), but for the else case multiple statements are allowed (as can be seen in the syntax diagram). You may enclose them in an extra `Begin..End` block, but it is not necessary.

Note that the semicolon in front of the `else` clause is optional. This may lead to confusion if the last label contains an if statement.

In the following example, when `a` equals 3, nothing will be executed:

```
case a of
1 :
  DoSomething;
2 :
  If b>1 then
    DoSomething
else
  DoSomethingElse;
end;
```

But in the following example, when `a` equals 3, `DoSomethingElse` will be executed.

```
case a of
1 :
  DoSomething;
2 :
  If b>1 then
    DoSomething;
else
  DoSomethingElse;
end;
```

When in doubt or to avoid confusion, you can enclose the last statement in an extra `Begin..End` block.

Remark Contrary to Turbo Pascal, duplicate case labels are not allowed in Free Pascal, so the following code will generate an error when compiling:

```
Var i : integer;
...
Case i of
 3 : DoSomething;
 1..5 : DoSomethingElse;
end;
```

The compiler will generate a `Duplicate case label` error when compiling this, because the 3 also appears (implicitly) in the range `1..5`. This is similar to Delphi syntax.

The following are valid case statements:

```
Case C of
 'a' : WriteLn ('A pressed');
 'b' : WriteLn ('B pressed');
 'c' : WriteLn ('C pressed');
else
  WriteLn ('unknown letter pressed : ',C);
end;
```

Or

```
Case C of
 'a','e','i','o','u' : WriteLn ('vowel pressed');
 'y' : WriteLn ('This one depends on the language');
else
  WriteLn ('Consonant pressed');
end;
```

```
Case Number of
 1..10   : WriteLn ('Small number');
 11..100 : WriteLn ('Normal, medium number');
else
 WriteLn ('HUGE number');
end;
```

The `else` clause can contain multiple statements:

```
Case Number of
 1..10   : WriteLn ('Small number');
 11..100 : WriteLn ('Normal, medium number');
else
  WriteLn ('HUGE number');
  Writeln('How did we get this much ?');
end;
```

Free Pascal allows the use of strings as case labels, and in that case the case variable must also be a string. When using string types, the case variable and the various labels are compared in a case-sensitive way.

```
Case lowercase(OS) of
 'windows',
 'dos'   : WriteLn ('Microsoft platform);
 'macos',
 'darwin' : Writeln('Apple platform');
 'linux',
 'freebsd',
 'netbsd' : Writeln('Community platform');
else
  WriteLn ('Other platform');
end;
```

The case with strings is equivalent to a series of `if then else` statements, no optimizations are performed.

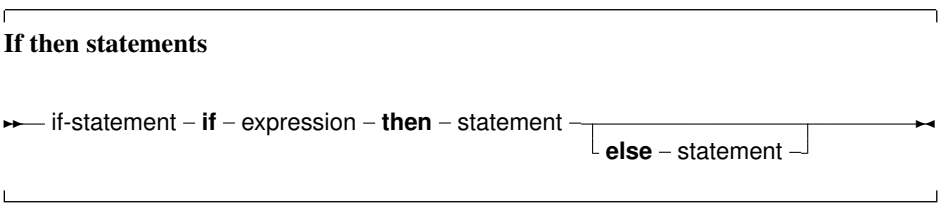
However, ranges are allowed, and are the equivalent of an

```
if (value>=beginrange) and (value<=endrange) then
  begin
  end;
```

Remark Note that in Delphi mode, the use of strings as case labels is not allowed.

13.2.3 The `If..then..else` statement

The `If .. then .. else..` prototype syntax is



The expression between the `if` and `then` keywords must have a `Boolean` result type. If the expression evaluates to `True` then the statement following the `then` keyword is executed.

If the expression evaluates to `False`, then the statement following the `else` keyword is executed, if it is present.

Some points to note:

- Be aware of the fact that the boolean expression by default will be short-cut evaluated, meaning that the evaluation will be stopped at the point where the outcome is known with certainty.
- Also, before the `else` keyword, no semicolon (`;`) is allowed, but all statements can be compound statements.
- In nested `If.. then .. else` constructs, some ambiguity may arise as to which `else` statement pairs with which `if` statement. The rule is that the `else` keyword matches the first `if` keyword (searching backwards) not already matched by an `else` keyword.

For example:

```
If exp1 Then
  If exp2 then
    Stat1
else
  stat2;
```

Despite its appearance, the statement is syntactically equivalent to

```
If exp1 Then
   begin
   If exp2 then
      Stat1
   else
      stat2
   end;
```

and not to

```
{ NOT EQUIVALENT }
If exp1 Then
   begin
   If exp2 then
      Stat1
   end
else
   stat2;
```

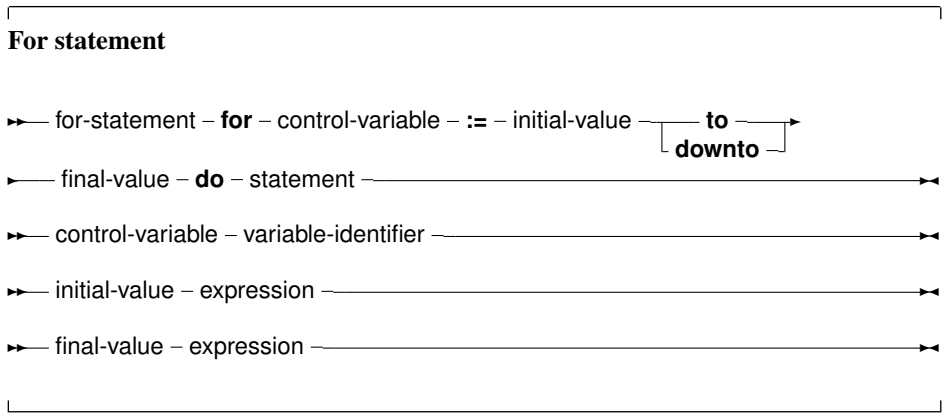
If it is this latter construct which is needed, the `begin` and `end` keywords must be present. When in doubt, it is better to add them.

The following is a valid statement:

```
If Today in [Monday..Friday] then
  WriteLn ('Must work harder')
else
  WriteLn ('Take a day off.');
```

13.2.4 The `For..to/downto..do` statement

Free Pascal supports the `For` loop construction. A `for` loop is used in case one wants to calculate something a fixed number of times. The prototype syntax is as follows:



Here, `Statement` can be a compound statement. When the `For` statement is encountered, the control variable is initialized with the initial value, and is compared with the final value. What happens next depends on whether `to` or `downto` is used:

1. In the case `To` is used, if the initial value is larger than the final value then `Statement` will never be executed.
2. In the case `DownTo` is used, if the initial value is less than the final value then `Statement` will never be executed.

After this check, the statement after `Do` is executed. After the execution of the statement, the control variable is increased or decreased by 1, depending on whether `To` or `Downto` is used. The control variable must be an ordinal type, no other types can be used as counters in a loop.

Remark

- Free Pascal always calculates the upper bound exactly once before initializing the counter variable with the initial value.
- It is not allowed to change (i. e. assign a value to) the value of a loop variable inside the loop.
- the loop can be exited using the `break` system routine.
- the loop can be forced to the next iteration (if any) using the `continue` system routine.
- The value of the loop variable is undefined after a loop has completed or if a loop is not executed at all. However, if the loop was terminated prematurely with an exception or a `break` or `goto` statement, the loop variable retains the value it had when the loop was exited.
- if the loop variable is the `Result` variable, and the loop is ended with `exit`, the last value before exit will be retained.
- For nested procedures, a loop variable must be a local variable. If you declare a loop variable outside the nested procedure where the loop is, the compiler will complain. It is however allowed to use a global variable in a procedure.
- The compiler does not explicitly forbid jumping with a `goto` statement into a for loop block, but doing so will result in unpredictable behaviour.

The following are valid loops:

```
For Day := Monday to Friday do Work;
For I := 100 downto 1 do
  WriteLn ('Counting down : ',i);
For I := 1 to 7*dwarfs do KissDwarf(i);
```

The following will generate an error:

```
For I:=0 to 100 do
  begin
  DoSomething;
  I:=I*2;
  end;
```

because the loop variable `I` cannot be assigned to inside the loop.

The following will also generate an error:

```
program test;

{$ifdef fpc}
{$mode delphi}
{$h+}
{$endif}

procedure Proc;
var
  i: integer;
  procedure Nested;

  begin
    for i := 1 to 2 do ;
  end;

begin
end;

begin
end.
```

because the variable `I` is not defined in `Nested` and it's not a global variable either.

But the following will compile:

```
program test;

{$ifdef fpc}
{$mode delphi}
{$h+}
{$endif}

var
  i: integer;
```

```
procedure Nested;

begin
  for i := 1 to 2 do ;
end;

begin
end.
```

Inside the loop statement the `Break` and `Continue` system routines can be used to jump to the end or just after the end of the `For` block statement: in the case of `Continue`, this means that the for loop condition is again evaluated. In the case of nested loops, only the innermost loop is considered. Note that `Break` and `Continue` are not reserved words and therefore can be overloaded.

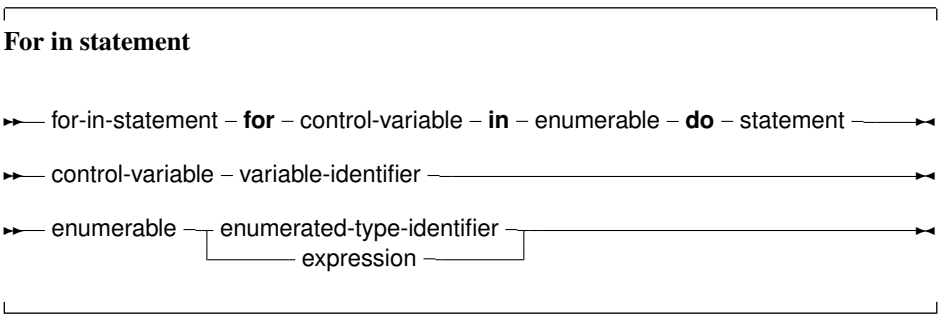
Note that for enumeration values with assigned values, the non-named values will also be iterated over:

```
program Project1;
type
  TEnum = (a, b = 100);
var
  e: TEnum;
begin
  for e := low(TEnum) to high(TEnum) do
    writeln(ord(e));
end.
```

Will print 101 values: 0 to 100.

13.2.5 The `For..in..do` statement

As of version 2.4.2, Free Pascal supports the `For..in` loop construction. A `for..in` loop is used in case one wants to calculate something a fixed number of times with an enumerable loop variable. The prototype syntax is as follows:



Here, `Statement` can be a compound statement. The enumerable must be an expression that consists of a fixed number of elements: the loop variable will be made equal to each of the elements in turn and the statement following the `do` keyword will be executed.

The enumerable expression can be one of five cases:

1. An enumeration type identifier. The loop will then be over all elements of the enumeration type. The control variable must be of the enumeration type. An enumeration type with assigned values cannot be used.

2. A set value. The loop will then be over all elements in the set, the control variable must be of the base type of the set.
3. An array value. The loop will be over all elements in the array, and the control variable must have the same type as an element in the array. As a special case, a string is regarded as an array of characters.
4. An enumerable class, object, or extended record instance. This is an instance of any structured type that supports the `IEnumerator` and `IEnumerable` interfaces. In this case, the control variable's type must equal the type of the `IEnumerator.GetCurrent` return value.
5. Any type for which an `enumerator` operator is defined. The `enumerator` operator must return a structured type that implements the `IEnumerator` interface. The type of the control variable's type must equal the type of the enumerator's `GetCurrent` return value type.

The simplest case of the `for..in` loop is using an enumerated type:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  d : TWeekday;

begin
  for d in TWeekday do
    writeln(d);
end.
```

This will print all week days to the screen.

The above `for..in` construct is equivalent to the following `for..to` construct:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  d : TWeekday;

begin
  for d:=Low(TWeekday) to High(TWeekday) do
    writeln(d);
end.
```

A second case of `for..in` loop is when the enumerable expression is a set, and then the loop will be executed once for each element in the set:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  Week : set of TWeekDay
       = [monday, tuesday, wednesday, thursday, friday];
  d : TWeekday;
```

```
begin
  for d in Week do
    writeln(d);
end.
```

This will print the names of the week days to the screen. Note that the variable `d` is of the same type as the base type of the set.

The above `for..in` construct is equivalent to the following `for..to` construct:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  Week : set of TWeekDay
       = [monday, tuesday, wednesday, thursday, friday];

  d : TWeekday;

begin
  for d:=Low(TWeekday) to High(TWeekday) do
    if d in Week then
      writeln(d);
end.
```

The third possibility for a `for..in` loop is when the enumerable expression is an array:

```
var
  a : Array[1..7] of string
    = ('monday','tuesday','wednesday','thursday',
       'friday','saturday','sunday');

Var
  S : String;

begin
  For s in a do
    Writeln(s);
end.
```

This will also print all days in the week, and is equivalent to

```
var
  a : Array[1..7] of string
    = ('monday','tuesday','wednesday','thursday',
       'friday','saturday','sunday');

Var
  i : integer;

begin
  for i:=Low(a) to high(a) do
    Writeln(a[i]);
end.
```

A `string` type is equivalent to an `array of char`, and therefore a string can be used in a `for..in` loop. The following will print all letters in the alphabet, each letter on a line:

```
Var
  c : char;

begin
 for c in 'abcdefghijklmnopqrstuvwxyz' do
   writeln(c);
end.
```

Note that multi-dimensional arrays are also supported:

```
uses
  SysUtils;

type
  TTestStringArray = array[0..10] of String;

Var
  TwoD : array[0..3] of TTestStringArray;

var
  i,j : integer;
  S : String;
begin
  for i:=0 to 3 do
    for j:=0 to 10 do
      TwoD[i,J]:=Format('%.2dx%.2d',[i,j]);
  for S in twod do
    Writeln(S);
end.
```

This will loop over all dimensions from left to right.

The fourth possibility for a `for..in` loop is using classes. A class can implement the `IEnumerable` interface, which is defined as follows:

```
IEnumerable = interface(IInterface)
  function GetEnumerator: IEnumerator;
end;
```

The actual return type of the `GetEnumerator` must not necessarily be an `IEnumerator` interface, instead, it can be a class which implements the methods of `IEnumerator`:

```
IEnumerator = interface(IInterface)
  function GetCurrent: TObject;
  function MoveNext: Boolean;
  procedure Reset;
  property Current: TObject read GetCurrent;
end;
```

The `Current` property and the `MoveNext` method must be present in the class returned by the `GetEnumerator` method. The actual type of the `Current` property need not be a `TObject`. When encountering a `for..in` loop with a class instance as the "in" operand, the compiler will check each of the following conditions:

- Whether the class in the enumerable expression implements a method `GetEnumerator`
- Whether the result of `GetEnumerator` is a class with the following method:

  ```
  Function MoveNext : Boolean
  ```

- Whether the result of `GetEnumerator` is a class with the following read-only property:

  ```
  Property Current : AType;
  ```

  The type of the property must match the type of the control variable of the `for..in` loop.

Neither the `IEnumerator` nor the `IEnumerable` interfaces must actually be declared by the enumerable class: the compiler will detect whether these interfaces are present using the above checks. The interfaces are only defined for Delphi compatibility and are not used internally. (it would also be impossible to enforce their correctness).

The Classes unit contains a number of classes that are enumerable:

TFPList Enumerates all pointers in the list.

TList Enumerates all pointers in the list.

TCollection Enumerates all items in the collection.

TStringList Enumerates all strings in the list.

TComponent Enumerates all child components owned by the component.

Thus, the following code will also print all days in the week:

```
{$mode objfpc}
uses classes;

Var
  Days : TStrings;
  D : String;

begin
  Days:=TStringList.Create;
  try
    Days.Add('Monday');
    Days.Add('Tuesday');
    Days.Add('Wednesday');
    Days.Add('Thursday');
    Days.Add('Friday');
    Days.Add('Saturday');
    Days.Add('Sunday');
    For D in Days do
      Writeln(D);
  Finally
    Days.Free;
  end;
end.
```

Note that the compiler enforces type safety: declaring `D` as an integer will result in a compiler error:

```
testsl.pp(20,9) Error: Incompatible types: got "AnsiString" expected "LongInt"
```

The above code is equivalent to the following:

```
{$mode objfpc}
uses classes;

Var
  Days : TStrings;
  D : String;
  E : TStringsEnumerator;

begin
  Days:=TStringList.Create;
  try
    Days.Add('Monday');
    Days.Add('Tuesday');
    Days.Add('Wednesday');
    Days.Add('Thursday');
    Days.Add('Friday');
    Days.Add('Saturday');
    Days.Add('Sunday');
    E:=Days.getEnumerator;
    try
      While E.MoveNext do
        begin
        D:=E.Current;
        Writeln(D);
        end;
    Finally
      E.Free;
    end;
  Finally
    Days.Free;
  end;
end.
```

Both programs will output the same result.

The fifth and last possibility to use a `for..in` loop can be used to enumerate almost any type, using the `enumerator` operator. The `enumerator` operator must return a class that has the same signature as the `IEnumerator` approach above. The following code will define an enumerator for the `Integer` type:

```
Type


TEvenEnumerator = Class
  FCurrent : Integer;
  FMax : Integer;
  Function MoveNext : Boolean;
  Property Current : Integer Read FCurrent;
end;

Function TEvenEnumerator.MoveNext : Boolean;
```

```
begin
  FCurrent:=FCurrent+2;
  Result:=FCurrent<=FMax;
end;

operator enumerator(i : integer) : TEvenEnumerator;

begin
  Result:=TEvenEnumerator.Create;
  Result.FMax:=i;
end;

var
  I : Integer;
  m : Integer = 4;

begin
  For I in M do
    Writeln(i);
end.
```

The loop will print all nonzero even numbers smaller or equal to the enumerable. (2 and 4 in the case of the example).

Care must be taken when defining enumerator operators: the compiler will find and use the first available enumerator operator for the enumerable expression. For classes this also means that the `GetEnumerator` method is not even considered. The following code will define an enumerator operator which extracts the object from a stringlist:

```
{$mode objfpc}
uses classes;

Type
  TDayObject = Class
    DayOfWeek : Integer;
    Constructor Create(ADayOfWeek : Integer);
  end;

  TObjectEnumerator = Class
    FList : TStrings;
    FIndex : Integer;
    Function GetCurrent : TDayObject;
    Function MoveNext: boolean;
    Property Current : TDayObject Read GetCurrent;
  end;

Constructor TDayObject.Create(ADayOfWeek : Integer);

begin
  DayOfWeek:=ADayOfWeek;
end;

Function TObjectEnumerator.GetCurrent : TDayObject;
begin
```

```
  Result:=FList.Objects[Findex] as TDayObject;
end;

Function TObjectEnumerator.MoveNext: boolean;

begin
  Inc(FIndex);
  Result:=(FIndex<FList.Count);
end;

operator enumerator (s : TStrings) : TObjectEnumerator;

begin
  Result:=TObjectEnumerator.Create;
  Result.Flist:=S;
  Result.FIndex:=-1;
end;

Var
  Days : TStrings;
  D : String;
  O : TdayObject;

begin
  Days:=TStringList.Create;
  try
    Days.AddObject('Monday',TDayObject.Create(1));
    Days.AddObject('Tuesday',TDayObject.Create(2));
    Days.AddObject('Wednesday',TDayObject.Create(3));
    Days.AddObject('Thursday',TDayObject.Create(4));
    Days.AddObject('Friday',TDayObject.Create(5));
    Days.AddObject('Saturday',TDayObject.Create(6));
    Days.AddObject('Sunday',TDayObject.Create(7));
    For O in Days do
      Writeln(O.DayOfWeek);
  Finally
    Days.Free;
  end;
end.
```

The above code will print the day of the week for each day in the week.

If a class is not enumerable, the compiler will report an error when it is encountered in a `for...in` loop.

Remark

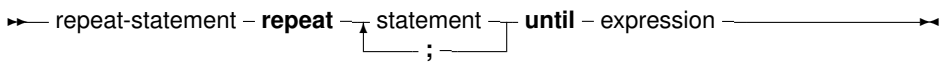
- A loop can be exited prematurely using the `break` system routine.
- The next iteration of a loop can be forced using the `continue` system routine.
- Like the `for..to` loop, it is not allowed to change (i. e. assign a value to) the value of a loop control variable inside the loop.
- The value of the loop variable is undefined after a loop has completed or if a loop is not executed at all. However, if the loop was terminated prematurely with an exception or a `break` or `goto` statement, the loop variable retains the value it had when the loop was exited.

- if the loop variable is the `Result` variable, and the loop is ended with `exit`, the last value before exit will be retained.

13.2.6 The `Repeat..until` statement

The `repeat` statement is used to execute a statement until a certain condition is reached. The statement will be executed at least once. The prototype syntax of the `Repeat..until` statement is

Repeat statement



This will execute the statements between `repeat` and `until` up to the moment when `Expression` evaluates to `True`. Since the `expression` is evaluated *after* the execution of the statements, they are executed at least once.

Be aware of the fact that the boolean expression `Expression` will be short-cut evaluated by default, meaning that the evaluation will be stopped at the point where the outcome is known with certainty.

The following are valid `repeat` statements

```
repeat
  WriteLn ('I =',i);
  I := I+2;
until I>100;

repeat
 X := X/2
until x<10e-3;
```

Note that the last statement before the `until` keyword does not need a terminating semicolon, but it is allowed.

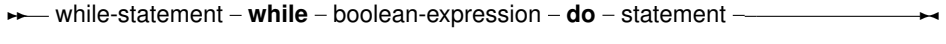
The `Break` and `Continue` system routines can be used to jump to the end or just after the end of the `repeat .. until` statement. In the case of `Continue`, this has the effect of immediatly checking the loop condition, bypassing all code between `Continue` and the `until` keyword. In the case of nested loops, only the innermost loop is considered. Note that `Break` and `Continue` are not reserved words and therefore can be overloaded.

13.2.7 The `While..do` statement

A `while` statement is used to execute a statement as long as a certain condition holds. In difference with the `repeat` loop, this may imply that the statement is never executed.

The prototype syntax of the `While..do` statement is

While statements



This will execute `Statement` as long as `Expression` evaluates to`True`. Since `Expression` is evaluated *before* the execution of `Statement`, it is possible that `Statement` isn't executed at all. `Statement` can be a compound statement.

Be aware of the fact that the boolean expression `Expression` will be short-cut evaluated by default, meaning that the evaluation will be stopped at the point where the outcome is known with certainty.

