# Free Pascal :
# Reference guide.

Reference guide for Free Pascal, version 2.2
Document version 2.0
January 2010

Michaël Van Canneyt

# Contents


**1 Pascal Tokens** **8**

1.1 Symbols . . . 8
1.2 Comments . . . 8
1.3 Reserved words . . . 9
1.3.1 Turbo Pascal reserved words . . . 9
1.3.2 Delphi reserved words . . . 10
1.3.3 Free Pascal reserved words . . . 10
1.3.4 Modifiers . . . 10
1.4 Identifiers . . . 10
1.5 Hint directives . . . 11
1.6 Numbers . . . 12
1.7 Labels . . . 13
1.8 Character strings . . . 13

**2 Constants** **15**

2.1 Ordinary constants . . . 15
2.2 Typed constants . . . 16
2.3 Resource strings . . . 16

**3 Types** **17**

3.1 Base types . . . 17
3.1.1 Ordinal types . . . 18
Integers . . . 18
Boolean types . . . 19
Enumeration types . . . 20
Subrange types . . . 21
3.1.2 Real types . . . 22
3.2 Character types . . . 22
3.2.1 Char . . . 22
3.2.2 Strings . . . 22
3.2.3 Short strings . . . 23

3.2.4 Ansistrings . . . 23
3.2.5 WideStrings . . . 25
3.2.6 Constant strings . . . 26
3.2.7 PChar - Null terminated strings . . . 26
3.3 Structured Types . . . 27
Packed structured types . . . 27
3.3.1 Arrays . . . 28
Static arrays . . . 29
Dynamic arrays . . . 30
Packing and unpacking an array . . . 32
3.3.2 Record types . . . 32
3.3.3 Set types . . . 36
3.3.4 File types . . . 37
3.4 Pointers . . . 37
3.5 Forward type declarations . . . 39
3.6 Procedural types . . . 40
3.7 Variant types . . . 41
3.7.1 Definition . . . 41
3.7.2 Variants in assignments and expressions . . . 42
3.7.3 Variants and interfaces . . . 43

**4 Variables** **44**
4.1 Definition . . . 44
4.2 Declaration . . . 44
4.3 Scope . . . 46
4.4 Initialized variables . . . 46
4.5 Thread Variables . . . 47
4.6 Properties . . . 47

**5 Objects** **51**
5.1 Declaration . . . 51
5.2 Fields . . . 52
5.3 Static fields . . . 53
5.4 Constructors and destructors . . . 53
5.5 Methods . . . 55
5.5.1 Declaration . . . 55
5.5.2 Method invocation . . . 55
Static methods . . . 55
Virtual methods . . . 56
Abstract methods . . . 57
5.6 Visibility . . . 58

**6 Classes** **60**
6.1 Class definitions . . . 60
6.2 Class instantiation . . . 63
6.3 Methods . . . 64
6.3.1 Declaration . . . 64
6.3.2 invocation . . . 64
6.3.3 Virtual methods . . . 64
6.3.4 Class methods . . . 65
6.3.5 Message methods . . . 66
6.3.6 Using inherited . . . 68
6.4 Properties . . . 69

**7 Interfaces** **74**
7.1 Definition . . . 74
7.2 Interface identification: A GUID . . . 75
7.3 Interface implementations . . . 76
7.4 Interfaces and COM . . . 77
7.5 CORBA and other Interfaces . . . 77

**8 Generics** **78**
8.1 Introduction . . . 78
8.2 Generic class definition . . . 78
8.3 Generic class specialization . . . 80
8.4 A word about scope . . . 81

**9 Expressions** **84**
9.1 Expression syntax . . . 85
9.2 Function calls . . . 86
9.3 Set constructors . . . 87
9.4 Value typecasts . . . 88
9.5 Variable typecasts . . . 89
9.6 Unaligned typecasts . . . 89
9.7 The @ operator . . . 90
9.8 Operators . . . 91
9.8.1 Arithmetic operators . . . 91
9.8.2 Logical operators . . . 92
9.8.3 Boolean operators . . . 92
9.8.4 String operators . . . 93
9.8.5 Set operators . . . 93
9.8.6 Relational operators . . . 94
9.8.7 Class operators . . . 95

**10 Statements** **97**
10.1 Simple statements . . . 97
10.1.1 Assignments . . . 97
10.1.2 Procedure statements . . . 98
10.1.3 Goto statements . . . 99
10.2 Structured statements . . . 100
10.2.1 Compound statements . . . 100
10.2.2 The `Case` statement . . . 101
10.2.3 The `If..then..else` statement . . . 102
10.2.4 The `For..to/downto..do` statement . . . 103
10.2.5 The `Repeat..until` statement . . . 104
10.2.6 The `While..do` statement . . . 105
10.2.7 The `With` statement . . . 105
10.2.8 Exception Statements . . . 107
10.3 Assembler statements . . . 107

**11 Using functions and procedures** **109**
11.1 Procedure declaration . . . 109
11.2 Function declaration . . . 110
11.3 Function results . . . 110
11.4 Parameter lists . . . 111
11.4.1 Value parameters . . . 111
11.4.2 Variable parameters . . . 112
11.4.3 Out parameters . . . 113
11.4.4 Constant parameters . . . 114
11.4.5 Open array parameters . . . 114
11.4.6 Array of const . . . 115
11.5 Function overloading . . . 117
11.6 Forward defined functions . . . 118
11.7 External functions . . . 119
11.8 Assembler functions . . . 120
11.9 Modifiers . . . 120
11.9.1 alias . . . 121
11.9.2 cdecl . . . 122
11.9.3 export . . . 122
11.9.4 inline . . . 122
11.9.5 interrupt . . . 123
11.9.6 local . . . 123
11.9.7 nostackframe . . . 123
11.9.8 overload . . . 123

11.9.9 pascal . . . 124
11.9.10 public . . . 124
11.9.11 register . . . 125
11.9.12 safecall . . . 125
11.9.13 saveregisters . . . 126
11.9.14 softfloat . . . 126
11.9.15 stdcall . . . 126
11.9.16 varargs . . . 126
11.10 Unsupported Turbo Pascal modifiers . . . 126

**12 Operator overloading** **127**
12.1 Introduction . . . 127
12.2 Operator declarations . . . 127
12.3 Assignment operators . . . 128
12.4 Arithmetic operators . . . 130
12.5 Comparision operator . . . 131

**13 Programs, units, blocks** **133**
13.1 Programs . . . 133
13.2 Units . . . 134
13.3 Unit dependencies . . . 136
13.4 Blocks . . . 137
13.5 Scope . . . 138
13.5.1 Block scope . . . 138
13.5.2 Record scope . . . 139
13.5.3 Class scope . . . 139
13.