The following are valid `while` statements:

```
I := I+2;
while i<=100 do
  begin
  WriteLn ('I =',i);
  I := I+2;
  end;
X := X/2;
while x>=10e-3 do
  X := X/2;
```

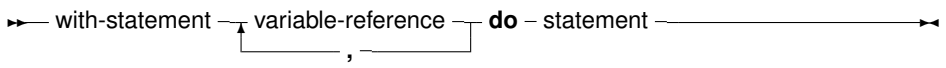
They correspond to the example loops for the `repeat` statements.

In the while statement the `Break` and `Continue` system routines can be used to jump to the end or just after the end of the `While` block. In the case of `Continue`, this has the effect of immediatly checking the loop condition, bypassing all code between `Continue` and the end of the loop block. In the case of nested loops, only the innermost loop is considered. Note that `Break` and `Continue` are not reserved words and therefore can be overloaded.

13.2.8 The `With` statement

The `with` statement serves to access the elements of a record or object or class, without having to specify the element's name each time. The syntax for a `with` statement is

With statement



The variable reference must be a variable of a record, object or class type. In the `with` statement, any variable reference, or method reference is checked to see if it is a field or method of the record or object or class. If so, then that field is accessed, or that method is called. Given the declaration:

```
Type
  Passenger = Record
    Name : String[30];
    Flight : String[10];
  end;

Var
  TheCustomer : Passenger;
```

The following statements are completely equivalent:

```
TheCustomer.Name := 'Michael';
TheCustomer.Flight := 'PS901';
```

and

```
With TheCustomer do
  begin
  Name := 'Michael';
  Flight := 'PS901';
  end;
```

The statement

```
With A,B,C,D do Statement;
```

is equivalent to

```
With A do
 With B do
  With C do
   With D do Statement;
```

This also is a clear example of the fact that the variables are tried *last to first*, i. e., when the compiler encounters a variable reference, it will first check if it is a field or method of the last variable. If not, then it will check the last-but-one, and so on. The following example shows this;

```
Program testw;
Type AR = record
      X,Y : Longint;
     end;
     PAR = ^Ar;

Var S,T : Ar;
begin
  S.X := 1;S.Y := 1;
  T.X := 2;T.Y := 2;
  With S,T do
    WriteLn (X,' ',Y);
end.
```

The output of this program is

```
2 2
```

Showing thus that the `X,Y` in the `WriteLn` statement match the `T` record variable.

Remark When using a `With` statement with a pointer, or a class, it is not permitted to change the pointer or the class in the `With` block. With the definitions of the previous example, the following illustrates what it is about:

```
Var p : PAR;

begin
  With P^ do
   begin
```

```
  // Do some operations
  P:=OtherP;
  X:=0.0;  // Wrong X will be used !!
  end;
```

The reason the pointer cannot be changed is that the address is stored by the compiler in a temporary register. Changing the pointer won't change the temporary address. The same is true for classes.

Remark There is a subtle difference between Delphi and Free Pascal when resolving identifiers in a `With` statement: Free Pascal will always search first in the scope of the expression used in the `With` statement, and only then will it search in the local scope. Delphi will first search in the local scope and then only in the with scope:

```
Type
  TField = Record
    Procedure Updating;
  end;

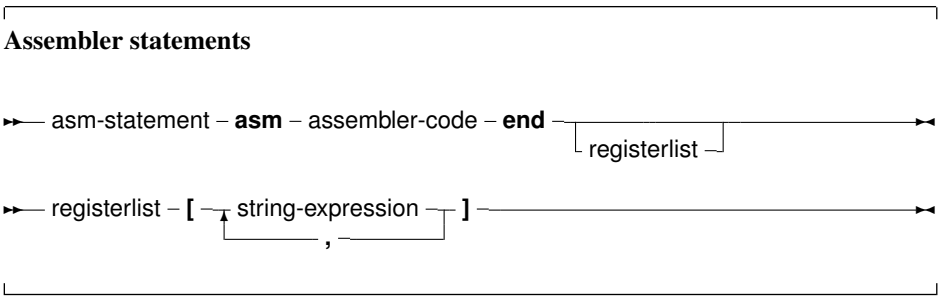
procedure Test;
var
  Updating: Boolean;
  F : TField;
begin
  Updating := Solo;
  // ..
  with F do
    Updating // TField.Updating in FPC, variable updating in Delphi
  // ..
```

13.2.9 Exception Statements

Free Pascal supports exceptions. Exceptions provide a convenient way to program error and error-recovery mechanisms, and are closely related to classes. Exception support is explained in chapter 17, page 272

13.3 Assembler statements

An assembler statement allows to insert assembler code right in the Pascal code.



More information about assembler blocks can be found in the Programmer's Guide. The register list is used to indicate the registers that are modified by an assembler statement in the assembler block. The compiler stores certain results in the registers. If the registers are modified in an assembler

statement, the compiler should, sometimes, be told about it. The registers are denoted with their Intel names for the I386 processor, i. e., 'EAX', 'ESI' etc... As an example, consider the following assembler code:

```
asm
  Movl $1,%ebx
  Movl $0,%eax
  addl %eax,%ebx
end ['EAX','EBX'];
```

This will tell the compiler that it should save and restore the contents of the EAX and EBX registers when it encounters this asm statement.

Free Pascal supports various styles of assembler syntax. By default, AT&T syntax is assumed for the 80386 and compatibles platform. The default assembler style can be changed with the {$asmmode xxx} switch in the code, or the -R command-line option. More about this can be found in the Programmer's Guide.

Chapter 14

Using functions and procedures

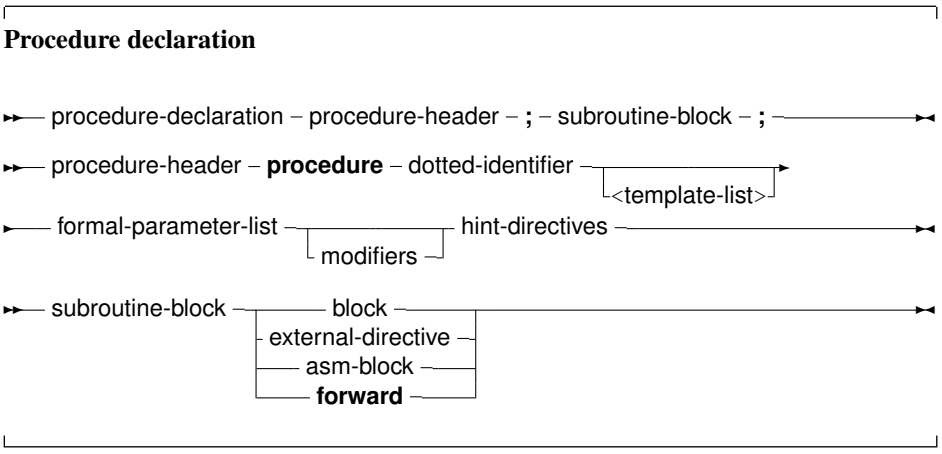
Free Pascal supports the use of functions and procedures. It supports

- Function overloading, i. e. functions with the same name but different parameter lists.
- `Const` parameters.
- Open arrays (i. e. arrays without bounds).
- Variable number of arguments as in C.
- Return-like construct as in C, through the `Exit` keyword.

Remark In many of the subsequent paragraphs the words `procedure` and `function` will be used interchangeably. The statements made are valid for both, except when indicated otherwise.

14.1 Procedure declaration

A procedure declaration defines an identifier and associates it with a block of code. The procedure can then be called with a procedure statement.



See section 14.4, page 221 for the list of parameters. Note that procedures can be generic, see section 8.5, page 138 for more details.

A procedure declaration that is followed by a block implements the action of the procedure in that block. The following is a valid procedure:

```
Procedure DoSomething (Para : String);
begin
  Writeln ('Got parameter : ',Para);
  Writeln ('Parameter in upper case : ',Upper(Para));
end;
```

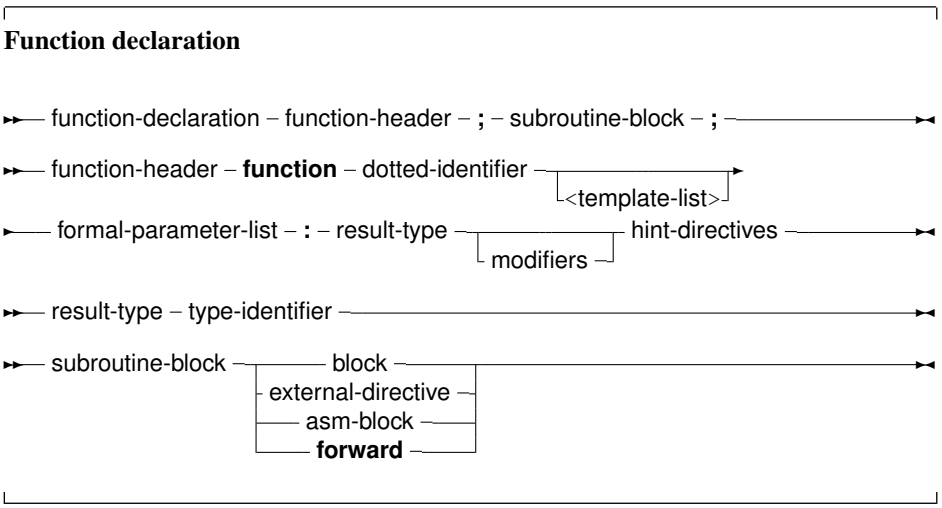
As with most identifiers, a procedure can have hint directives (section 1.5, page 17):

```
procedure something; deprecated;
```

Note that it is possible that a procedure calls itself.

14.2 Function declaration

A function declaration defines an identifier and associates it with a block of code. The block of code will return a result. The function can then be called inside an expression, or with a procedure statement, if extended syntax is on.



The result type of a function can be any previously declared type. contrary to Turbo Pascal, where only simple types could be returned. Note that functions can be generic, see section 8.5, page 138 for more details.

Just as procedures, functions can have hint directives (section 1.5, page 17):

```
function something : string; deprecated;
```

14.3 Function results

The result of a function can be set by setting the result variable: this can be the function identifier or, (only in `ObjFPC` or `Delphi` mode) the special `Result` identifier:

```
Function MyFunction : Integer;

begin
  MyFunction:=12; // Return 12
end;
```

In `Delphi` or `ObjFPC` mode, the above can also be coded as:

```
Function MyFunction : Integer;

begin
  Result:=12;
end;
```

As an extension to Delphi syntax, the `ObjFPC` mode also supports a special extension of the `Exit` procedure:

```
Function MyFunction : Integer;

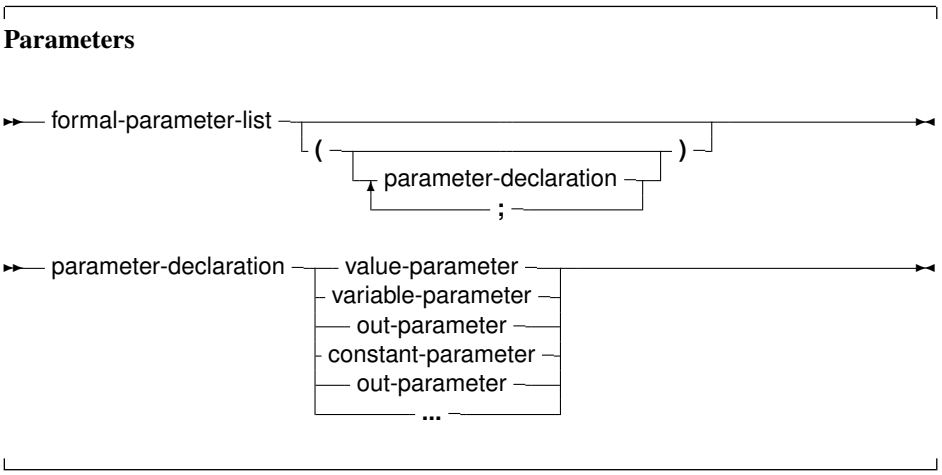
begin
  Exit(12);
end;
```

The `Exit` call sets the result of the function and jumps to the final `End` of the function declaration block. It can be seen as the equivalent of the C `return` instruction.

Remark Function results are treated as pass-by-reference parameters. That is especially important for managed types: The function result may be non-nil on entry, and set to a valid instance of the type.

14.4 Parameter lists

When arguments must be passed to a function or procedure, these parameters must be declared in the formal parameter list of that function or procedure. The parameter list is a declaration of identifiers that can be referred to only in that procedure or function's block.

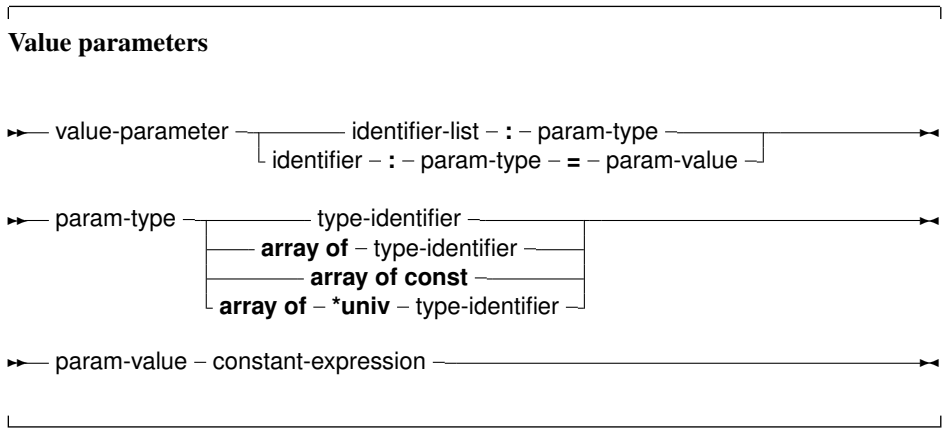


Constant parameters, out parameters and variable parameters can also be `untyped` parameters if they have no type identifier.

As of version 1.1, Free Pascal supports default values for both constant parameters and value parameters, but only for simple types. The compiler must be in `OBJFPC` or `DELPHI` mode to accept default values.

14.4.1 Value parameters

Value parameters are declared as follows:



When parameters are declared as value parameters, the procedure gets *a copy* of the parameters that the calling statement passes. Any modifications to these parameters are purely local to the procedure's block, and do not propagate back to the calling block.

A block that wishes to call a procedure with value parameters must pass assignment compatible parameters to the procedure. This means that the types should not match exactly, but can be converted to the actual parameter types. This conversion code is inserted by the compiler itself.

Care must be taken when using value parameters: value parameters makes heavy use of the stack, especially when using large parameters. The total size of all parameters in the formal parameter list should be below 32K for portability's sake (the Intel version limits this to 64K).

Open arrays can be passed as value parameters. See section 14.4.5, page 226 for more information on using open arrays.

For a parameter of a simple type (i. e. not a structured type), a default value can be specified. This can be an untyped constant. If the function call omits the parameter, the default value will be passed on to the function. For dynamic arrays or other types that can be considered as equivalent to a pointer, the only possible default value is `Nil`.

The following example will print 20 on the screen:

```
program testp;

Const
  MyConst = 20;

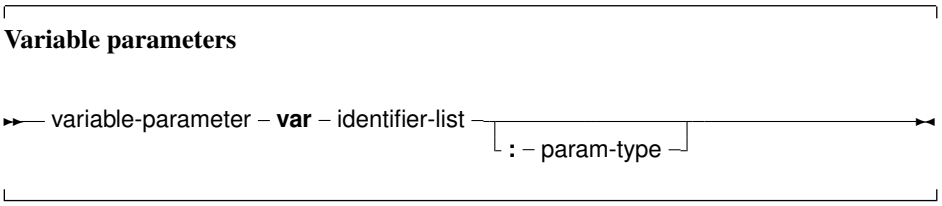
Procedure MyRealFunc(I : Integer = MyConst);

begin
  Writeln('Function received : ',I);
end;

begin
  MyRealFunc;
end.
```

14.4.2 Variable parameters

Variable parameters are declared as follows:



When parameters are declared as variable parameters, the procedure or function accesses immediately the variable that the calling block passed in its parameter list. The procedure gets a pointer to the variable that was passed, and uses this pointer to access the variable's value. From this, it follows that any changes made to the parameter, will propagate back to the calling block. This mechanism can be used to pass values back in procedures. Because of this, the calling block must pass a parameter of *exactly* the same type as the declared parameter's type. If it does not, the compiler will generate an error.

Variable and constant parameters can be untyped. In that case the variable has no type, and hence is incompatible with all other types. However, the address operator can be used on it, or it can be passed to a function that has also an untyped parameter. If an untyped parameter is used in an assignment, or a value must be assigned to it, a typecast must be used.

File type variables must always be passed as variable parameters.

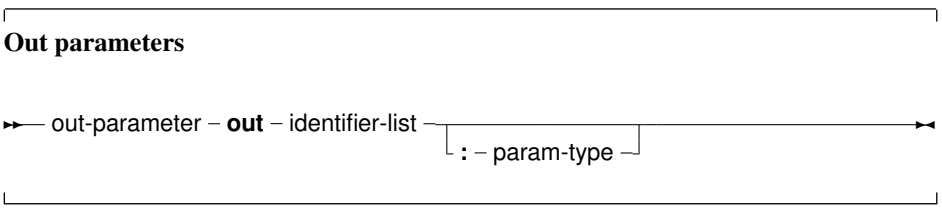
Open arrays can be passed as variable parameters. See section 14.4.5, page 226 for more information on using open arrays.

Remark

- Note that default values are not supported for variable parameters. This would make little sense since it defeats the purpose of being able to pass a value back to the caller.
- A function result is internally treated as a variable parameter, and may have a non-zero (or non-nil) initial value. This is especially important for managed types.

14.4.3 Out parameters

Out parameters (output parameters) are declared as follows:



The purpose of an `out` parameter is to pass values back to the calling routine: the variable is passed by reference. The initial value of the parameter on function entry is discarded, and should not be used.

If a variable must be used to pass a value to a function and retrieve data from the function, then a variable parameter must be used. If only a value must be retrieved, a `out` parameter can be used.

Needless to say, default values are not supported for `out` parameters.

The difference of `out` parameters and parameters by reference is very small (however, see below for managed types): the former gives the compiler more information about what happens to the arguments when passed to the procedure: it knows that the variable does not have to be initialized prior to the call. The following example illustrates this:

```
Procedure DoA(Var A : Integer);

begin
  A:=2;
  Writeln('A is ',A);
end;

Procedure DoB(Out B : Integer);

begin
  B:=2;
  Writeln('B is ',B);
end;

Var
  C,D : Integer;

begin
  DoA(C);
  DoB(D);
end.
```

Both procedures `DoA` and `DoB` do practically the same. But `DoB`'s declaration gives more information to the compiler, allowing it to detect that `D` does not have to initialized before `DoB` is called. Since the parameter `A` in `DoA` can receive a value as well as return one, the compiler notices that `C` was not initialized prior to the call to `DoA`:

```
home: >fpc -S2 -vwhn testo.pp
testo.pp(19,8) Hint: Variable "C" does not seem to be initialized
```

This shows that it is better to use `out` parameters when the parameter is used only to return a value.

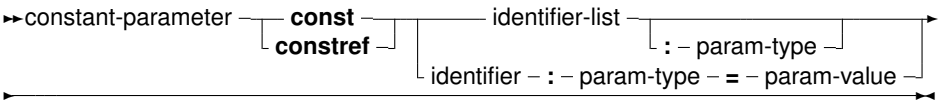
Remark `Out` parameters are only supported in `Delphi` and `ObjFPC` mode. For the other modes, `out` is a valid identifier.

Remark For managed types (reference counted types), using `Out` parameters incurs some overhead: the compiler must be sure that the value is correctly initialized (i. e. has a reference count of zero (0)). This initialization is normally done by the caller.

14.4.4 Constant parameters

In addition to variable parameters and value parameters Free Pascal also supports constant parameters. A constant parameter can be specified as follows:

Constant parameters



Specifying a parameter as constant using `const` or `constref` is giving the compiler a hint that the contents of the parameter will not be changed by the called routine. `constref` additionally tells the compiler the value should be passed by reference (just as a `var` parameter).

This allows the compiler to perform optimizations which it could not do otherwise, and also to perform certain checks on the code inside the routine: namely, it can forbid assignments to the parameter. Furthermore a const parameter cannot be passed on to another function that requires a variable parameter: the compiler can check this as well. The main use for this is reducing the stack size, hence improving performance, and still retaining the semantics of passing by value...

Remark Contrary to earlier versions of Delphi, no assumptions should be made about how `const` parameters are passed to the underlying routine. In particular, the assumption that parameters with large size are passed by reference is not correct. For this the `constref` parameter type should be used, which is available as of version 2.5.1 of the compiler.

An exception is the `stdcall` calling convention: for compatibility with COM standards, large const parameters are passed by reference.

Remark Note that specifying `const` is a contract between the programmer and the compiler. It is the programmer who tells the compiler that the contents of the const parameter will not be changed when the routine is executed, it is *not* the compiler who tells the programmer that the parameter will not be changed.

This is particularly important and visible when using refcounted types. For such types, the (invisible) incrementing and decrementing of any reference count is omitted when `const` is used. Doing so often allows the compiler to omit invisible try/finally frames for these routines.

As a side effect, the following code will produce not the expected output:

```
Var
  S : String = 'Something';

Procedure DoIt(Const T : String);

begin
  S:='Something else';
  Writeln(T);
end;

begin
  DoIt(S);
end.
```

Will write

```
Something else
```

This behavior is by design.

Constant parameters can also be untyped. See section 14.4.2, page 223 for more information about untyped parameters.

As for value parameters, constant parameters can get default values.

Open arrays can be passed as constant parameters. See section 14.4.5, page 226 for more information on using open arrays.

14.4.5 Open array parameters

Free Pascal supports the passing of open arrays, i. e. a procedure can be declared with an array of unspecified length as a parameter, as in Delphi. Open array parameters can be accessed in the procedure or function as an array that is declared with starting index 0, and last element index `High(parameter)`. For example, the parameter

```
Row : Array of Integer;
```

would be equivalent to

```
Row : Array[0..N-1] of Integer;
```

Where `N` would be the actual size of the array that is passed to the function. `N-1` can be calculated as `High(Row)`.

Specifically, if an empty array is passed, then `High(Parameter)` returns -1, while `low(Parameter)` returns 0.

Open parameters can be passed by value, by reference or as a constant parameter. In the latter cases the procedure receives a pointer to the actual array. In the former case, it receives a copy of the array. In a function or procedure, open arrays can only be passed to functions which are also declared with open arrays as parameters, *not* to functions or procedures which accept arrays of fixed length. The following is an example of a function using an open array:

```
Function Average (Row : Array of integer) : Real;
Var I : longint;
    Temp : Real;
begin
  Temp := Row[0];
  For I := 1 to High(Row) do
    Temp := Temp + Row[i];
  Average := Temp / (High(Row)+1);
end;
```

As of FPC 2.2, it is also possible to pass partial arrays to a function that accepts an open array. This can be done by specifying the range of the array which should be passed to the open array.

Given the declaration

```
Var
  A : Array[1..100];
```

the following call will compute and print the average of the 100 numbers:

```
  Writeln('Average of 100 numbers: ',Average(A));
```

But the following will compute and print the average of the first and second half:

```
  Writeln('Average of first 50 numbers: ',Average(A[1..50]));
  Writeln('Average of last  50 numbers: ',Average(A[51..100]));
```

14.4.6 Macpas: UNIV typed parameters

In Macpas mode, you can specify a `UNIV` (universal) parameter. This signifies that any parameter with the exact same byte size as the declared parameter can be passed as the value of the parameter: Given the following procedure declaration:

```
procedure HDispose( var theHandle: univ Handle);
```

You can pass value of any type that has the same size as `Handle`.

This comes in especially handy for typed procedures. Given the following definition:

```
Type
  TComparer = function (theIndex: Int32; theValuePtr : univ Pointer): boolean;

function Find(
      theValuePtr    : univ UnivPtr;
      theFirstIndex  : Int32;
      theLastIndex   : Int32;
      SmallerThan    : TComparer;
      EqualTo        : TComparer;
      var theFoundIndex   : Int32): boolean;
```

You can call `Find` as follows

```
 Result:=Find(@theCmdID,1,kCmdCount,IsSmallerThan, IsEqualTo,aIndex):
```

Where the two functions `IsSmallerThan` and `IsEqualTo` are defined as:

```
function IsSmallerThan(theIndex : Int32; aPtr : PtrInt): boolean;
function IsEqualTo(theIndex : Int32; aPtr  : PtrInt): boolean
```

Because `PtrInt` has the same size as `Pointer`, this will compile.

14.4.7 Array of const

In Object Pascal or Delphi mode, Free Pascal supports the `Array of Const` construction to pass parameters to a subroutine.

This is a special case of the `Open array` construction, where it is allowed to pass any expression in an array to a function or procedure. The expression must have a simple result type: structures cannot be passed as an argument. This means that all ordinal, float or string types can be passed, as well as pointers, classes and interfaces (since the latter two are actually pointers).

The elements of the `array of const` are converted to a special variant record:

```
Type
  PVarRec = ^TVarRec;
  TVarRec = record
     case VType : Ptrint of
       vtInteger    : (VInteger: Longint);
       vtBoolean    : (VBoolean: Boolean);
       vtChar       : (VChar: Char);
       vtWideChar   : (VWideChar: WideChar);
       vtExtended   : (VExtended: PExtended);
       vtString     : (VString: PShortString);
       vtPointer    : (VPointer: Pointer);
       vtPChar      : (VPChar: PChar);
       vtObject     : (VObject: TObject);
       vtClass      : (VClass: TClass);
       vtPWideChar  : (VPWideChar: PWideChar);
```

```
       vtAnsiString : (VAnsiString: Pointer);
       vtCurrency   : (VCurrency: PCurrency);
       vtVariant    : (VVariant: PVariant);
       vtInterface  : (VInterface: Pointer);
       vtWideString : (VWideString: Pointer);
       vtInt64      : (VInt64: PInt64);
       vtQWord      : (VQWord: PQWord);
   end;
```

Therefore, inside the procedure body, the `array of const` argument is equivalent to an open array of `TVarRec`:

```
Procedure Testit (Args: Array of const);

Var I : longint;

begin
  If High(Args)<0 then
    begin
    Writeln ('No aguments');
    exit;
    end;
  Writeln ('Got ',High(Args)+1,' arguments :');
  For i:=0 to High(Args) do
    begin
    write ('Argument ',i,' has type ');
    case Args[i].vtype of
      vtinteger    :
        Writeln ('Integer, Value :',args[i].vinteger);
      vtboolean    :
        Writeln ('Boolean, Value :',args[i].vboolean);
      vtchar       :
        Writeln ('Char, value : ',args[i].vchar);
      vtextended   :
        Writeln ('Extended, value : ',args[i].VExtended^);
      vtString     :
        Writeln ('ShortString, value :',args[i].VString^);
      vtPointer    :
        Writeln ('Pointer, value : ',Longint(Args[i].VPointer));
      vtPChar      :
        Writeln ('PChar, value : ',Args[i].VPChar);
      vtObject     :
        Writeln ('Object, name : ',Args[i].VObject.Classname);
      vtClass      :
        Writeln ('Class reference, name :',Args[i].VClass.Classname);
      vtAnsiString :
        Writeln ('AnsiString, value :',AnsiString(Args[I].VAnsiString);
    else
        Writeln ('(Unknown) : ',args[i].vtype);
    end;
    end;
end;
```

In code, it is possible to pass an arbitrary array of elements to this procedure:

```
  S:='Ansistring 1';
  T:='AnsiString 2';
  Testit ([]);
  Testit ([1,2]);
  Testit (['A','B']);
  Testit ([TRUE,FALSE,TRUE]);
  Testit (['String','Another string']);
  Testit ([S,T])  ;
  Testit ([P1,P2]);
  Testit ([@testit,Nil]);
  Testit ([ObjA,ObjB]);
  Testit ([1.234,1.234]);
  TestIt ([AClass]);
```

If the procedure is declared with the `cdecl` modifier, then the compiler will pass the array as a C compiler would pass it. This, in effect, emulates the C construct of a variable number of arguments, as the following example will show:

```
program testaocc;
{$mode objfpc}

Const
  P : PChar = 'example';
  Fmt : PChar =
        'This %s uses printf to print numbers (%d) and strings.'#10;

// Declaration of standard C function printf:
procedure printf (fm : pchar; args : array of const);cdecl; external 'c';

begin
 printf(Fmt,[P,123]);
end.
```

Remark Note that this is not true for Delphi, so code relying on this feature will not be portable.

Remark Note that there is no support for `DWord` (or cardinal) arguments in `array of const`. They are converted to `vtInteger`/`vtLongint`. This is for Delphi compatibility, and the compiler will ignore any resulting range checks when in mode Delphi.

14.4.8 Untyped parameters

Variable, Out and constant parameters can be untyped. In that case the variable has no type in the procedure of function, and hence is incompatible with all other types: The compiler simply passes the address of the passed variable to the routine, so all that is available in the called routine is an address, with no type information attached to it. This is also true for const parameters.

Using an analogy

```
procedure Something(const Data; Len: cint);
```

Is equivalent to the following C/C++ declaration:

```
void Something(void* Data; int Len);
```

This means pretty much all that one can do in the routine is use the address operator or pass the argument to another function that has also an untyped parameter.

Inside a routine with an untyped parameter, if the untyped parameter is used in an expression, or a value must be assigned to it a typecast must always be used.

The following example shows this:

```
{$mode objfpc}
uses types;

procedure doit(const d);

begin
  Writeln('As integer: ',PInteger(@D)^);
  Writeln('As Byte   : ',PByte(@D)^);
end;

Var
  D : Integer;

begin
  D:=$0FFFFFF0;
  DoIt(D);
end.
```

This will write:

```
As integer: 268435440
As Byte   : 240
```

Note that because an address is needed, no constant expressions can be passed to the function or procedure, i. e. given the above definition, the following will not work:

```
 DoIt($0FFFFFF0);
```

This will result in the following error:

```
Error: Variable identifier expected
```

14.4.9 Ellipsis parameter

In macpas mode you can have an ellipsis parameter. The ellipsis (`...`) parameter can appear only once, and must be the last parameter in the parameter list. Additionally it can only appear after at least one other parameter. It has the same effect as the `varargs` modifier: it tells the compiler that a variable number of arguments is possible.

This means the following external definitions are equivalent in macpas mode:

```
Procedure printf(format: PAnsiChar;...); cdecl; external;

Procedure printf(format: PAnsiChar); varargs; cdecl; external;
```

14.4.10 Managed types and reference counts

Some types (Unicodestring, Ansistring, interfaces, dynamic arrays) are treated somewhat specially by the compiler: the data has a reference count which is increased or decreased depending on how many reference to the data exists.

The qualifiers for parameters in function or procedure calls influence what happens to the reference count of the managed types:

- nothing (pass by value): the reference count of the parameter is increased on entry and decreased on exit.
- `out`: the reference count of the value that is passed in is decreased by 1, and the variable that's passed into the procedure is initialized to "empty" (usually `Nil`, but that is an implementation detail which should not be relied on).
- `var` nothing happens to the reference count. A reference to the original variable is passed in, and changing it or reading it has exactly the same effect as changing/reading the original variable.
- `const` this case is slightly tricky. Nothing happens to the reference count because you can pass non-values here. In particular, you can pass a class implementing an interface rather than the interface itself which can cause the class to be freed unexpectedly.

Remark The function result is internally treated as a var parameter to the function, and the same rules as for var parameters apply.

The following example demonstrates the dangers:

```
{$mode objfpc}

Type
  ITest = Interface
    Procedure DoTest(ACount : Integer);
  end;

  TTest = Class(TInterfacedObject,ITest)
    Procedure DoTest(ACount : Integer);
    Destructor destroy; override;
  end;

Destructor TTest.Destroy;

begin
  Writeln('Destroy called');
end;

Procedure TTest.DoTest(ACount : Integer);

begin
  Writeln('Test ',ACount,' : ref count: ',RefCount);
end;

procedure DoIt1(x: ITest; ACount : Integer);

begin
  // Reference count is increased
  x.DoTest(ACount);
  // And decreased
end;

procedure DoIt2(const x: ITest; ACount : Integer);
```

```
begin
  // No change to reference count.
  x.DoTest(ACount);
end;

Procedure Test1;

var
  y: ITest;
begin
  y := TTest.Create;
  // Ref. count is 1 at this point.
  y.DoTest(1);
  // Calling DoIT will increase reference count and decrease on exit.
  DoIt1(y,2);
  // Reference count is still one.
  y.DoTest(3);
end;

Procedure Test2;

var
  Y : TTest;
begin
  Y := TTest.Create; // no count on the object yet
  // Ref. count is 0 at this point.
  y.DoTest(3);
  // Ref count will remain zero.
  DoIt2(y,4);
  Y.DoTest(5);
  Y.Free;
end;

Procedure Test3;

var
  Y : TTest;
begin
  Y := TTest.Create; // no count on the object yet
  // Ref. count is 0 at this point.
  y.DoTest(6);
  // Ref count will remain zero.
  DoIt1(y,7);
  y.DoTest(8);
end;


begin
  Test1;
  Test2;
  Test3;
end.
```

The output of this example is:

```
Test 1 : ref count: 1
Test 2 : ref count: 2
Test 3 : ref count: 1
Destroy called
Test 3 : ref count: 0
Test 4 : ref count: 0
Test 5 : ref count: 0
Destroy called
Test 6 : ref count: 0
Test 7 : ref count: 1
Destroy called
Test 8 : ref count: 0
```

As can be seen, in test3, the reference count is decreased from 1 to 0 at the end of the DoIt call, causing the instance to be freed before the call returns.

The following small program demonstrates the reference counts used in strings:

```
{$mode objfpc}
{$H+}

Procedure ByVar(Var S : string);

begin
  Writeln('By var, ref count : ',StringRefCount(S));
end;

Procedure ByConst(Const S : string);

begin
  Writeln('Const, ref count : ',StringRefCount(S));
end;

Procedure ByVal(S : string);

begin
  Writeln('Value, ref count : ',StringRefCount(S));
end;

Function FunctionResult(Var S : String) : String;

begin
  Writeln('Function argument, ref count : ',StringRefCount(S));
  Writeln('Function result, ref count : ',StringRefCount(Result));
end;

Var
  S,T : String;

begin
  S:='Some string';
  Writeln('Constant       : ',StringRefCount(S));
  UniqueString(S);
```

```
  Writeln('Unique         : ',StringRefCount(S));
  T:=S;
  Writeln('After Assign   : ',StringRefCount(S));
  ByVar(S);
  ByConst(S);
  ByVal(S);
  UniqueString(S);
  T:=FunctionResult(S);
  Writeln('After function : ',StringRefCount(S));
end.
```

14.5 Function overloading

Function overloading simply means that the same function is defined more than once, but each time with a different formal parameter list. The parameter lists must differ at least in one of its elements type. When the compiler encounters a function call, it will look at the function parameters to decide which one of the defined functions it should call. This can be useful when the same function must be defined for different types.

For example, in a hypothetical RTL, the `Random` procedure could be defined as:

```
Random(Max : Longint);
Random(Max : Double);
Random(Max,Ofsset : Longint);
```

When the compiler encounters a call to the `Random` function, it will first search which function it should use. It therefore checks the parameters in a function call, and looks if there is a function definition which matches the specified parameter list. If the compiler finds such a function, a call is inserted to that function. If no such function is found, a compiler error is generated.

Functions that have a `cdecl` modifier cannot be overloaded. (Technically, because this modifier prevents the mangling of the function name by the compiler).

Prior to version 1.9 of the compiler, the overloaded functions needed to be in the same unit. Now the compiler will continue searching in other units if it doesn't find a matching version of an overloaded function in one unit, and if the `overload` keyword is present.

If the `overload` keyword is not present, then all overloaded versions must reside in the same unit.

Remark Note that if you redefine (and overload) compiler internal functions, the original compiler functions are no longer available. This is for instance the case with `Inc()` and `Dec()`

14.6 Forward declared functions

A function can be declared without having it followed by its implementation, by having it followed by the `forward` procedure. The effective implementation of that function must follow later in the module. The function can be used after a `forward` declaration as if it had been implemented already. The following is an example of a forward declaration.

```
Program testforward;
Procedure First (n : longint); forward;
Procedure Second;
begin
  WriteLn ('In second. Calling first...');
```

```
  First (1);
end;
Procedure First (n : longint);
begin
  WriteLn ('First received : ',n);
end;
begin
  Second;
end.
```

A function can be forward declared only once. Likewise, in units, it is not allowed to have a forward declared function of a function that has been declared in the interface part. The interface declaration counts as a `forward` declaration. The following unit will give an error when compiled:

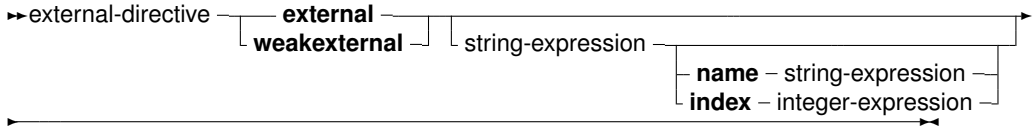
```
Unit testforward;
interface
Procedure First (n : longint);
Procedure Second;
implementation
Procedure First (n : longint); forward;
Procedure Second;
begin
  WriteLn ('In second. Calling first...');
  First (1);
end;
Procedure First (n : longint);
begin
  WriteLn ('First received : ',n);
end;
end.
```

Reversely, functions declared in the interface section cannot be declared forward in the implementation section. Logically, since they already have been declared.

14.7 External functions

The `external` modifier can be used to declare a function that resides in an external object file. It allows to use the function in some code, and at linking time, the object file containing the implementation of the function or procedure must be linked in.

External directive



It replaces, in effect, the function or procedure code block. As an example:

```
program CmodDemo;
{$Linklib c}
Const P : PChar = 'This is fun !';
Function strlen (P : PChar) : Longint; cdecl; external;
begin
  WriteLn ('Length of (',p,') : ',strlen(p))
end.
```

Remark The parameters in the declaration of the `external` function should match exactly the ones in the declaration in the object file.

If the `external` modifier is followed by a string constant:

```
external 'lname';
```

Then this tells the compiler that the function resides in library "lname". The compiler will then automatically link this library to the program.

The name that the function has in the library can also be specified:

```
external 'lname' name 'Fname';
```

This tells the compiler that the function resides in library "lname", but with name "Fname". The compiler will then automatically link this library to the program, and use the correct name for the function. Under WINDOWS and OS/2, the following form can also be used:

```
external 'lname' Index Ind;
```

This tells the compiler that the function resides in library "lname", but with index `Ind`. The compiler will then automatically link this library to the program, and use the correct index for the function.

Finally, the external directive can be used to specify the external name of the function:

```
external name 'Fname';
{$L myfunc.o}
```

This tells the compiler that the function has the name "Fname". The correct library or object file (in this case myfunc.o) must still be linked, ensuring that the function "Fname" is indeed included in the linking stage.

Sometimes an external symbol may be missing. If the platform supports it, the `weakexternal` keyword can be used instead of the `external` keyword:

```
function _test: cardinal; weakexternal libname name '_TEST';
```

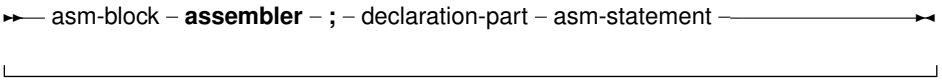
If the function is not there at runtime, the function reference will be `Nil`. This means that before using such a function, you need to test if it is actually there:

```
if @_test<>Nil then
  writeln(_test);
```

14.8 Assembler functions

Functions and procedures can be completely implemented in assembly language. To indicate this, use the `assembler` keyword:

Assembler functions

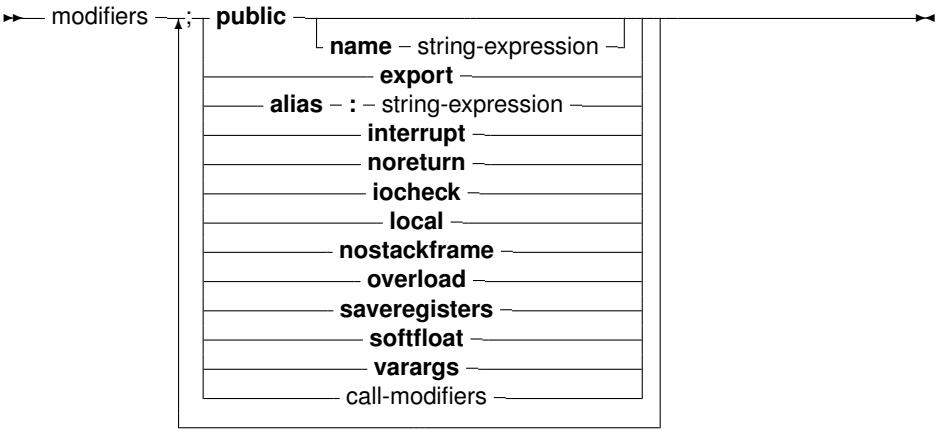


Contrary to Delphi, the assembler keyword must be present to indicate an assembler function. For more information about assembler functions, see the chapter on using assembler in the Programmer's Guide.

14.9 Modifiers

A function or procedure declaration can contain modifiers. Here we list the various possibilities:

Modifiers



The call modifiers can be found in the section on procedural types (section 3.6, page 63). Free Pascal doesn't support all Turbo Pascal modifiers (although it parses them for compatibility), but does support a number of additional modifiers. They are used mainly for assembler and reference to C object files.

14.9.1 alias

The `alias` modifier allows the programmer to specify a different name for a procedure or function. This is mostly useful for referring to this procedure from assembly language constructs or from another object file. As an example, consider the following program:

```
Program Aliases;

Procedure Printit;[public, alias:'DOIT'];
begin
  WriteLn ('In Printit (alias : "DOIT")');
end;
```

```
Procedure myprint; external name 'DOIT';

begin

asm
  call myprint
end;

end.
```

Remark The specified alias is inserted straight into the assembly code, thus it is case sensitive.

The `alias` modifier does not make the symbol public to other modules, unless the routine is also declared in the interface part of a unit, or the `public` modifier is used to force it as public. Consider the following:

```
unit testalias;

interface

procedure testroutine;

implementation

procedure testroutine;alias:'ARoutine';
begin
  WriteLn('Hello world');
end;

end.
```

This will make the routine `testroutine` available publicly to external object files under the label name `ARoutine`.

Remark The `alias` directive is considered deprecated. Please use the `public name` directive. See section 14.9.18, page 243.

14.9.2 cdecl

The `cdecl` modifier can be used to declare a function that uses a C type calling convention. This must be used when accessing functions residing in an object file generated by standard C compilers, but must also be used for Pascal functions that are to be used as callbacks for C libraries.

The `cdecl` modifier allows to use C function in the code. For external C functions, the object file containing the `C` implementation of the function or procedure must be linked in. As an example:

```
program CmodDemo;
{$LINKLIB c}
Const P : PChar = 'This is fun !';
Function StrLen(P: PChar): Longint;cdecl; external name 'strlen';
begin
  WriteLn ('Length of (',p,') : ',StrLen(p));
end.
```

When compiling this, and linking to the C-library, the `strlen` function can be called throughout the program. The `external` directive tells the compiler that the function resides in an external object file or library with the "strlen" name (see 14.7).

Remark The parameters in our declaration of the `C` function should match exactly the ones in the declaration in `C`.

For functions that are not external, but which are declared using `cdecl`, no external linking is needed. These functions have some restrictions, for instance the `array of const` construct can not be used (due the way this uses the stack). On the other hand, the `cdecl` modifier allows these functions to be used as callbacks for routines written in C, as the latter expect the "cdecl" calling convention.

Remark In macpas mode, the `cdecl` modifier can be abbreviated to `c`.

14.9.3 compilerproc

The `compilerproc` modifier is used internally by the compiler. It marks helper routines which you should never call directly, they are used by the compiler to implement routines for type conversions, concatenate string etc.

As a user, you should never use this modifier. You should also never call them directly, the compiler will issue calls to such procedures.

14.9.4 cppdecl

The `cppdecl` modifier can be used to declare a function that uses a C++ type calling convention. This must be used when accessing functions residing in an object file generated by standard C++ compilers, but must also be used for Pascal functions that are to be used as callbacks for C++ libraries.

Note that linking to C++ support is experimental at best, so this should be used with extreme care.

14.9.5 export

The `export` modifier is used to export names when creating a shared library or an executable program. This means that the symbol will be publicly available, and can be imported from other programs. For more information on this modifier, consult the section on "Making libraries" in the Programmer's Guide.

14.9.6 hardfloat

The `hardfloat` modifier is used to indicate that the compiler must use a calling convention that passes certain floating point parameters in VFP registers. (Used on ARM only)

14.9.7 inline

Procedures that are declared `inline` are copied to the places where they are called. This has the effect that there is no actual procedure call, the code of the procedure is just copied to where the procedure is needed, this results in faster execution speed if the function or procedure is used a lot. It is obvious that inlining large functions does not make sense.

By default, `inline` procedures are not allowed. Inline code must be enabled using the command-line switch `-Si` or `{$inline on}` directive.

Remark

1. `inline` is only a hint for the compiler. This does *not* automatically mean that all calls are inlined; sometimes the compiler may decide that a function simply cannot be inlined, or that a particular call to the function cannot be inlined. If so, the compiler will emit a warning.
2. In old versions of Free Pascal, inline code was *not* exported from a unit. This meant that when calling an inline procedure from another unit, a normal procedure call will be performed. Only inside units, `Inline` procedures are really inlined. As of version 2.0.2, inline works across units.
3. Recursive inline functions are not allowed. i. e. an inline function that calls itself is not allowed.
4. You can specify `inline` in the interface of a unit or in the implementation, the result is the same.
5. If you declare an inline function in the interface part of the unit, then the implementation must appear before the first use of this function, or the compiler will not inline it.

14.9.8 interrupt

The `interrupt` keyword is used to declare a routine which will be used as an interrupt handler. On entry to this routine, all the registers will be saved and on exit, all registers will be restored and an interrupt or trap return will be executed (instead of the normal return from subroutine instruction).

On platforms where a return from interrupt does not exist, the normal exit code of routines will be done instead. For more information on the generated code, consult the Programmer's Guide.

14.9.9 iocheck

The `iocheck` keyword is used to declare a routine which causes generation of I/O result checking code within a `{$IOCHECKS ON}` block whenever it is called.

The result is that if a call to this procedure is generated, the compiler will insert I/O checking code if the call is within a `{$IOCHECKS ON}` block.

This modifier is intended for RTL internal routines, not for use in application code.

14.9.10 local

The `local` modifier allows the compiler to optimize the function: a local function cannot be in the interface section of a unit: it is always in the implementation section of the unit. From this it follows that the function cannot be exported from a library.

On Linux, the local directive results in some optimizations. On Windows, it has no effect. It was introduced for Kylix compatibility.

14.9.11 MS_ABI_Default

The `MS_ABI_Default` modifier is used to indicate that the compiler must use the Microsoft version of the x86-64 calling convention. This is the default calling convention Win64, and is only supported on x86-64.

14.9.12 MS_ABI_CDecl

The `MS_ABI_CDecl` modifier is used to indicate that the compiler must use the Microsoft version of the x86-64 calling convention, with the exception that 'array of const' are interpreted as cdecl varargs parameters, instead of the regular pascal array of const.

14.9.13 MWPascal

The `MWPascal` modifier is used to indicate that the compiler must use the Metrowerks Pascal calling convention. This calling convention behaves the same as cdecl, except that const record parameters are passed by reference instead of by value. Supported on all platforms.

14.9.14 noreturn

The `noreturn` modifier can be used to tell the compiler the procedure does not return. This information can used by the compiler to avoid emitting warnings about uninitialized variables or results not being set.

In the following example, the compiler will not emit a warning that the result may not be set in function `f`:

```
procedure do_halt;noreturn;
begin
  halt(1);
end;

function f(i : integer) : integer ;

begin
  if (i<0)  then
    do_halt
  else
    result:=i;
end;
```

14.9.15 nostackframe

The `nostackframe` modifier can be used to tell the compiler it should not generate a stack frame for this procedure or function. By default, a stack frame is always generated for each procedure or function, but the compiler will omit it if it can.

For asm routines (pure assembler) this directive can be used to omit generating a stack frame.

One should be extremely careful when using this modifier: most procedures or functions need a stack frame. Particularly for debugging they are needed.

14.9.16 overload

The `overload` modifier tells the compiler that this function is overloaded. It is mainly for Delphi compatibility, as in Free Pascal, all functions and procedures can be overloaded without this modifier.

There are two cases where the `overload` modifier is mandatory: if a function must be overloaded that resides in another unit. Both functions must be declared with the `overload` modifier: the `overload` modifier tells the compiler that it should continue looking for overloaded versions in other units.

Secondly, if it concerns methods part of a class, if the overloaded methods are not in the same class. Without the `overload` keyword, the compiler will not look for overloaded methods in parent classes.

The following example illustrates the first case. Take the first unit:

```
unit ua;
```

```
interface

procedure DoIt(A : String); overload;

implementation

procedure DoIt(A : String);

begin
  Writeln('ua.DoIt received ',A)
end;

end.
```

And a second unit, which contains an overloaded version:

```
unit ub;

interface

procedure DoIt(A : Integer); overload;

implementation

procedure DoIt(A : integer);

begin
  Writeln('ub.DoIt received ',A)
end;

end.
```

And the following program, which uses both units:

```
program uab;

uses ua,ub;

begin
  DoIt('Some string');
end.
```

When the compiler starts looking for the declaration of `DoIt`, it will find one in the ub unit. Without the `overload` directive, the compiler would give an argument mismatch error:

```
home: >fpc uab.pp
uab.pp(6,21) Error: Incompatible type for arg no. 1:
Got "Constant String", expected "SmallInt"
```

With the `overload` directive in place at both locations, the compiler knows it must continue searching for an overloaded version with matching parameter list. Note that *both* declarations must have the `overload` modifier specified; it is not enough to have the modifier in unit ub. This is to prevent unwanted overloading: the programmer who implemented the `ua` unit must mark the procedure as fit for overloading.

14.9.17 pascal

The `pascal` modifier can be used to declare a function that uses the classic Pascal type calling convention (passing parameters from left to right). For more information on the Pascal calling convention, consult the Programmer's Guide.

14.9.18 public

The `Public` keyword is used to declare a function globally in a unit. This is useful if the function should not be accessible from the unit file (i. e. another unit/program using the unit doesn't see the function), but must be accessible from the object file. As an example:

```
Unit someunit;
interface
Function First : Real;
Implementation
Function First : Real;
begin
  First := 0;
end;
Function Second : Real; [Public];
begin
  Second := 1;
end;
end.
```

If another program or unit uses this unit, it will not be able to use the function `Second`, since it isn't declared in the interface part. However, it will be possible to access the function `Second` at the assembly-language level, by using its mangled name (see the Programmer's Guide).

The `public` modifier can also be followed by a `name` directive to specify the assembler name, as follows:

```
Unit someunit;
interface
Function First : Real;
Implementation
Function First : Real;
begin
  First := 0;
end;
Function Second : Real; Public name 'second';
begin
  Second := 1;
end;
end.
```

The assembler symbol as specified by the "public name" directive will be "second", in all lowercase letters.

14.9.19 register

The `register` keyword is used for compatibility with Delphi. In version 1.0.x of the compiler, this directive has no effect on the generated code. As of the 1.9.X versions, this directive is supported.

For the Intel i386 platform, the first three arguments are passed in registers EAX, EDX and ECX. For other platforms, this keyword has no effect, the default platform ABI calling convention is used. In general, the use of this directive is not recommended, unless you know what you're doing.

14.9.20 safecall

The `safecall` modifier ressembles closely the `stdcall` modifier. It sends parameters from right to left on the stack. Additionally, the called procedure saves and restores all registers.

More information about this modifier can be found in the Programmer's Guide, in the section on the calling mechanism and the chapter on linking.

14.9.21 saveregisters

The `saveregisters` modifier tells the compiler that all CPU registers should be saved prior to calling this routine. Which CPU registers are saved, depends entirely on the CPU.

14.9.22 softfloat

The `softfloat` modifier makes sense only on the ARM architecture. Tells the compiler to se the (platform-specific) calling convention for passing floating point values when software-emulated floating point support is active. (Not supported on all platforms: currently only ARM).

14.9.23 stdcall

The `stdcall` modifier pushes the parameters from right to left on the stack, it also aligns all the parameters to a default alignment.

More information about this modifier can be found in the Programmer's Guide, in the section on the calling mechanism and the chapter on linking.

14.9.24 SYSV_ABI_Default

The `SYS_ABI_Default` modifier is used to indicate that the compiler must use the x86-64-specific calling convention that conforms to the System V AMD64 ABI This is the default calling convention all x86-64 platforms except Win64, and is only supported on x86-64.

14.9.25 SYSV_ABI_CDecl

The `SYSV_ABI_CDecl` modifier is used to indicate that the compiler must use the x86-64-specific calling convention that conforms to the System V AMD64 ABI, with the exception that 'array of const' are interpreted as cdecl varargs parameters, instead of the regular pascal array of const.

14.9.26 VectorCall

The `VectorCall` modifier is used to indicate that the compiler must use vector call calling convention. This is a variant of the `MS_ABI_Default` calling convention that passes certain floating point parameters in vector registers. only supported on on x86-64 CPUs.

14.9.27 varargs

This modifier can only be used together with the `cdecl` modifier, for external C procedures. It indicates that the procedure accepts a variable number of arguments after the last declared variable. These arguments are passed on without any type checking. It is equivalent to using the `array of const` construction for `cdecl` procedures, without having to declare the `array of const`. The square brackets around the variable arguments do not need to be used when this form of declaration is used.

The following declarations are two ways of referring to the same function in the C library:

```
Function PrintF1(fmt : pchar); cdecl; varargs;
                               external 'c' name 'printf';
Function PrintF2(fmt : pchar; Args : Array of const); cdecl;
                               external 'c' name 'printf';
```

But they must be called differently:

```
PrintF1('%d %d\n',1,1);
PrintF2('%d %d\n',[1,1]);
```

14.9.28 winapi

This modifier allows you to specify the native calling conventions for the current platform: the compiler will then select the correct calling convention depending on the OS architecture: `stdcall` on all windows platforms and `cdecl` on all other platforms.

The following example demonstrates this:

```
function libusb_init(var ctx:plibusb_context):integer;winapi;external libusb1;
```

14.10 Unsupported Turbo Pascal modifiers

The modifiers that exist in Turbo Pascal, but aren't supported by Free Pascal, are listed in table (14.1).

Table 14.1: Unsupported modifiers

Modifier	Why not supported ?
Near	Free Pascal is a 32-bit compiler.
Far	Free Pascal is a 32-bit compiler.

The compiler will give a warning when it encounters these modifiers, but will otherwise completely ignore them.

Chapter 15

Operator overloading

15.1 Introduction

Free Pascal supports operator overloading. This means that it is possible to define the action of some operators on self-defined types, and thus allow the use of these types in mathematical expressions.

In addition, operator overloading can be used for some special actions that are strictly speaking not operators in the mathematical sense of the word. This is true for the procedures `Inc` and `Dec` (which are usable as a procedure only) and for implicit and explicit assignment (`:=` and typecasting). The enumerator operator (to define an enumerable series for a type) is also part of this class of operators.

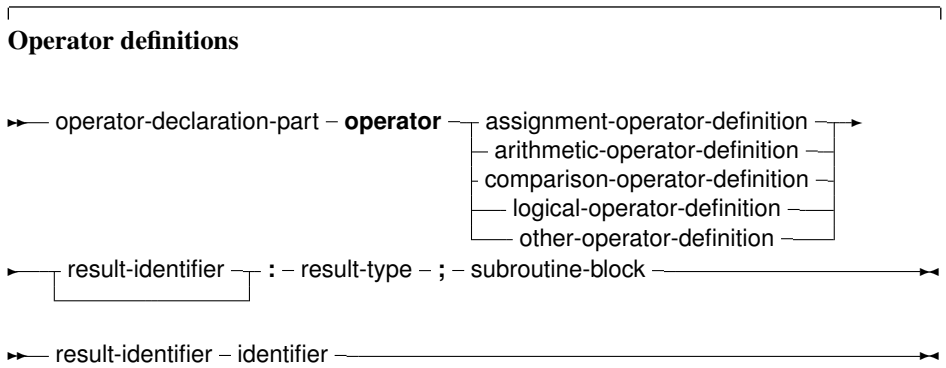
Defining the action of an operator is much like the definition of a function or procedure, only there are some restrictions on the possible definitions, as will be shown in the subsequent.

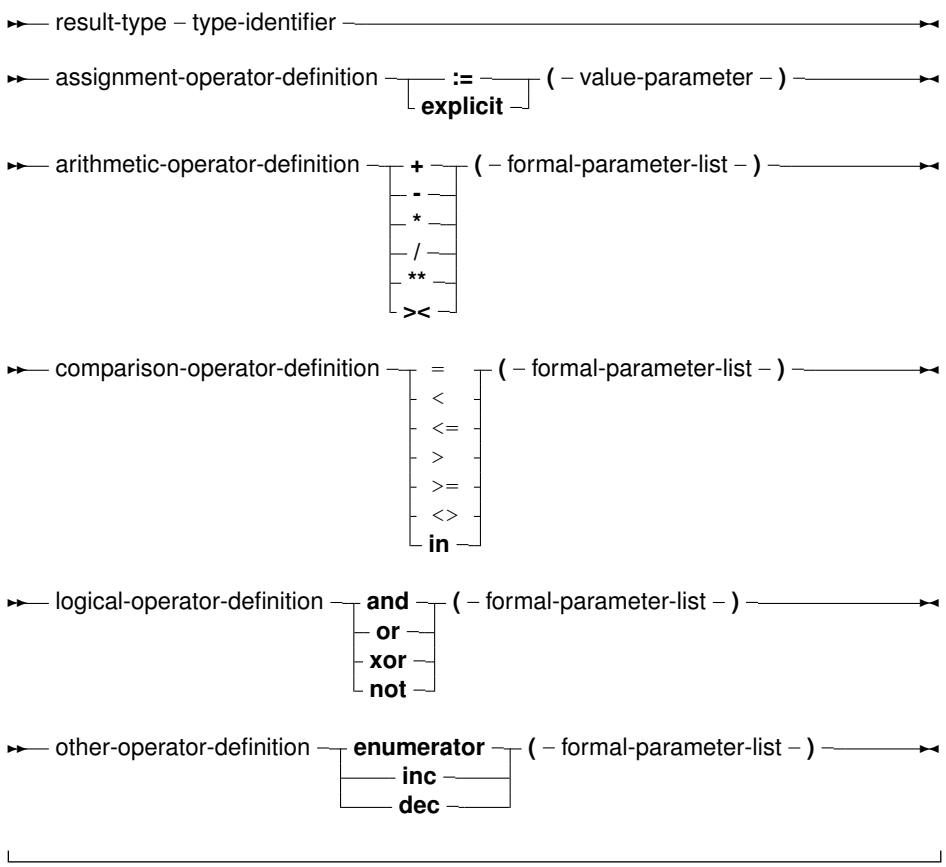
Operator overloading is, in essence, a powerful notational tool; but it is also not more than that, since the same results can be obtained with regular function calls. When using operator overloading, it is important to keep in mind that some implicit rules may produce some unexpected results. This will be indicated.

There are 2 ways to declare operator overloads. The first way is the original way as implemented in Free Pascal, and is the most powerful. It is discussed in this chapter. There is also a second way, which can only be implemented for Advanced (or extended) records, it is discussed in the chapter on extended records.

15.2 Operator declarations

To define the action of an operator is much like defining a function:





The parameter list for a comparison operator or an arithmetic operator must always contain two parameters, with the exception of the unary minus or plus, where only one parameter is needed. The result type of the comparison operator(s) must be `Boolean`.

User-defined simple types can be used for operators, as well as records and arrays. There are some limitations on operator overloading:

- This way of declaring operators is only supported in `ObjFPC` and `FPC` modes.
- It is not possible to define operators on classes.
- The + and - operators cannot be defined on enumerated types.
- When the `ArrayOperators` modeswitch is in effect:

  ```
  {$modeSwitch ArrayOperators}
  ```

  then the + operator cannot be overloaded on dynamic arrays, because it is handled internally by the compiler.

Remark When compiling in `Objfpc` mode, the result identifier may be dropped. The result can then be accessed through the standard `Result` symbol.

If the result identifier is dropped and the compiler is not in one of these modes, a syntax error will occur.

The statement block contains the necessary statements to determine the result of the operation. It can contain arbitrary large pieces of code; it is executed whenever the operation is encountered in some expression. The result of the statement block must always be defined; error conditions are not

checked by the compiler, and the code must take care of all possible cases, throwing a run-time error if some error condition is encountered.

In the following, the three types of operator definitions will be examined. As an example, throughout this chapter the following type will be used to define overloaded operators on:

```
type
  complex = record
    re : real;
    im : real;
  end;
```

This type will be used in all examples.

The sources of the Run-Time Library contain two units that heavily use operator overloading:

ucomplex This unit contains a complete calculus for complex numbers.

matrix This unit contains a complete calculus for matrices.

15.3 Assignment operators

The assignment operator defines the action of a assignment of one type of variable to another. The result type must match the type of the variable at the left of the assignment statement, the single parameter to the assignment operator must have the same type as the expression at the right of the assignment operator.

This system can be used to declare a new type, and define an assignment for that type. For instance, to be able to assign a newly defined type "Complex"

```
Var
  C,Z : Complex; // New type complex

begin
  Z:=C;  // assignments between complex types.
end;
```

The following assignment operator would have to be defined:

```
Operator := (C : Complex) z : complex;
```

To be able to assign a real type to a complex type as follows:

```
var
  R : real;
  C : complex;

begin
  C:=R;
end;
```

the following assignment operator must be defined:

```
Operator := (r : real) z : complex;
```

As can be seen from this statement, it defines the action of the operator := with at the right a real expression, and at the left a complex expression.

An example implementation of this could be as follows:

```
operator := (r : real) z : complex;

begin
  z.re:=r;
  z.im:=0.0;
end;
```

As can be seen in the example, the result identifier (z in this case) is used to store the result of the assignment. When compiling in Delphi mode or ObjFPC mode, the use of the special identifier Result is also allowed, and can be substituted for the z, so the above would be equivalent to

```
operator := (r : real) z : complex;

begin
  Result.re:=r;
  Result.im:=0.0;
end;
```

The assignment operator is also used to convert types from one type to another. The compiler will consider all overloaded assignment operators till it finds one that matches the types of the left hand and right hand expressions. If no such operator is found, a "type mismatch" error is given.

Remark The assignment operator is not commutative; the compiler will never reverse the role of the two arguments. In other words, given the above definition of the assignment operator, the following is *not* possible:

```
var
  R : real;
  C : complex;

begin
  R:=C;
end;
```

If the reverse assignment should be possible then the assignment operator must be defined for that as well. (This is not so for reals and complex numbers.)

Remark The assignment operator is also used in implicit type conversions. This can have unwanted effects. Consider the following definitions:

```
operator := (r : real) z : complex;
function exp(c : complex) : complex;
```

Then the following assignment will give a type mismatch:

```
Var
  r1,r2 : real;

begin
  r1:=exp(r2);
end;
```

The mismatch occurs because the compiler will encounter the definition of the `exp` function with the complex argument. It implicitly converts `r2` to a complex, so it can use the above `exp` function. The result of this function is a complex, which cannot be assigned to `r1`, so the compiler will give a "type mismatch" error. The compiler will not look further for another `exp` which has the correct arguments.

It is possible to avoid this particular problem by specifying

```
  r1:=system.exp(r2);
```

When doing an explicit typecast, the compiler will attempt an implicit conversion if an assignment operator is present. That means that

```
Var
  R1 : T1;
  R2 : T2;

begin
  R2:=T2(R1);
```

Will be handled by an operator

```
Operator := (aRight: T1) Res: T2;
```

However, an `Explicit` operator can be defined, and then it will be used instead when the compiler encounters a typecast.

The reverse is not true: In a regular assignment, the compiler will not consider explicit assignment operators.

Given the following definitions:

```
uses
  sysutils;

type
  TTest1 = record
    f: LongInt;
  end;
  TTest2 = record
    f: String;
  end;
  TTest3 = record
    f: Boolean;
  end;
```

It is possible to create assignment operators:

```
operator := (aRight: TTest1) Res: TTest2;
begin
  Writeln('Implicit TTest1 => TTest2');
  Res.f := IntToStr(aRight.f);
end;

operator := (aRight: TTest1) Res: TTest3;
begin
```

```
  Writeln('Implicit TTest1 => TTest3');
  Res.f := aRight.f <> 0;
end;
```

But one can also define typecasting operators:

```
operator Explicit(aRight: TTest2) Res: TTest1;
begin
  Writeln('Explicit TTest2 => TTest1');
  Res.f := StrToIntDef(aRight.f, 0);
end;

operator Explicit(aRight: TTest1) Res: TTest3;
begin
  Writeln('Explicit TTest1 => TTest3');
  Res.f := aRight.f <> 0;
end;
```

Thus, the following code

```
var
  t1: TTest1;
  t2: TTest2;
  t3: TTest3;
begin
  t1.f := 42;
  // Implicit
  t2 := t1;
  // theoretically explicit, but implicit op will be used,
  // because no explicit operator is defined
  t2 := TTest2(t1);
  // the following would not compile,
  // no assignment operator defined (explicit one won't be used here)
  //t1 := t2;
  // Explicit
  t1 := TTest1(t2);
  // first explicit (TTest2 => TTest1) then implicit (TTest1 => TTest3)
  t3 := TTest1(t2);
  // Implicit
  t3 := t1;
  // explicit
  t3 := TTest3(t1);
end.
```

will produce the following output:

```
Implicit TTest1 => TTest2
Implicit TTest1 => TTest2
Explicit TTest2 => TTest1
Explicit TTest2 => TTest1
Implicit TTest1 => TTest3
Implicit TTest1 => TTest3
Explicit TTest1 => TTest3
```

15.4 Arithmetic operators

Arithmetic operators define the action of a binary operator. Possible operations are:

multiplication To multiply two types, the * multiplication operator must be overloaded.

division To divide two types, the / division operator must be overloaded.

addition To add two types, the + addition operator must be overloaded. Note that the + can not be used for dynamic arrays, as this is an internal compiler operation for array concatenation (see also the modeswitch ARRAYOPERATORS).

subtraction To subtract two types, the − subtraction operator must be overloaded.

exponentiation To exponentiate two types, the ** exponentiation operator must be overloaded.

Unary minus is used to take the negative of the argument following it.

Symmetric Difference To take the symmetric difference of 2 structures, the >< operator must be overloaded.

The definition of an arithmetic operator takes two parameters, except for unary minus, which needs only one parameter. The first parameter must be of the type that occurs at the left of the operator, the second parameter must be of the type that is at the right of the arithmetic operator. The result type must match the type that results after the arithmetic operation.

To compile an expression as

```
var
  R : real;
  C,Z : complex;

begin
  C:=R*Z;
end;
```

One needs a definition of the multiplication operator as:

```
Operator * (r : real; z1 : complex) z : complex;

begin
  z.re := z1.re * r;
  z.im := z1.im * r;
end;
```

As can be seen, the first operator is a real, and the second is a complex. The result type is complex.

Multiplication and addition of reals and complexes are commutative operations. The compiler, however, has no notion of this fact so even if a multiplication between a real and a complex is defined, the compiler will not use that definition when it encounters a complex and a real (in that order). It is necessary to define both operations.

So, given the above definition of the multiplication, the compiler will not accept the following statement:

```
var
  R : real;
  C,Z : complex;
```

```
begin
  C:=Z*R;
end;
```

Since the types of `Z` and `R` don't match the types in the operator definition.

The reason for this behavior is that it is possible that a multiplication is not always commutative. E. g. the multiplication of a `(n,m)` with a `(m,n)` matrix will result in a `(n,n)` matrix, while the multiplication of a `(m,n)` with a `(n,m)` matrix is a `(m,m)` matrix, which needn't be the same in all cases.

15.5 Comparison operator

The comparison operator can be overloaded to compare two different types or to compare two equal types that are not basic types. If the operands are not simple types, the result type of a comparison operator need not always be a boolean, but then they cannot be used in an `if`, `repeat` or `while` statement.

The comparison operators that can be overloaded are:

equal to (=) To determine if two variables are equal.

unequal to (<>) To determine if two variables are different.

less than (<) To determine if one variable is less than another.

greater than (>) To determine if one variable is greater than another.

greater than or equal to (>=) To determine if one variable is greater than or equal to another.

less than or equal to (<=) To determine if one variable is greater than or equal to another.

If there is no separate operator for *unequal to* (<>), then, to evaluate a statement that contains the *unequal to* operator, the compiler uses the *equal to* operator (=), and negates the result. The opposite is not true: if no "equal to" but an "unequal to" operator exists, the compiler will not use it to evaluate an expression containing the equal (=) operator.

As an example, the following operator allows to compare two complex numbers:

```
operator = (z1, z2 : complex) b : boolean;
```

the above definition allows comparisons of the following form:

```
Var
  C1,C2 : Complex;

begin
  If C1=C2 then
    Writeln('C1 and C2 are equal');
end;
```

The comparison operator definition needs two parameters, with the types that the operator is meant to compare. Here also, the compiler doesn't apply commutativity: if the two types are different, then it is necessary to define two comparison operators.

In the case of complex numbers, it is, for instance necessary to define 2 comparisons: one with the complex type first, and one with the real type first.

Given the definitions

```
operator = (z1 : complex;r : real) b : boolean;
operator = (r : real; z1 : complex) b : boolean;
```

the following two comparisons are possible:

```
Var
  R,S : Real;
  C : Complex;

begin
  If (C=R) or (S=C) then
   Writeln ('Ok');
end;
```

Note that the order of the real and complex type in the two comparisons is reversed.

The following example shows that the result type does not need to be a boolean:

```
Type
  TMyRec = record a,b : integer; end;

operator = (x,y : TMyRec) r : string;

begin
  if (x.a=y.a) and (x.b=y.b) then
    R:='equal'
  else
    R:='differ';
end;

var
  x,y : TMyRec;

begin
  x.a:=1;
  y.a:=1;
  Writeln(x=y);
  x.a:=2;
  y.a:=3;
  Writeln(x=y);
end.
```

When executed, this example will print

```
equal
differ
```

obviously, a statement as

```
if (x=y) then
  writeln('Equal');
```

Will not compile, since the if statement needs a boolean check:

```
Error: Incompatible types: got "ShortString" expected "Boolean"
```

15.6 In operator

As of version 2.6 of Free pascal, the `In` operator can be overloaded as well. The first argument of the in operator must be the operand on the left of the `in` keyword. The following overloads the in operator for records:

```
{$mode objfpc}{$H+}

type
  TMyRec = record A: Integer end;

operator in (const A: Integer; const B: TMyRec): boolean;
begin
  Result := A = B.A;
end;

var
  R: TMyRec;
begin
  R.A := 10;
  Writeln(1 in R); // false
  Writeln(10 in R); // true
end.
```

The `in` operator can also be overloaded for other types than ordinal types, as in the following example:

```
{$mode objfpc}{$H+}

type
  TMyRec = record A: Integer end;

operator in (const A: TMyRec; const B: TMyRec): boolean;
begin
  Result := A.A = B.A;
end;

var
  S,R: TMyRec;
begin
  R.A := 10;
  S.A:=1;
  Writeln(S in R); // false
  Writeln(R in R); // true
end.
```

15.7 Logical operators

Logical operators `and`, `or`, `xor` and `not` can be overloaded. These operators are normally used on simple types in two different ways:

1. As boolean operators, in which case the result is a boolean (except for `not`)

2. As bitwise operators, in which case the result is an ordinal type.

When overloading these operators, the result type is not restricted. This means you can define the operators as boolean logic operators:

```
Type
  Rec = record
    a,b : Boolean;
  end;

Operator and (r1,r2 : Rec) z : boolean;

begin
  z:=(R1.a and R2.a) or (R1.b and r2.b);
end;

Operator or (r1,r2 : Rec) z : Boolean;

begin
  z:=(R1.a or R2.a) and (R1.b or r2.b)
end;

Operator xor (r1,r2 : Rec) z : Boolean;

begin
  z:=(R1.a xor R2.a) and (R1.b xor r2.b)
end;

Operator not (r1 : Rec) z : rec;

begin
  z.A:=not R1.a;
  z.B:=not R1.b;
end;


var
r1,r2 : Rec;

begin
  Writeln(r1 and r2);
  Writeln(r1 or r2);
  Writeln(r1 xor r2);
  Writeln((not r1).a);
end.
```

But it is also possible to have different return types:

```
Operator and (r1,r2 : Rec) z : string;

begin
  Str(Ord((R1.a and R2.a) or (R1.b and r2.b)),Z);
end;
```

The compiler will always check the return type to determine the final type of an expression, and assignments will be checked for type safety.

15.8 Inc/Dec operators

The `Inc` and `Dec` procedures are – for optimization purposes – compiler intrinsics. They can be overridden in the usual manner by declaring a procedure:

```
procedure Inc(var s : String);

begin
  Inc(S,'.');
end;

procedure Inc(var s : String; T : String);

begin
 S:=S+T;
end;
```

But because they are compiler intrinsics, they can also be implemented as operators for custom types:

```
operator Inc (s,T : String) : z : string;

begin
  Z:=S+T;
end;
```

15.9 The enumerator operator

The enumerator operator can be used to define an enumerator for any type. It must return a class, object, or extended record that has the same signature as the IEnumerator interface. Note that this operator is not recognized by the compiler under the Delphi syntax mode.

The following code will define an enumerator for the Integer type:

```
Type
  TEvenEnumerator = Class
    FCurrent : Integer;
    FMax : Integer;
    Function MoveNext : Boolean;
    Property Current : Integer Read FCurrent;
  end;

Function TEvenEnumerator.MoveNext : Boolean;

begin
  FCurrent:=FCurrent+2;
  Result:=FCurrent<=FMax;
end;
```

```
operator enumerator(i : integer) : TEvenEnumerator;

begin
  Result:=TEvenEnumerator.Create;
  Result.FMax:=i;
end;

var
  I : Integer;
  m : Integer = 4;

begin
  For I in M do
    Writeln(i);
end.
```

The loop will print all nonzero even numbers smaller or equal to the enumerable. (2 and 4 in the case of the example).

More about enumerators and the enumerator operator can be found in the section 13.2.5, page 206.

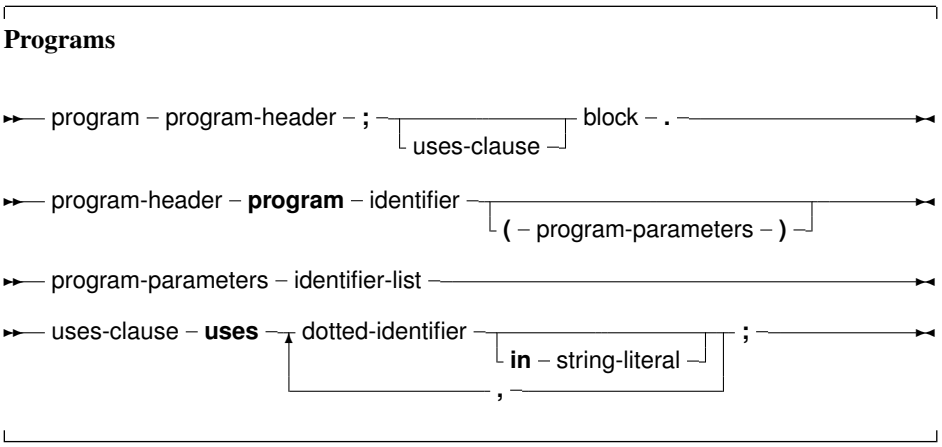
Chapter 16

Programs, units, blocks

A Pascal program can consist of modules called `units`. A unit can be used to group pieces of code together, or to give someone code without giving the sources. Both programs and units consist of code blocks, which are mixtures of statements, procedures, and variable or type declarations.

16.1 Programs

A Pascal program consists of the program header, followed possibly by a "uses" clause, and a block.



The program header is not required. If present, it will be used by the compiler to name the program. If absent, the source filename will be used.

The uses clause serves to identify all units that are needed by the program. All identifiers which are declared in the interface section of the units in the uses clause are added to the known identifiers of the program. The system unit doesn't have to be in this list, since it is always loaded by the compiler.

The order in which the units appear is significant, it determines in which order they are initialized. Units are initialized in the same order as they appear in the uses clause. Identifiers are searched in the opposite order, i. e. when the compiler searches for an identifier, then it looks first in the last unit in the uses clause, then the last but one, and so on. This is important in case two units declare different types with the same identifier.

The compiler will look for compiled versions or source versions of all units in the uses clause in the unit search path. If the unit filename was explicitly mentioned using the `in` keyword, the source is taken from the filename specified:

```
program programb;

uses unita in '..\unita.pp';
```

unita is searched in the parent directory of the current working directory of the compiler. You can add a `{$UNITPATH ..}` directive to make sure that the unit will be found no matter where the compiler's current working directory is.

When the compiler looks for unit files, it adds the extension .ppu to the name of the unit. On LINUX and in operating systems where filenames are case sensitive when looking for a unit, the following mechanism is used:

1. The unit is first looked for in the original case.
2. The unit is looked for in all-lowercase letters.
3. The unit is looked for in all-uppercase letters.

Additionally, If a unit name is longer than 8 characters, the compiler will first look for a unit name with this length, and then it will truncate the name to 8 characters and look for it again. For compatibility reasons, this is also true on platforms that support long file names.

Note that the above search is performed in each directory in the search path.

The program block contains the statements that will be executed when the program is started. Note that these statements need not necessarily be the first statements that are executed: the initialization code of the units may also contain statements that are executed prior to the program code.

The structure of a program block is discussed below.

A note on the program identifier:

- It can be used to refer to a variable, in order to distinguish between a local and a global variable:

```
program mytest;

var
  x: integer = 1; // global

procedure test;

var
  x: integer = 9; // local
begin
  writeln(x); // 9
  writeln(mytest.x); // 1
end;

begin
  test;
end.
```

- When linking in resources using a wildcard, the filename of the program is used to search the resource file. The name will not be used as the name of the resource file. So in a file called `my.pas` the following code

```
program mytest;
```

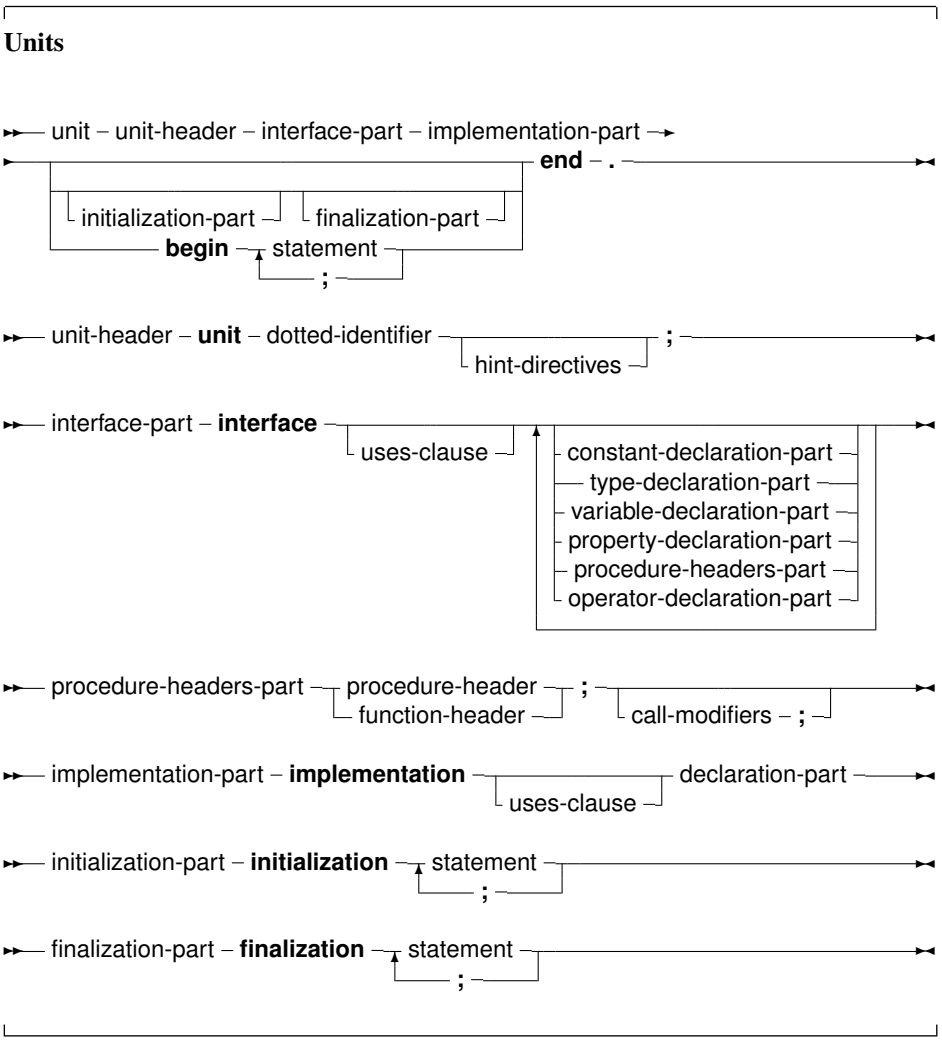
```
{$R *.res}

begin
end;
```

will link in file `my.res`

16.2 Units

A unit contains a set of declarations, procedures and functions that can be used by a program or another unit. The syntax for a unit is as follows:



As can be seen from the syntax diagram, a unit always consists of a interface and an implementation part. Optionally, there is an initialization block and a finalization block, containing code that will be executed when the program is started, and when the program stops, respectively.

The unit header can be accompagnied by hint directives (section 1.5, page 17):

```
unit old deprecated;
```

Both the interface part or implementation part can be empty, but the keywords `Interface` and `implementation` must be specified. With one exception: in macpas mode, the `implementation` keyword may be omitted if the implementation is empty.. . The following is a completely valid unit;

```
unit a;

interface

implementation

end.
```

The interface part declares all identifiers that must be exported from the unit. This can be constant, type or variable identifiers, and also procedure or function identifier declarations. The interface part cannot contain code that is executed: only declarations are allowed. The following is a valid interface part:

```
unit a;

interface

uses b;

Function MyFunction : SomeBType;

Implementation
```

The type `SomeBType` is defined in unit `b`.

All functions and methods that are declared in the interface part must be implemented in the implementation part of the unit, except for declarations of external functions or procedures. If a declared method or function is not implemented in the implementation part, the compiler will give an error, for example the following:

```
unit unita;

interface

Function MyFunction : Integer;

implementation

end.
```

Will result in the following error:

```
unita.pp(5,10) Error: Forward declaration not solved "MyFunction:SmallInt;"
```

The implementation part is primarily intended for the implementation of the functions and procedures declared in the interface part. However, it can also contain declarations of its own: the declarations inside the implementation part are *not* accessible outside the unit.

The initialization and finalization part of a unit are optional.

The initialization block is used to initialize certain variables or execute code that is necessary for the correct functioning of the unit. The initialization parts of the units are executed in the order that the

compiler loaded the units when compiling a program. They are executed before the first statement of the program is executed.

The finalization part of the units are executed in the reverse order of the initialization execution. They are used for instance to clean up any resources allocated in the initialization part of the unit, or during the lifetime of the program. The finalization part is always executed in the case of a normal program termination: whether it is because the final `end` is reached in the program code or because a `Halt` instruction was executed somewhere.

In case the program stops during the execution of the initialization blocks of one of the units, only the units that were already initialized will be finalized. Note that in difference with Delphi, in Free Pascal a `finalization` block can be present without an `Initialization` block. That means the following will compile in Free Pascal, but not in Delphi.

```
Finalization
  CleanupUnit;
end.
```

An initialization section by itself (i. e. without finalization) may simply be replaced by a statement block. That is, the following:

```
Initialization
  InitializeUnit;
end.
```

is completely equivalent to

```
Begin
  InitializeUnit;
end.
```

16.3 Namespaces: Dotted Units

As can be seen in the syntax diagram for a unit, a unit name can contain dots. This means that the units can be organized in namespaces.

So, the following is a correct unit declaration:

```
unit a.b;

interface

Function C : integer;

implementation

Function C : integer;

begin
  Result:=1;
end;

end.
```

The unit can be used as follows:

```
program d;

uses a.b;

begin
  Writeln(c);
end.
```

When resolving symbols, unit scopes always take precedence over symbols inside units.
Given the following units:

```
unit myunit;

interface

var
  test: record
    a: longint;
  end;

implementation

initialization
  test.a:=2;
end.
```

and

```
unit myunit.test;

interface

var
  a: longint;

implementation

initialization
  a:=1;
end.
```

The following program will resolve `myunit.test.a` to the variable `a` in unit `myunit.test`:

```
uses
   myunit, myunit.test;

begin
  Writeln('myunit.test.a : ',myunit.test.a);
end.
```

So it will print:

```
myunit.test.a : 1
```

Reversing the order of the units will not change this:

```
uses
   myunit.test, myunit;

begin
  Writeln('myunit.test.a : ',myunit.test.a);
end.
```

will also print

```
myunit.test.a : 1
```

Similarly, the following program will resolve `myunit.test.a` to the variable `a` in unit `myunit.test`:

```
uses
   myunit.test, myunit;

begin
  Writeln('a : ',a);
end.
```

it will print:

```
a : 1
```

Similarly, the following program will resolve `test.a` to the variable `test.a` in unit `myunit`:

```
uses
   myunit.test, myunit;

begin
  Writeln('test.a : ',test.a);
end.
```

will print

```
test.a : 2
```

16.4 Unit dependencies

When a program uses a unit (say unitA) and this units uses a second unit, say unitB, then the program depends indirectly also on `unitB`. This means that the compiler must have access to unitB when trying to compile the program. If the unit is not present at compile time, an error occurs.

Note that the identifiers from a unit on which a program depends indirectly, are not accessible to the program. To have access to the identifiers of a unit, the unit must be in the uses clause of the program or unit where the identifiers are needed.

Units can be mutually dependent, that is, they can reference each other in their uses clauses. This is allowed, on the condition that at least one of the references is in the implementation section of the unit. This also holds for indirect mutually dependent units.

If it is possible to start from one interface uses clause of a unit, and to return there via uses clauses of interfaces only, then there is circular unit dependence, and the compiler will generate an error. For example, the following is not allowed:

```
Unit UnitA;
interface
Uses UnitB;
implementation
end.

Unit UnitB
interface
Uses UnitA;
implementation
end.
```

But this is allowed:

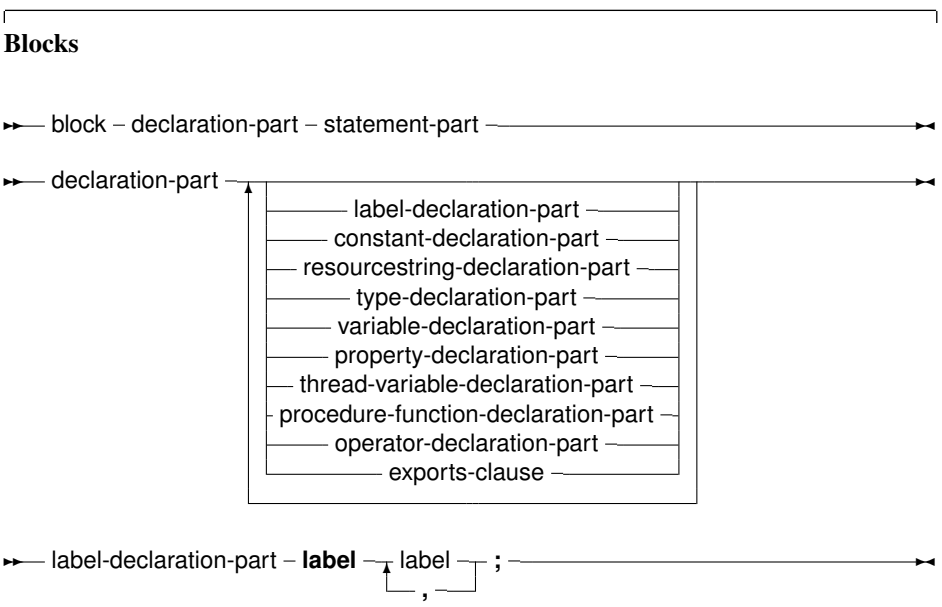
```
Unit UnitA;
interface
Uses UnitB;
implementation
end.
Unit UnitB
implementation
Uses UnitA;
end.
```

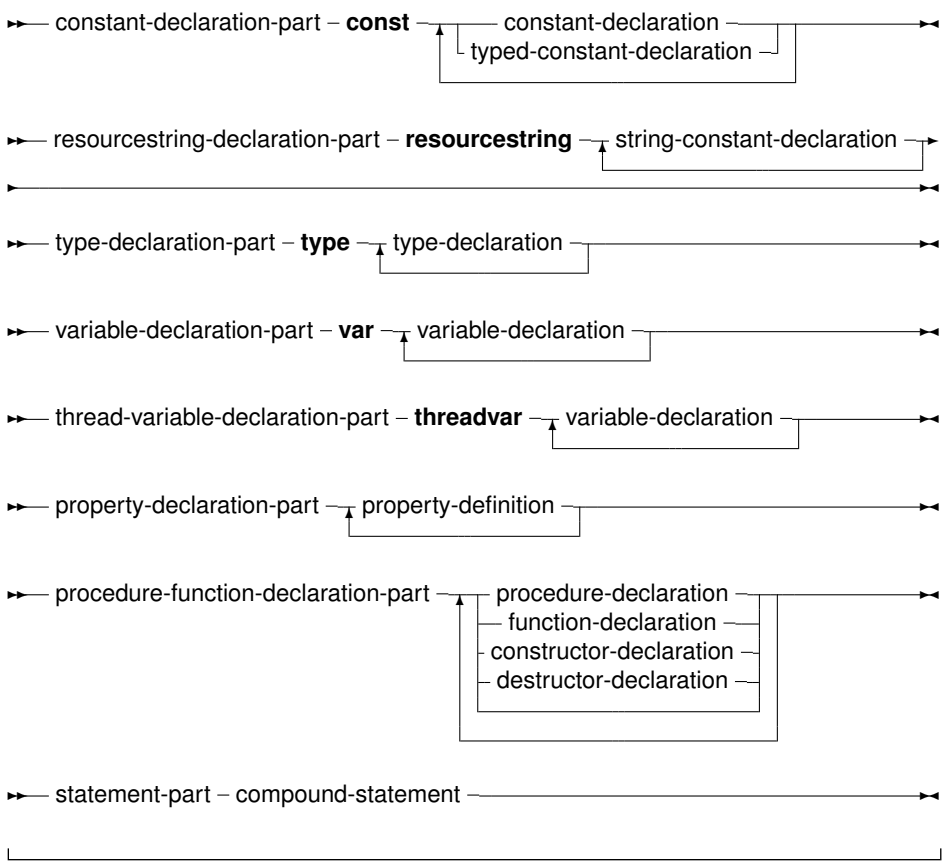
Because UnitB uses UnitA only in its implementation section.

In general, it is a bad idea to have unit interdependencies, even if it is only in implementation sections.

16.5 Blocks

Units and programs are made of blocks. A block is made of declarations of labels, constants, types, variables and functions or procedures. Blocks can be nested in certain ways, i. e., a procedure or function declaration can have blocks in themselves. A block looks like the following:





Labels that can be used to identify statements in a block are declared in the label declaration part of that block. Each label can only identify one statement.

Constants that are to be used only in one block should be declared in that block's constant declaration part.

Variables that are to be used only in one block should be declared in that block's variable declaration part.

Types that are to be used only in one block should be declared in that block's type declaration part.

Lastly, functions and procedures that will be used in that block can be declared in the procedure/function declaration part.

These four declaration parts can be intermixed, there is no required order other than that you cannot use (or refer to) identifiers that have not yet been declared.

After the different declaration parts comes the statement part. This contains any actions that the block should execute. All identifiers declared before the statement part can be used in that statement part.

16.6 Scope

Identifiers are valid from the point of their declaration until the end of the block in which the declaration occurred. The range where the identifier is known is the *scope* of the identifier. The exact scope of an identifier depends on the way it was defined.

16.6.1 Block scope

The *scope* of a variable declared in the declaration part of a block, is valid from the point of declaration until the end of the block. If a block contains a second block, in which the identifier is redeclared, then inside this block, the second declaration will be valid. Upon leaving the inner block, the first declaration is valid again. Consider the following example:

```
Program Demo;
Var X : Real;
{ X is real variable }
Procedure NewDeclaration
Var X : Integer;  { Redeclare X as integer}
begin
 // X := 1.234; {would give an error when trying to compile}
 X := 10; { Correct assignment}
end;
{ From here on, X is Real again}
begin
 X := 2.468;
end.
```

In this example, inside the procedure, `X` denotes an integer variable. It has its own storage space, independent of the variable `X` outside the procedure.

16.6.2 Record scope

The field identifiers inside a record definition are valid in the following places:

1. To the end of the record definition.
2. Field designators of a variable of the given record type.
3. Identifiers inside a `With` statement that operates on a variable of the given record type.

16.6.3 Class scope

A component identifier (one of the items in the class' component list) is valid in the following places:

1. From the point of declaration to the end of the class definition.
2. In all descendent types of this class, unless it is in the private part of the class declaration.
3. In all method declaration blocks of this class and descendent classes.
4. In a `With` statement that operators on a variable of the given class's definition.

Note that method designators are also considered identifiers.

16.6.4 Unit scope

All identifiers in the interface part of a unit are valid from the point of declaration, until the end of the unit. Furthermore, the identifiers are known in programs or units that have the unit in their uses clause.

Identifiers from indirectly dependent units are *not* available. Identifiers declared in the implementation part of a unit are valid from the point of declaration to the end of the unit.

The system unit is automatically used in all units and programs. Its identifiers are therefore always known, in each Pascal program, library or unit.

The rules of unit scope imply that an identifier of a unit can be redefined. To have access to an identifier of another unit that was redeclared in the current unit, precede it with that other units name, as in the following example:

```
unit unitA;
interface
Type
  MyType = Real;
implementation
end.
Program prog;
Uses UnitA;

{ Redeclaration of MyType}
Type MyType = Integer;
Var A : Mytype;      { Will be Integer }
    B : UnitA.MyType { Will be real }
begin
end.
```

This is especially useful when redeclaring the system unit's identifiers.

Note that when you use 2 units declare the same symbol, the definition of the symbol in the last unit in the uses clause is used, i.e. the compiler searches in used units from last to first. Thus, in the following set of units:

```
unit unitA;

interface

Type
  MyType = Real;

implementation
end.

unit unitB;

interface

Type
  MyType = integer;

implementation
end.

unit unitC;

interface

uses unitA, unitB;
```

```
var
  MyVar : MyType;

implementation

end.
```

`MyVar` will be of type `integer`.

The compiler will always search in the current unit first before searching in the uses clauses. Thus, in the following units, the type of `MyVar` is again `integer`.

```
unit unitA;

interface

Type
  MyType = Real;

implementation

end.

unit unitB;

interface

Type
  MyType = integer;

implementation

uses unitA;

var
  MyVar : MyType;
end.
```

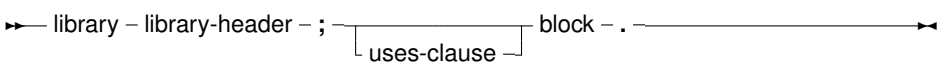
The compiler will first search for `MyType` in the current unit, and if it is not found there, it will look in the uses clauses.

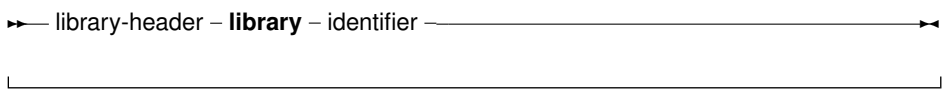
16.7 Libraries

Free Pascal supports making of dynamic libraries (DLLs under Win32 and OS/2) through the use of the `Library` keyword.

A Library is just like a unit or a program:

Libraries

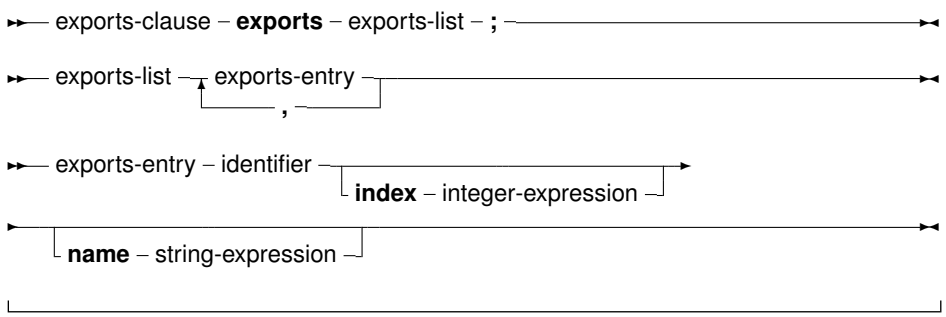




By default, functions and procedures that are declared and implemented in library are not available to a programmer that wishes to use this library.

In order to make functions or procedures available from the library, they must be exported in an exports clause:

Exports clause



Under Win32, an index clause can be added to an exports entry. An index entry must be a positive number larger or equal than 1, and less than `MaxInt`.

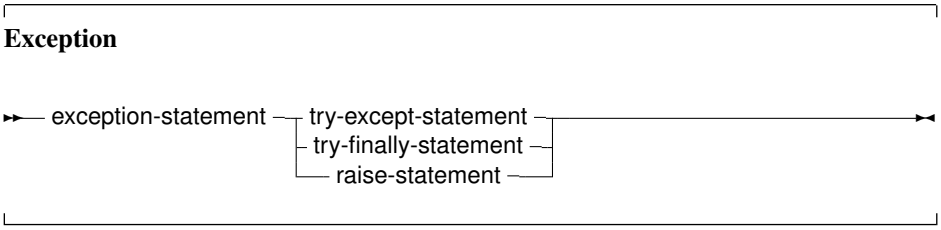
Optionally, an exports entry can have a name specifier. If present, the name specifier gives the exact name (case sensitive) by which the function will be exported from the library.

If neither of these constructs is present, the functions or procedures are exported with the exact names as specified in the exports clause.

Chapter 17

Exceptions

Exceptions provide a convenient way to program error and error-recovery mechanisms, and are closely related to classes. Exception support is based on three constructs:



Raise statement To raise an exception. This is usually done to signal an error condition. It is however also usable to abort execution and immediately return to a well-known point in the executable.

Try ... Except statement This is a structured statement. This block of statements serve to catch exceptions raised within the scope of the block, and to provide exception-recovery code.

Try ... Finally statement Is also a structured statement. This block of statements force code to be executed irrespective of an exception occurrence or not. They generally serve to clean up memory or close files in case an exception occurs. The compiler generates many implicit `Try ... Finally` blocks around procedure, to force memory consistency.

Remark Because exceptions use classes, you need one of the object pascal modes to be able to use them:

```
{$MODE OBJFPC}
```

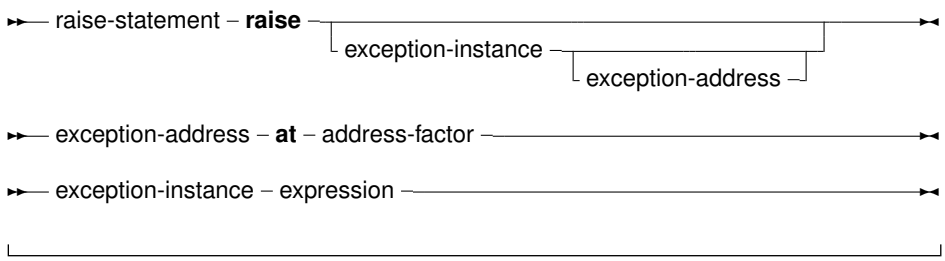
Or

```
{$MODE DELPHI}
```

17.1 The raise statement

The `raise` statement is as follows:





This statement will raise an exception. If it is specified, the exception instance must be an initialized instance of any class, which is the raise type. The exception address and frame are optional. If they are not specified, the compiler will provide the address by itself. If the exception instance is omitted, then the current exception is re-raised. This construct can only be used in an exception handling block (see further).

Remark Control *never* returns after an exception block. The control is transferred to the first `try...finally` or `try...except` statement that is encountered when unwinding the stack. If no such statement is found, the Free Pascal Run-Time Library will generate a run-time error 217 (see also section 17.5, page 276). The exception address will be printed by the default exception handling routines.

As an example: The following division checks whether the denominator is zero, and if so, raises an exception of type `EDivException`

```
Type EDivException = Class(Exception);
Function DoDiv (X,Y : Longint) : Integer;
begin
  If Y=0 then
    Raise EDivException.Create ('Division by Zero would occur');
  Result := X Div Y;
end;
```

The class `Exception` (section 17.5, page 276) is defined in the SysUtils unit of the RTL (see Unit Reference).

Remark Although the `Exception` class is used as the base class for exceptions throughout the code, this is just an unwritten agreement: the class can be of any type, and need not be a descendent of the `Exception` class.

Of course, most code depends on the unwritten agreement that an exception class descends from `Exception`.

The following code shows how to omit an error reporting routine from the stack shown in the exception handler:

```
{$mode objfpc}
uses sysutils;

procedure error(Const msg : string);

begin
  raise exception.create(Msg) at
    get_caller_addr(get_frame),
    get_caller_frame(get_frame);
end;

procedure test2;
```

```
begin
  error('Error');
end;

begin
  test2;
end.
```

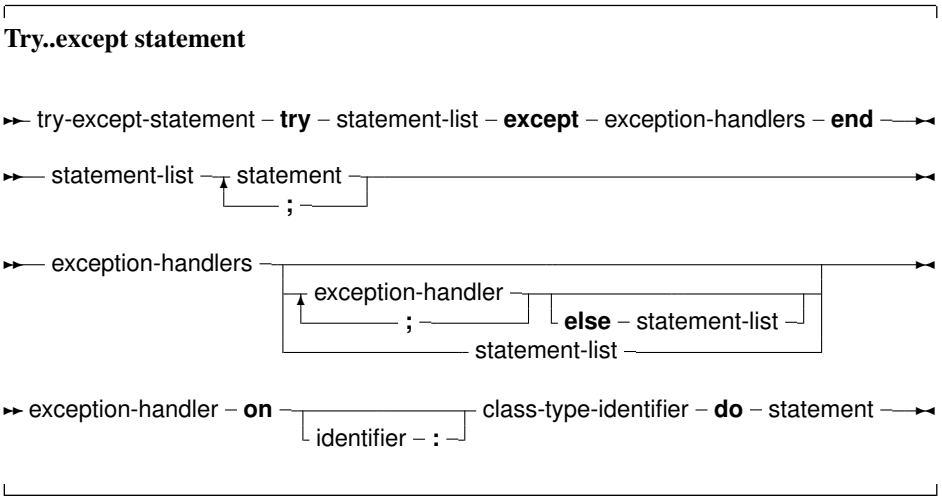
The program, when run, will show a backtrace as follows:

```
An unhandled exception occurred at $00000000004002D3 :
Exception : Error
  $00000000004002D3 line 15 of testme.pp
  $00000000004002E6 line 19 of testme.pp
```

Line 15 is in procedure `Test2`, not in `Error`, which actually raised the exception.

17.2 The try...except statement

A `try...except` exception handling block is of the following form:



If no exception is raised during the execution of the `statement list`, then all statements in the list will be executed sequentially, and the except block will be skipped, transferring program flow to the statement after the final `end`.

If an exception occurs during the execution of the `statement list`, the program flow will be transferred to the except block. Statements in the statement list between the place where the exception was raised and the exception block are ignored.

In the exception handling block, the type of the exception is checked, and if there is an exception handler where the class type matches the exception object type, or is a parent type of the exception object type, then the statement following the corresponding `Do` will be executed. The first matching type is used. After the `Do` block was executed, the program continues after the `End` statement.

The identifier in an exception handling statement is optional, and declares an exception object. It can be used to manipulate the exception object in the exception handling code. The scope of this declaration is the statement block following the `Do` keyword.

If none of the `On` handlers matches the exception object type, then the statement list after `else` is executed. If no such list is found, then the exception is automatically re-raised. This process allows to nest `try...except` blocks.

If, on the other hand, the exception was caught, then the exception object is destroyed at the end of the exception handling block, before program flow continues. The exception is destroyed through a call to the object's `Destroy` destructor.

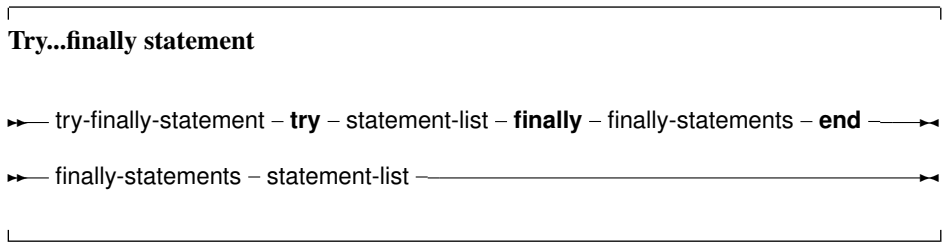
As an example, given the previous declaration of the `DoDiv` function, consider the following

```
Try
  Z := DoDiv (X,Y);
Except
  On EDivException do Z := 0;
end;
```

If `Y` happens to be zero, then the DoDiv function code will raise an exception. When this happens, program flow is transferred to the except statement, where the Exception handler will set the value of `Z` to zero. If no exception is raised, then program flow continues past the last `end` statement. To allow error recovery, the `Try ... Finally` block is supported. A `Try...Finally` block ensures that the statements following the `Finally` keyword are guaranteed to be executed, even if an exception occurs.

17.3 The try...finally statement

A `Try..Finally` statement has the following form:



If no exception occurs inside the `statement List`, then the program runs as if the `Try`, `Finally` and `End` keywords were not present, unless an `exit` command is given: an exit command first executes all statements in the finally blocks before actually exiting.

If, however, an exception occurs, the program flow is immediately transferred from the point where the exception was raised to the first statement of the `Finally statements`.

All statements after the finally keyword will be executed, and then the exception will be automatically re-raised. Any statements between the place where the exception was raised and the first statement of the `Finally Statements` are skipped.

As an example consider the following routine:

```
Procedure Doit (Name : string);
Var F : Text;
begin
  Assign (F,Name);
  Rewrite (name);
  Try
    ... File handling ...
```

```
  Finally
    Close(F);
  end;
end;
```

If during the execution of the file handling an exception occurs, then program flow will continue at the `close(F)` statement, skipping any file operations that might follow between the place where the exception was raised, and the `Close` statement. If no exception occurred, all file operations will be executed, and the file will be closed at the end.

Note that an `Exit` statement enclosed by a `try .. finally` block, will still execute the finally block. Reusing the previous example:

```
Procedure Doit (Name : string);
Var
  F : Text;
  B : Boolean;
begin
  B:=False;
  Assign (F,Name);
  Rewrite (name);
  Try
    // ... File handling ...
    if B then
      exit; // Stop processing prematurely
    // More file handling
  Finally
    Close(F);
  end;
end;
```

The file will still be closed, even if the processing ends prematurely using the `Exit` statement.

17.4 Exception handling nesting

It is possible to nest `Try...Except` blocks with `Try...Finally` blocks. Program flow will be done according to a `lifo` (last in, first out) principle: The code of the last encountered `Try...Except` or `Try...Finally` block will be executed first. If the exception is not caught, or it was a finally statement, program flow will be transferred to the last-but-one block, *ad infinitum*.

If an exception occurs, and there is no exception handler present which handles this exception, then a run-time error 217 will be generated. When using the SysUtils unit, a default handler is installed which will show the exception object message, and the address where the exception occurred, after which the program will exit with a `Halt` instruction.

17.5 Exception classes

The sysutils unit contains a great deal of exception handling. It defines the base exception class, `Exception`

```
Exception = class(TObject)
private
```

```
  fmessage : string;
  fhelpcontext : longint;
public
  constructor create(const msg : string);
  constructor createres(indent : longint);
  property helpcontext : longint read fhelpcontext write fhelpcontext;
  property message : string read fmessage write fmessage;
end;
ExceptClass = Class of Exception;
```

And uses this declaration to define quite a number of exceptions, for instance:

```
{ mathematical exceptions }
EIntError = class(Exception);
EDivByZero = class(EIntError);
ERangeError = class(EIntError);
EIntOverflow = class(EIntError);
EMathError = class(Exception);
```

The SysUtils unit also installs an exception handler. If an exception is unhandled by any exception handling block, this handler is called by the Run-Time library. Basically, it prints the exception address, and it prints the message of the Exception object, and exits with an exit code of 217. If the exception object is not a descendent object of the `Exception` object, then the class name is printed instead of the exception message.

It is recommended to use the `Exception` object or a descendant class for all `raise` statements, since then the message field of the exception object can be used.

Chapter 18

Using assembler

Free Pascal supports the use of assembler in code, but not inline assembler macros. To have more information on the processor specific assembler syntax and its limitations, see the Programmer's Guide.

18.1 Assembler statements

The following is an example of assembler inclusion in Pascal code.

```
...
Statements;
...
Asm
  the asm code here
  ...
end;
...
Statements;
```

The assembler instructions between the `Asm` and `end` keywords will be inserted in the assembler generated by the compiler. Conditionals can be used in assembler code, the compiler will recognize them, and treat them as any other conditionals.

18.2 Assembler procedures and functions

Assembler procedures and functions are declared using the `Assembler` directive. This permits the code generator to make a number of code generation optimizations.

The code generator does not generate any stack frame (entry and exit code for the routine) if it contains no local variables and no parameters. In the case of functions, ordinal values must be returned in the accumulator. In the case of floating point values, these depend on the target processor and emulation options.

Appendix A

Pascal grammar

Below is a formal grammar of the pascal language in EBNF notation. It is not 100% complete, some parts such as assembler or objective-c have not been specified.

⟨*module*⟩ = ⟨*program_module*⟩ | ⟨*library_module*⟩ | ⟨*package_module*⟩ | ⟨*unit_module*⟩ ;

⟨*program_module*⟩ = ⟨*program_header*⟩ **";"** [⟨*full_uses_clause*⟩] ⟨*block*⟩ **"."** ;

⟨*program_header*⟩ = **"program"** ⟨*identifier*⟩ [**"("** ⟨*program_parameters*⟩ **")"**] ;

⟨*program_parameters*⟩ = ⟨*identifier_list*⟩ ;

⟨*full_uses_clause*⟩ = **"uses"** ⟨*full_uses_item*⟩ **","** ⟨*full_uses_item*⟩ **";"** ;

⟨*full_uses_item*⟩ = ⟨*identifier*⟩ [**"in"** ⟨*string_literal*⟩] ;

⟨*library_module*⟩ = ⟨*library_header*⟩ **";"** [⟨*uses_clause*⟩] ⟨*block*⟩ **"."** ;

⟨*library_header*⟩ = **"library"** ⟨*identifier*⟩ ;

⟨*package_module*⟩ = ⟨*package_header*⟩ **";"** [⟨*contains_clause*⟩] [⟨*requires_clause*⟩] **"end"** **"."** ;

⟨*package_header*⟩ = **"package"** ⟨*identifier*⟩ ;

⟨*contains_clause*⟩ = **"contains"** ⟨*full_uses_item*⟩ **","** ⟨*full_uses_item*⟩ **";"** ;

⟨*requires_clause*⟩ = **"requires"** ⟨*qualifier_list*⟩ **";"** ;

⟨*unit_module*⟩ = ⟨*unit_header*⟩ ⟨*interface_section*⟩ ⟨*implementation_section*⟩ ([⟨*initialization_section*⟩] [⟨*finalization_section*⟩]
| **"begin"** ⟨*statement_list*⟩) **"end"** **"."** ;

⟨*unit_header*⟩ = **"unit"** ⟨*qualifier*⟩ [⟨*hintdirectives*⟩] **";"** ;

⟨*interface_section*⟩ = **"interface"** [⟨*uses_clause*⟩] [⟨*interface_parts*⟩] ;

⟨*uses_clause*⟩ = **"uses"** ⟨*identifier*⟩ **","** ⟨*identifier*⟩ **";"** ;

⟨*interface_parts*⟩ = ⟨*interface_part*⟩ ⟨*interface_part*⟩ ;

⟨*interface_part*⟩ = ⟨*constant_declaration_block*⟩
  | ⟨*type_declaration_block*⟩
  | ⟨*variable_declaration_block*⟩
  | ⟨*property_declaration_block*⟩
  | ⟨*procedure_headers_block*⟩ ;

⟨*implementation_section*⟩ = **"implementation"** [⟨*uses_clause*⟩] ⟨*declaration_part*⟩ ;

⟨*initialization_section*⟩ = **"initialization"** ⟨*statement_list*⟩ ;

⟨*finalization_section*⟩ = **"finalization"** ⟨*statement_list*⟩ ;

⟨*exports_clause*⟩ = **"exports"** ⟨*exports_list*⟩ **";"** ;

⟨*exports_list*⟩ = ⟨*exports_entry*⟩ **","** ⟨*exports_entry*⟩ ;

⟨*exports_entry*⟩ = ⟨*identifier*⟩ [**"index"** ⟨*integer_constant*⟩] [**"name"** ⟨*string_constant*⟩] ;

⟨*procedure_headers_block*⟩ = ⟨*procedure_header*⟩
  | ⟨*function_header*⟩
  | ⟨*operator_header*⟩;

⟨*procedure_header*⟩ = ⟨*generic_procedure_header*⟩
  | ⟨*regular_procedure_header*⟩ ;

⟨*generic_procedure_header*⟩ = **"generic"** **"procedure"** ⟨*qualifier*⟩ ⟨*generic_template_list*⟩ ⟨*formal_parameter_list*⟩
    ";" [⟨*call_modifiers*⟩ **";"**] [⟨*hintdirectives*⟩] ;

⟨*regular_procedure_header*⟩ = **"procedure"** ⟨*qualifier*⟩ [⟨*generic_template_list*⟩] ⟨*formal_parameter_list*⟩
    ";" [⟨*call_modifiers*⟩ **";"**] [⟨*hintdirectives*⟩] ;

⟨*function_header*⟩ = ⟨*generic_function_header*⟩
  | ⟨*regular_function_header*⟩ ;

⟨*generic_function_header*⟩ = **"generic"** **"function"** ⟨*qualifier*⟩ [**"<"** ⟨*generic_template_list*⟩ **">"**
    ] ⟨*formal_parameter_list*⟩ **":"** ⟨*type*⟩ [⟨*call_modifiers*⟩ **";"**] [⟨*hintdirectives*⟩] ;

⟨*regular_function_header*⟩ = **"function"** ⟨*qualifier*⟩ [**"<"** ⟨*generic_template_list*⟩ **">"**] ⟨*formal_parameter_list*⟩
    ":" ⟨*type*⟩ [⟨*call_modifiers*⟩ **";"**] [⟨*hintdirectives*⟩] ;

⟨*operator_header*⟩ = **"operator"** ⟨*operator_definition*⟩ [⟨*result_identifier*⟩] **":"** ⟨*type*⟩ ;

⟨*operator_definition*⟩ = ⟨*assignment_operator_definition*⟩
  | ⟨*arithmetic_operator_definition*⟩
  | ⟨*comparison_operator_definition*⟩
  | ⟨*logical_operator_definition*⟩
  | ⟨*other_operator_definition*⟩ ;

⟨*generic_template_list*⟩ = ⟨*constrained_template_list*⟩
  | ⟨*free_template_list*⟩ ;

⟨*constrained_template_list*⟩ = ⟨*constrained_templates*⟩ **";"** ⟨*constrained_templates*⟩ ;

⟨*free_template_list*⟩ = ⟨*identifier*⟩ **","** ⟨*qualifier*⟩ ;

⟨*constrained_templates*⟩ = ⟨*identifier_list*⟩ **":"** ⟨*type_constraint*⟩ **","** ⟨*type_constraint*⟩ ;

⟨*identifier_list*⟩ = ⟨*identifier*⟩ **","** ⟨*identifier*⟩ ;

⟨*type_constraint*⟩ = ⟨*identifier*⟩
| **"class"**
| **"record"** ;

⟨*block*⟩ = ⟨*declaration_block*⟩ ⟨*statement_block*⟩ ;

⟨*declaration_block*⟩ = ⟨*label_declaration_block*⟩
| ⟨*constant_declaration_block*⟩
| ⟨*resourcestring_declaration_block*⟩
| ⟨*class_func_proc_declaration_block*⟩
| ⟨*func_proc_declaration_block*⟩
| ⟨*type_declaration_block*⟩
| ⟨*variable_declaration_block*⟩
| ⟨*threadvariable_declaration_block*⟩
| ⟨*exports_clause*⟩ ;

⟨*func_proc_declaration_block*⟩ = ⟨*procedure_declaration*⟩
| ⟨*function_declaration*⟩
| ⟨*constructor_declaration*⟩
| ⟨*destructor_declaration*⟩
| ⟨*operator_declaration*⟩ ;

⟨*statement_block*⟩ = ⟨*compound_statement*⟩ ;

⟨*label_declaration_block*⟩ = **"label"** ⟨*label_def*⟩ [**","** ⟨*label_def*⟩] **";"** ;

⟨*constant_declaration_block*⟩ = **"const"** ⟨*constant_declaration*⟩ [⟨*hintdirectives*⟩] ;

⟨*constant_declaration*⟩ = ⟨*identifier*⟩ **":"** ⟨*constant_expression*⟩ **";"**
| ⟨*identifier*⟩ **":"** ⟨*type*⟩ **"="** ⟨*constant_expression*⟩ **";"** ;

⟨*resourcestring_declaration_block*⟩ = **"resourcestring"** ⟨*string_constant_declaration*⟩ ;

⟨*string_constant_declaration*⟩ = ⟨*identifier*⟩ **"="** ⟨*string_constant_literal*⟩ **";"** ;

⟨*variable_declaration_block*⟩ = **"var"** ⟨*variable_declaration*⟩ ;

⟨*threadvariable_declaration_block*⟩ = **"threadvar"** ⟨*variable_declaration*⟩ ;

⟨*variable_declaration*⟩ = ⟨*identifier_list*⟩ **":"** ⟨*type_*⟩ [**"="** ⟨*expression*⟩] [⟨*variable_modifiers*⟩]
[⟨*hintdirectives*⟩];

⟨*variable_modifiers*⟩ = **"absolute"** ⟨*absolute_target*⟩
| **";" "export"**
| **"cvar"**
| **"external"** [⟨*string_constant*⟩] [**"name"** ⟨*string_constant*⟩] ;

⟨*absolute_target*⟩ = ⟨*identifier*⟩ | ⟨*expression*⟩ ;

⟨*type_declaration_block*⟩ = **"type"** ⟨*type_declaration*⟩ ;

⟨*hintdirective*⟩ = **"deprecated"** [⟨*string_constant*⟩]
  | **"experimental"**
  | **"platform"**
  | **"unimplemented"** ;

⟨*hintdirectives*⟩ = ⟨*hintdirective*⟩ **";"** ⟨*hintdirective*⟩ ;

⟨*typed_constant*⟩ = ⟨*factor*⟩ | ⟨*procedural_constant*⟩ ;

⟨*type_declaration*⟩ = ⟨*identifier*⟩ **"="** ⟨*type*⟩ [⟨*hintdirectives*⟩] **";"** ;

⟨*type*⟩ = [**"type"**] ⟨*simple_type*⟩
  | ⟨*pointer_type*⟩
  | ⟨*full_procedural_type*⟩
  | ⟨*forward_class_type*⟩
  | ⟨*structured_type*⟩
  | ⟨*class_reference_type*⟩
  | ⟨*generic_type*⟩
  | ⟨*specialized_type*⟩
  | **"helper"** ⟨*helper_type*⟩ ;

⟨*forward_class_type*⟩ = **"class"** ;

⟨*class_reference_type*⟩ = **"class"** **"of"** ⟨*identifier*⟩ ;

⟨*pointer_type*⟩ = **"^"** ⟨*qualifier*⟩ ;

⟨*simple_type*⟩ = ⟨*string_type*⟩
  | ⟨*ordinal_type*⟩ ;

⟨*string_type*⟩ = **"string"** **"["** ⟨*constant_expression*⟩ **"]"**
  | **"string"** **"("** ⟨*positive_integer*⟩ **")"**
  | **"ansistring"**
  | **"widestring"**
  | **"unicodestring"** ;

⟨*type_name*⟩ = ⟨*identifier*⟩ ;

⟨*subrange_type*⟩ = ⟨*simple_constant_expression*⟩ **".."** ⟨*simple_constant_expression*⟩ ;

⟨*ordinal_type*⟩ = ⟨*enumerated_type*⟩
  | ⟨*builtin_ordinal*⟩
  | ⟨*subrange_type*⟩ ;

⟨*builtin_ordinal*⟩ = **"byte"**
  | **"shortint"**
  | **"word"**
  | **"smallint"**
  | **"cardinal"**
  | **"longint"**
  | **"qword"**
  | **"int64"** ;

⟨*enumerated_type*⟩ = **"("** ⟨*enumerated_item*⟩ **","** ⟨*enumerated_item*⟩ **")"** ;

⟨*enumerated_item*⟩ = ⟨*qualifier*⟩
  | ⟨*qualifier*⟩ **"="** ⟨*expression*⟩
  | ⟨*qualifier*⟩ **":="** ⟨*expression*⟩ ;

⟨*structured_type*⟩ = ⟨*set_type*⟩
  | ⟨*file_type*⟩
  | ⟨*interface_type*⟩
  | **"packed"** ⟨*unpacked_type*⟩
  | **"bitpacked"** ⟨*unpacked_type*⟩ ;

⟨*unpacked_type*⟩ = ⟨*array_type*⟩
  | ⟨*record_type*⟩
  | ⟨*object_type*⟩
  | ⟨*class_type*⟩ ;

⟨*set_type*⟩ = **"set"** **"of"** ⟨*ordinal_type*⟩ ;

⟨*file_type*⟩ = **"file"** [**"of"** ⟨*type*⟩]
  | **"text"** ;

⟨*array_type*⟩ = ⟨*array*⟩ [**"["** ⟨*ordinal_type*⟩ **","** ⟨*ordinal_type*⟩ **"]"**] **"of"** ⟨*type*⟩ ;

⟨*qualifier_list*⟩ = ⟨*qualifier*⟩ **","** ⟨*qualifier*⟩ ;

⟨*record_type*⟩ = **"record"** ⟨*record_component_list*⟩ [⟨*variant_part*⟩] [**";"**] **"end"** ;

⟨*record_component_list*⟩ = ⟨*record_visibility_specifier*⟩
  | ⟨*field_definition*⟩ [**";"**]
  | ⟨*record_method_definition*⟩
  | ⟨*property_definition*⟩
  | ⟨*record_operator_definition*⟩ ;

⟨*field_definition*⟩ = ⟨*identifier_list*⟩ **":"** ⟨*type*⟩ ;

⟨*variant_part*⟩ = **"case"** [⟨*identifier*⟩ **":"**] ⟨*type*⟩ ⟨*of*⟩ ⟨*variant_case*⟩ [**";"** [⟨*variant_case*⟩]] ;

⟨*variant_case*⟩ = ⟨*constant*⟩ **","** ⟨*constant*⟩ **":"** **"("** ⟨*variant_field_list*⟩ **")"** ;

⟨*variant_field_list*⟩ = ⟨*field_definition*⟩ **";"** ⟨*field_definition*⟩ ;

⟨*record_visibility_specifier*⟩ = **"public"**
  | [**"strict"**] **"private"** ;

⟨*record_method_definition*⟩ = ⟨*function_header*⟩
  | ⟨*procedure_header*⟩ ;

⟨*record_operator_definition*⟩ = **"class"** ⟨*operator_header*⟩;

⟨*full_procedural_type*⟩ = ⟨*procedural_type*⟩ [**";"** ⟨*call_modifiers*⟩];

⟨*procedural_type*⟩ = ⟨*func_proc_header*⟩
  | ⟨*func_proc_header*⟩ **"of"** **"object"**
  | ⟨*func_proc_header*⟩ **"is"** **"nested"** ;

⟨*func_proc_header*⟩ = **"function"** ⟨*formal_parameter_list*⟩ [**":"** ⟨*type*⟩]
  | **"procedure"** ⟨*formal_parameter_list*⟩ ;

⟨*call_modifiers*⟩ = **"inline"**
  | **"register"**
  | **"cdecl"**
  | **"cppdecl"**
  | **"pascal"**
  | **"stdcall"**
  | **"safecall"**
  | **"hardfloat"**
  | **"local"**
  | **"mwpascal"**
  | **"ms_abi_default"**
  | **"ms_abi_cdecl"**
  | **"nostackframe"**
  | **"saveregisters"**
  | **"softfloat"**
  | **"sysv_abi_default"**
  | **"sysv_abi_cdecl"**
  | **"vectorcall"**
  | **"varargs"** ;

⟨*property_definition*⟩ = ⟨*identifier*⟩ [⟨*property_interface*⟩] ⟨*property_specifiers*⟩ ;

⟨*property_interface*⟩ = [⟨*property_parameter_list*⟩] **":"** ⟨*qualifier*⟩ ;

⟨*property_parameter_list*⟩ = **"["** ⟨*parameter_declaration*⟩ **";"** ⟨*parameter_declaration*⟩ **"]"** ;

⟨*property_specifiers*⟩ = [**"index"** ⟨*expression*⟩] [⟨*read_specifier*⟩] [⟨*write_specifier*⟩] [⟨*default_specifier*⟩]
    ;

⟨*read_specifier*⟩ = **"read"** ⟨*qualifier*⟩ ;

⟨*write_specifier*⟩ = **"write"** ⟨*qualifier*⟩ ;

⟨*default_specifier*⟩ = **"default"** [⟨*constant*⟩] | **"nodefault"** ;

⟨*object_type*⟩ = **"object"** [**"abstract"** | **"sealed"**] [⟨*heritage*⟩] ⟨*object_component*⟩ **"end"** ;

⟨*heritage*⟩ = **"("** ⟨*qualifier*⟩ **")"** ;

⟨*object_component*⟩ = ⟨*object_visibility_specifier*⟩
  | ⟨*constant_declaration_block*⟩
  | ⟨*type_declaration_block*⟩
  | ⟨*object_field_definition*⟩
  | ⟨*object_method_definition*⟩
  | ⟨*property_declaration_block*⟩ ;

⟨*object_visibility_specifier*⟩ = **"private"** | **"protected"** | **"public"** ;

⟨*object_field_definition*⟩ = [[⟨*class*⟩] ⟨*var*⟩] ⟨*identifier_list*⟩ **":"** ⟨*type*⟩ **";"** [**"static"** **";"**] ;

⟨*object_method_definition*⟩ = ⟨*constructor_header*⟩ [⟨*method_directives*⟩]
  | ⟨*destructor_header*⟩ [⟨*method_directives*⟩]
  | ⟨*method_definition*⟩ [⟨*method_directives*⟩] ;

⟨*method_definition*⟩ = [⟨*class*⟩] ⟨*function_header*⟩
| [⟨*class*⟩] ⟨*procedure_header*⟩
| [⟨*class*⟩] ⟨*operator_header*⟩;

⟨*constructor_header*⟩ = **"constructor"** ⟨*identifier*⟩ ⟨*formal_parameter_list*⟩ ;

⟨*destructor_header*⟩ = **"destructor"** ⟨*qualifier*⟩ ⟨*formal_parameter_list*⟩ ;

⟨*method_directives*⟩ = [**"virtual"** **";"** [(**"abstract"** | **"reintroduce"**) **";"**]] [⟨*call_modifiers*⟩ **";"**] ;

⟨*class_type*⟩ = **"class"** [**"abstract"** | **"sealed"**] [⟨*class_heritage*⟩] ⟨*class_component*⟩ [**"end"**] ;

⟨*class_heritage*⟩ = **"("** ⟨*qualifier*⟩ **","** ⟨*qualifier*⟩ **")"** ;

⟨*class_component*⟩ = ⟨*class_visibility_specifier*⟩
| ⟨*constant_declaration_block*⟩
| ⟨*type_declaration_block*⟩
| [⟨*class*⟩] ⟨*var_declaration_block*⟩
| ⟨*class_field_definition*⟩ **";"**
| ⟨*class_method_definition*⟩ **";"**
| ⟨*class_property_definition*⟩ **";"** ;

⟨*class_visibility_specifier*⟩ = [**"strict"**] **"private"**
| [**"strict"**] **"protected"**
| **"public"**
| **"published"** ;

⟨*class_method_definition*⟩ = ⟨*constructor_header*⟩
| ⟨*destructor_header*⟩
| ⟨*method_definition*⟩ [⟨*class_method_directives*⟩ **";"**] [⟨*call_modifiers*⟩ **";"**] ;

⟨*class_method_directives*⟩ = **"virtual"** [⟨*abstract_or_reintroduce*⟩]
| **"dynamic"** [⟨*abstract_or_reintroduce*⟩]
| **"override"** [**";"** **"final"**]
| **"reintroduce"** [**";"** **"virtual"**]
| **"message"** (⟨*integer_constant*⟩ | ⟨*string_constant*⟩)
| **"static"** [**";"** **"inline"**] ;

⟨*abstract_or_reintroduce*⟩ = **";"** (**"abstract"** [**";"** **"overload"**] | **"reintroduce"**) ;

⟨*class_field_definition*⟩ = ⟨*identifier_list*⟩ **":"** ⟨*type*⟩ **";"** [**"static"** **";"**] ;

⟨*class_property_definition*⟩ = **"property"** ⟨*identifier*⟩ [⟨*property_interface*⟩] ⟨*class_property_specifiers*⟩ [⟨*hintdirectives*⟩] **";"** ;

⟨*class_property_specifiers*⟩ = [**"index"** ⟨*expression*⟩] [⟨*read_specifier*⟩] [⟨*write_specifier*⟩ | ⟨*implements_specifier*⟩] [⟨*default_specifier*⟩] [⟨*stored_specifier*⟩] [**"default"**] ;

⟨*implements_specifier*⟩ = **"implements"** ⟨*identifier*⟩ **","** ⟨*identifier*⟩ ;

⟨*stored_specifier*⟩ = **"stored"** ⟨*identifier*⟩;

⟨*interface_type*⟩ = ⟨*interface*⟩ [⟨*heritage*⟩] [⟨*guid*⟩] ⟨*interface_component*⟩ **"end"**;

⟨*interface_component*⟩ = ⟨*function_header*⟩ [**";"** ⟨*call_modifiers*⟩] **";"**
| ⟨*procedure_header*⟩ [**";"** ⟨*call_modifiers*⟩] **";"**
| ⟨*class_property_definition*⟩ **";"** ;

⟨*interface_property_specifiers*⟩ = [**"index"** ⟨*expression*⟩] [⟨*read_specifier*⟩] [⟨*write_specifier*⟩]
[⟨*default_specifier*⟩] [**"default"**] ;

⟨*guid*⟩ = **"["** ⟨*string_constant*⟩ **"]"** ;

⟨*generic_type*⟩ = [**"generic"**] ⟨*identifier*⟩ **"<"** ⟨*generic_template_list*⟩ **">"** **"="** ⟨*generic_type_def*⟩
";" ;

⟨*generic_type_def*⟩ = ⟨*class_type*⟩ | ⟨*object_type*⟩ | ⟨*interface_type*⟩ | ⟨*procedural_type*⟩ | ⟨*record_type*⟩
| ⟨*array_type*⟩ ;

⟨*specialized_type*⟩ = [**"specialize"**] ⟨*identifier*⟩ **"<"** ⟨*qualifier_list*⟩ **">"** ;

⟨*helper_type*⟩ = (⟨*class*⟩ | ⟨*record*⟩ | ⟨*type*⟩) **"helper"** [**"("** ⟨*base_helper*⟩ **")"**] ⟨*for*⟩ ⟨*qualifier*⟩
⟨*helper_component_list*⟩ **"end"** [⟨*hintdirectives*⟩] ;

⟨*helper_component_list*⟩ = ⟨*helper_component*⟩ **";"** ⟨*helper_component*⟩ ;

⟨*helper_component*⟩ = ⟨*method_definition*⟩
| ⟨*property_definition*⟩ ;

⟨*constructor_declaration*⟩ = ⟨*constructor_header*⟩ **";"** ⟨*subroutine_block*⟩ **";"** ;

⟨*destructor_declaration*⟩ = ⟨*destructor_header*⟩ **";"** ⟨*subroutine_block*⟩ **";"** ;

⟨*expression*⟩ = ⟨*simple_expression*⟩ [⟨*relop*⟩ ⟨*simple_expression*⟩] ;

⟨*relop*⟩ = **"<"** | **"<="** | **">"** | **">="** | **"="** | **"<>"** | **"in"** | **"is"** ;

⟨*simple_expression*⟩ = ⟨*term*⟩ ⟨*addop*⟩ ⟨*term*⟩ ;

⟨*simple_constant_expression*⟩ = ⟨*simple_expression*⟩ ;

⟨*addop*⟩ = **"+"** | **"-"** | **"or"** | **"xor"** | **"><"** ;

⟨*term*⟩ = ⟨*factor*⟩ ⟨*multop*⟩ ⟨*factor*⟩ ;

⟨*multop*⟩ = **"*"** | **"/"** | **"div"** | **"mod"** | **"and"** | **"shl"** | **"shr"** | **"as"** | **"«"** | **"»"** ;

⟨*factor*⟩ = ⟨*nested_expr_or_structured_const*⟩ ⟨*selector*⟩
| ⟨*varref_or_funcall_or_constid_or_cast*⟩
| ⟨*unsigned_constant*⟩
| **"not"** ⟨*factor*⟩
| ⟨*sign*⟩ ⟨*factor*⟩
| ⟨*set_constructor*⟩
| ⟨*address_factor*⟩
| ⟨*inherited_call*⟩ ;

⟨*varref_or_funcall_or_constid_or_cast*⟩ = ⟨*qualifier*⟩ [⟨*actual_parameter_list*⟩] ;

⟨*nested_expr_or_structured_const*⟩ = **"("** ⟨*element*⟩ (**","**|**";"**) [⟨*element*⟩] **")"** ;

⟨*element*⟩ = [**":"** ⟨*identifier*⟩ **":"**] ⟨*expression*⟩ ;

⟨*unsigned_constant*⟩ = ⟨*unsigned_number*⟩ | ⟨*character_string*⟩ | **"nil"** ;

⟨*sign*⟩ = **"+"** | **"-"** ;

⟨*actual_parameter_list*⟩ = **"("** [⟨*expression*⟩ **","** ⟨*expression*⟩] **")"** ;

⟨*set_constructor*⟩ = **"["** [⟨*set_group*⟩ **","** ⟨*set_group*⟩] **"]"** ;

⟨*set_group*⟩ = ⟨*expression*⟩ [**".."** ⟨*expression*⟩] ;

⟨*address_factor*⟩ = **"@"** (⟨*designator*⟩ | ⟨*nested_expr_or_structured_const*⟩) ;

⟨*statement_list*⟩ = ⟨*statement*⟩ **";"** [⟨*statement*⟩] ;

⟨*statement*⟩ = [⟨*label_def*⟩ **":"**] ⟨*unlabeled_statement*⟩ ;

⟨*unlabeled_statement*⟩ = ⟨*simple_statement*⟩
| ⟨*structured_statement*⟩
| ⟨*asm_statement*⟩ ;

⟨*simple_statement*⟩ = ⟨*inherited_call*⟩
| ⟨*assignment_or_call*⟩
| ⟨*goto_statement*⟩
| ⟨*raise_statement*⟩ ;

⟨*inherited_call*⟩ = **"inherited"** [⟨*identifier*⟩ [⟨*actual_parameter_list*⟩]] ;

⟨*assignment_or_call*⟩ = (⟨*designator*⟩ | ⟨*nested_expr_or_structured_const*⟩ ⟨*selector*⟩) [⟨*assignop*⟩ ⟨*expression*⟩] ;

⟨*assignop*⟩ = **":="** | **"+="** | **"-="** | **"*="** | **"/="** ;

⟨*goto_statement*⟩ = ⟨*goto*⟩ ⟨*label_def*⟩ ;

⟨*structured_statement*⟩ = ⟨*compound_statement*⟩
| ⟨*conditional_statement*⟩
| ⟨*loop_statement*⟩
| ⟨*with_statement*⟩
| ⟨*try_statement*⟩ ;

⟨*conditional_statement*⟩ = ⟨*case_statement*⟩
| ⟨*if_statement*⟩ ;

⟨*loop_statement*⟩ = ⟨*for_statement*⟩
| ⟨*repeat_statement*⟩
| ⟨*while_statement*⟩ ;

⟨*compound_statement*⟩ = **"begin"** ⟨*statement_list*⟩ **"end"** ;

⟨*case_statement*⟩ = ⟨*case*⟩ ⟨*expression*⟩ ⟨*of*⟩ ⟨*case_part*⟩ **";"** ⟨*case_part*⟩ [⟨*else_part*⟩] **"end"** ;

⟨*case_part*⟩ = ⟨*case_range*⟩ **","** ⟨*case_range*⟩ **":"** ⟨*statement*⟩ ;

⟨*case_range*⟩ = ⟨*constant_expression*⟩ [**".."** ⟨*constant_expression*⟩] ;

⟨*else_part*⟩ = **";"** (⟨*else*⟩ | **"otherwise"**) ⟨*statement_list*⟩ ;

⟨*if_statement*⟩ = ⟨*if*⟩ ⟨*expression*⟩ ⟨*then*⟩ ⟨*statement*⟩ [⟨*else*⟩ ⟨*statement*⟩] ;

⟨*for_statement*⟩ = ⟨*for_to_statement*⟩
| ⟨*for_downto_statement*⟩
| ⟨*for_in_statement*⟩ ;

⟨*for_to_statement*⟩ = ⟨*for*⟩ ⟨*identifier*⟩ **":="** ⟨*expression*⟩ **"to"** ⟨*expression*⟩ **"do"** ⟨*statement*⟩ ;

⟨*for_downto_statement*⟩ = ⟨*for*⟩ ⟨*identifier*⟩ **":="** ⟨*expression*⟩ **"downto"** ⟨*expression*⟩ **"do"** ⟨*statement*⟩ ;

⟨*for_in_statement*⟩ = ⟨*for*⟩ ⟨*identifier*⟩ ⟨*in*⟩ ⟨*enumerable*⟩ **"do"** ⟨*statement*⟩ ;

⟨*enumerable*⟩ = ⟨*enumerated_type*⟩
| ⟨*designator*⟩
| ⟨*set_constructor*⟩ ;

⟨*repeat_statement*⟩ = **"repeat"** ⟨*statement_list*⟩ **"until"** ⟨*expression*⟩ ;

⟨*while_statement*⟩ = **"while"** ⟨*expression*⟩ **"do"** ⟨*statement*⟩ ;

⟨*with_statement*⟩ = **"with"** ⟨*variable_reference*⟩ **","** ⟨*variable_reference*⟩ **"do"** ⟨*statement*⟩ ;

⟨*asm_statement*⟩ = **"asm"** ⟨*asm_statement_list*⟩ **"end"** [⟨*register_list*⟩] ;

⟨*register_list*⟩ = **"["** ⟨*string_constant*⟩ **","** ⟨*string_constant*⟩ **"]"** ;

⟨*raise_statement*⟩ = **"raise"** [⟨*expression*⟩ [⟨*exception_address*⟩]] ;

⟨*exception_address*⟩ = **"at"** ⟨*expression*⟩ [**","** ⟨*expression*⟩] ;

⟨*try_statement*⟩ = **"try"** ⟨*statement_list*⟩ (**"except"** ⟨*exceptionhandlers*⟩ | **"finally"** ⟨*statement_list*⟩) **"end"** ;

⟨*exceptionhandlers*⟩ = [⟨*statement_list*⟩ | ⟨*exception_handler*⟩ **";"** ⟨*exception_handler*⟩ [⟨*else*⟩ ⟨*statement_list*⟩]];

⟨*exception_handler*⟩ = **"on"** [⟨*identifier*⟩ **":"**] ⟨*type_name*⟩ **"do"** ⟨*statement*⟩ ;

⟨*procedure_declaration*⟩ = ⟨*procedure_header*⟩ **";"** ⟨*subroutine_block*⟩ **";"** ;

⟨*subroutine_block*⟩ = ⟨*block*⟩
| ⟨*external_directive*⟩
| ⟨*asm_block*⟩
| **"forward"** ;

⟨*function_declaration*⟩ = ⟨*function_header*⟩ **";"** ⟨*subroutine_block*⟩ **";"** ;

⟨*formal_parameter_list*⟩ = [**"("** ⟨*parameter_declaration*⟩ **";"** ⟨*parameter_declaration*⟩ **")"**] ;

⟨*parameter_declaration*⟩ = ⟨*constant_parameter*⟩
| ⟨*value_parameter*⟩
| ⟨*variable_parameter*⟩
| ⟨*out_parameter*⟩ ;

⟨*value_parameter*⟩ = ⟨*identifier_list*⟩ **":"** ⟨*parameter_type*⟩ [**"="** ⟨*default_parameter_value*⟩] ;

⟨*variable_parameter*⟩ = ⟨*var*⟩ ⟨*identifier_list*⟩ [**":"** ⟨*parameter_type*⟩] ;

⟨*out_parameter*⟩ = ⟨*out*⟩ ⟨*identifier_list*⟩ [**":"** ⟨*parameter_type*⟩] ;

⟨*constant_parameter*⟩ = (**"const"** | **"constref"**) ⟨*identifier_list*⟩ [**":"** ⟨*parameter_type*⟩ [**"="** ⟨*default_parameter_value*⟩]] ;

⟨*parameter_type*⟩ = [⟨*array*⟩ ⟨*of*⟩] (⟨*qualifier*⟩ | **"const"**) ;

⟨*external_directive*⟩ = **"external"** [⟨*string_constant*⟩ [**"name"** ⟨*string_constant*⟩ | **"index"** ⟨*integer_constant*⟩]] ;

⟨*asm_block*⟩ = **"assembler"** **";"** ⟨*asm_statement_list*⟩ ;

⟨*modifiers*⟩ = **";"** ⟨*modifier*⟩ **";"** ⟨*modifier*⟩ ;

⟨*modifier*⟩ = **"export"**
| **"alias"** **":"** ⟨*string_constant*⟩
| **"interrupt"**
| **"noreturn"**
| **"iocheck"**
| ⟨*call_modifier*⟩ ;

⟨*operator_implementation*⟩ = ⟨*operator_header*⟩ **";"** ⟨*subroutine_block*⟩ **";"** ;

⟨*assignment_operator_definition*⟩ = (**":="** | **"explicit"** | **"implicit"**) **"("** ⟨*parameter_list*⟩ **")"** ;

⟨*arithmetic_operator_definition*⟩ = (**"+"** | **"-"** | **"*"** | **"/"** | **"**"** | **"><"** | ⟨*div*⟩ | ⟨*mod*⟩ | ⟨*shl*⟩ | ⟨*shr*⟩) **"("** ⟨*parameter_list*⟩ **")"** ;

⟨*comparison_operator_definition*⟩ = (**"="** | **"<"** | **"<="** | **">"** | **">="** | **"<>"** | ⟨*in*⟩) **"("** ⟨*parameter_list*⟩ **")"** ;

⟨*logical_operator_definition*⟩ = (⟨*and*⟩ | ⟨*or*⟩ | ⟨*xor*⟩ | ⟨*not*⟩) **"("** ⟨*parameter_list*⟩ **")"** ;

⟨*other_operator_definition*⟩ = (**"enumerator"** | **"inc"** | **"dec"**) **"("** ⟨*parameter_list*⟩ **")"** ;

⟨*label_def*⟩ = ⟨*decimal_int*⟩ | ⟨*identifier*⟩ ;

⟨*identifier*⟩ = ⟨*letter*⟩ | **"_"** ⟨*letter*⟩ | ⟨*dec_digit*⟩ | **"_"** ;

⟨*designator*⟩ = (⟨*identifier*⟩ | ⟨*string*⟩) ⟨*selector*⟩ | ⟨*subrange*⟩ ;

⟨*selector*⟩ = **"."** ⟨*identifier*⟩ | **"["** ⟨*expression*⟩ **","** ⟨*expression*⟩ **"]"** | **"^"** | ⟨*actual_parameter_list*⟩ ;

⟨*qualifier*⟩ = ⟨*identifier*⟩ [**"."** ⟨*identifier*⟩] ;

⟨*string_constant*⟩ = ⟨*character_string*⟩ **"+"** ⟨*character_string*⟩ ;

⟨*integer_constant*⟩ = ⟨*signed_integer*⟩ | ⟨*unsigned_integer*⟩ ;

⟨*comment*⟩ = **"("** non_comment_characters **"*)"**
| **"{"** ⟨*non_comment_characters*⟩ **"}"**
| **"//"** ⟨*non_comment_character*⟩ ⟨*EOL*⟩ ;

⟨*letter*⟩ = **"A"** | **"B"** | **"C"** | **"D"** | **"E"** | **"F"** | **"G"** | **"H"** | **"I"** | **"J"** | **"K"** | **"L"** | **"M"**
| **"N"** | **"O"** | **"P"** | **"Q"** | **"R"** | **"S"** | **"T"** | **"U"** | **"V"** | **"W"** | **"X"** | **"Y"** | **"Z"**
| **"a"** | **"b"** | **"c"** | **"d"** | **"e"** | **"f"** | **"g"** | **"h"** | **"i"** | **"j"** | **"k"** | **"l"** | **"m"**
| **"n"** | **"o"** | **"p"** | **"q"** | **"r"** | **"s"** | **"t"** | **"u"** | **"v"** | **"w"** | **"x"** | **"y"** | **"z"** ;

⟨*binary_digit*⟩ = **"0"** | **"1"** ; ⟨*octal_digit*⟩ = ⟨*binary_digit*⟩ | **"2"** **"3"** | **"4"** |**"5"** | **"6"** | **"7"** ;
⟨*decimal_digit*⟩ = ⟨*octal_digit*⟩ | **"8"** | **"9"** ; ⟨*hexadecimal_digit*⟩ = ⟨*decimal_digit*⟩ | **"A"** |
"B" | **"C"** | **"D"** | **"E"** | **"F"** | **"a"** | **"b"** | **"c"** | **"d"** | **"e"** | **"f"** ;

⟨*binary_digit_sequence*⟩ = ⟨*binary_digit*⟩ (**"_"**| ⟨*binary_digit*⟩) ; ⟨*octal_digit_sequence*⟩ = ⟨*octal_digit*⟩
(**"_"**| ⟨*octal_digit*⟩) ; ⟨*decimal_digit_sequence*⟩ = ⟨*decimal_digit*⟩ (**"_"**| ⟨*decimal_digit*⟩) ;
⟨*hexadecimal_digit_sequence*⟩ = ⟨*hexadecimal_digit*⟩ (**"_"**| ⟨*hexadecimal_digit*⟩) ;

⟨*unsigned_integer*⟩ = ⟨*decimal_digit_sequence*⟩
| **"%"** ⟨*binary_digit_sequence*⟩
| **"&"** ⟨*octal_digit_sequence*⟩
| **"$"** ⟨*hexadecimal_digit_sequence*⟩ ;

⟨*signed_integer*⟩ = ⟨*sign*⟩ ⟨*unsigned_integer*⟩ ;

⟨*unsigned_real*⟩ = ⟨*dec_digit_sequence*⟩ [**"."** ⟨*dec_digit_sequence*⟩] [⟨*scale_factor*⟩] ;

⟨*scale_factor*⟩ = **"E"** | **"e"** [⟨*sign*⟩] ⟨*dec_digit_sequence*⟩ ;

⟨*unsigned_number*⟩ = ⟨*unsigned_real*⟩ | ⟨*unsigned_integer*⟩ ;

⟨*signed_number*⟩ = ⟨*sign*⟩ ⟨*unsigned_number*⟩ ;

⟨*quoted_string*⟩ = **"'"** ⟨*string_character*⟩ **"'"** ;

⟨*string_character*⟩ = **"'"** | ⟨*non_terminal_string_character*⟩ ;

⟨*control_string*⟩ = **"#"** ⟨*unsigned_integer*⟩ ;

⟨*character_string*⟩ = (⟨*quoted_string*⟩ | ⟨*control_string*⟩) ⟨*quoted_string*⟩ | ⟨*control_string*⟩ ;

Index

Abstract, 89
abstract, 82
Address, 186
Alias, 237
Ansistring, 36, 41, 42
Array, 46, 226, 227
    Constructor, 53
    Dynamic, 48
    Of const, 227
    Static, 47
array, 68
Array constructor, 53
Asm, 217
Assembler, 217, 236, 278

block, 266
Boolean, 29

Case, 200
cdecl, 238
Char, 33
Class, 92, 103
Class helpers, 158
Classes, 92
CodePage, 36, 38
    Conversion, 38
COM, 67, 129, 130
Comments, 13
Comp, 33
compilerproc, 239
Const, 23
    String, 23
Constants, 21
    Ordinary, 21
    String, 19, 22, 42
    Typed, 22
constants
    array, 51
Constructor, 85, 99, 183, 184
CORBA, 67, 130
cppdecl, 239
Currency, 33

Delphi
    Compatibility, 140
    Syntax, 140
    Type overloads, 142
Destructor, 85, 100
Directives
    Hint, 17
Dispatch, 108
DispatchInterfaces
    COM, 129
DispatchStr, 109
Double, 33

else, 200, 202
except, 274, 276
Exception, 272
Exceptions, 272
    Catching, 272, 274
    Classes, 276
    Handling, 275, 276
    Raising, 272
export, 239
Expression, 214, 215
Expressions, 178
Extended, 33
Extended records, 151
External, 235
external, 74, 236

Fields, 55, 83
File, 60
finally, 275, 276
For, 204, 206
    downto, 204
    in, 206
    to, 204
Forward, 62, 234
Function, 220
Functions, 219
    Assembler, 236, 278
    External, 235
    Forward, 234
    Generic, 138
    Modifiers, 237
    Overloaded, 234

generic, 133

Generics, 132
Overloads, 142
Type Restrictions, 137

hardfloat, 239
Hint directives, 17

Identifiers, 16
If, 202
index, 113, 236
Inherited, 102
inherited, 88, 117
inline, 239
interface, 122
Interfaces, 67, 69, 122
COM, 129
CORBA, 130
Implementations, 124
interrupt, 240
iocheck, 240

Labels, 19
Libraries, 270
library, 270
local, 240

Message, 108
message, 108
Method
Overloaded, 110
Methods, 86, 101
Abstract, 89
Class, 103
Final, 103
Message, 108
Static, 87, 106
Virtual, 87, 89, 102
Modifiers, 15, 237, 245
Alias, 237
cdecl, 238
compilerproc, 239
cppdecl, 239
export, 239
hardfloat, 239
inline, 239
interrupt, 240
iocheck, 240
local, 240
MS_ABI_CDecl, 240
MS_ABI_Default, 240
MWPascal, 241
noreturn, 241
nostackframe, 241
overload, 241
pascal, 243
public, 243
register, 243
safecall, 244
saveregisters, 244
softfloat, 244
stdcall, 244
SYSV_ABI_CDecl, 244
SYSV_ABI_Default, 244
varargs, 245
VectorCall, 244
winapi, 245
MS_ABI_CDecl, 240
MS_ABI_Default, 240
MWPascal, 241

name, 236
noreturn, 241
nostackframe, 241
Numbers, 18
Binary, 18
Decimal, 18
Hexadecimal, 18
Octal, 18
Real, 18

object, 81
Objective-Pascal, 168
Objective-Pascal Classes, 168
Objects, 81
Objects, abstract, 82
Objects, sealed, 82
Operators, 21, 44, 62, 178, 186, 187
Arithmetic, 187, 252
Assignment, 248
Binary, 252
Boolean, 189
Comparison, 253
Dynamic arrays, 190
Logical, 188
Relational, 192
Set, 190
String, 189
Unary, 188
operators, 246
otherwise, 200
overload, 241
overloading
operators, 246
Override, 102
override, 88

Packed, 56, 82, 99
Parameters, 221

Constant, 221, 224
ellipsis, 230
Open Array, 226, 227
Out, 223
Reference counted, 230
UNIV type, 226
Untyped, 229
Untypes, 221
Value, 222
Var, 113, 221, 223
pascal, 243
PChar, 38, 41
Pointer, 60
Private, 91, 94, 112
strict, 94
private, 82
Procedural, 63
Procedure, 63, 219
Procedures, 219
Generic, 138
program, 259
Properties, 79, 111
Array, 114
Indexed, 113
Property, 104, 111
Protected, 91, 94
Public, 91, 94
public, 82, 243
Published, 94, 112
PUnicodeChar, 41

Raise, 272
Rawbytestring, 40
Read, 112
Real, 33
Record, 55
Constant, 76
Record helpers, 158
register, 243
reintroduce, 102
Repeat, 214
Reserved words, 14
Delphi, 15
Modifiers, 15
Turbo Pascal, 14
Resourcestring, 23

safecall, 244
saveregisters, 244
Scope, 37, 50, 75, 81, 91, 94, 267
block, 268
Class, 268
record, 268
unit, 268
sealed, 82
Self, 86, 104
Set, 59
Shortstring, 35
Single, 33
softfloat, 244
specialize, 135
Statements, 196
Assembler, 217, 278
Assignment, 196
Case, 200
Compound, 200
Exception, 217
For, 204, 206
Goto, 198
if, 202
Loop, 204, 206, 214
Procedure, 198
Repeat, 214
Simple, 196
Structured, 199
While, 214
With, 215
Static class methods, 106
stdcall, 244
String, 19
Ansistring, 36
CodePage, 36
Codepage, 38
Constant, 42
Multi-byte, 41
RawByteString, 40
ShortString, 35
Single-byte string, 34
UnicodeString, 41
UTF8String, 40
WideString, 41
String literal, 19
Symbols, 12
Syntax diagrams, 10
SYSV_ABI_CDecl, 244
SYSV_ABI_Default, 244

Text, 60
then, 202
Thread Variables, 78
Threadvar, 78
Tokens, 12
Comments, 13
Identifiers, 16
Numbers, 18
Reserved words, 14
Strings, 19
Symbols, 12

try, 275, 276
Type, 25
  Compatibility, 29, 52, 71, 143, 184, 186, 222
  Generics, 132, 133
  Helpers, 158
  Restrictions, 137
  Specializing, 135
  Type overloads, 142
Typecast, 37, 38, 41, 184–186
  Unaligned, 186
  Value, 184
  Variable, 185
Types, 25
  Ansistring, 36
  Array, 46–48
  Base, 26
  Boolean, 29
  Char, 33
  Class, 92
  Class helpers, 158
  Constructor, 53
  Enumeration, 30
  Extended record, 151
  File, 60
  Float, 33
  Forward declaration, 62
  Integer, 28
  Object, 81
  Ordinal, 27
  PChar, 38, 42
  Pointer, 42, 60
  Procedural, 63
  Rawbytestring, 40
  Real, 33
  Record, 55
  Record helpers, 158
  Reference counted, 36, 38, 40, 41, 48, 50, 130
  Set, 59
  ShortString, 35
  String, 34
  Structured, 45
  Subrange, 32
  Type helpers, 158
  UnicodeString, 40, 41
  UTF8String, 40
  Variant, 67
  Widestring, 41

Unicodestring, 41
unit, 261, 268
UNIV, 226
uses, 259
UTF8String, 40

Var, 73
varargs, 245
Variable, 73
Variables, 73
  Initialized, 22, 75, 77
Variant, 67
VectorCall, 244
Virtual, 85, 87, 102, 108
Visibility, 81, 91, 122
  Private, 81, 94
  Protected, 94
  Public, 81, 94
  Published, 94
  Strict Private, 94
  Strict Protected, 94

While, 214
WideChar, 34
Widestring, 41
winapi, 245
With, 215
Write, 112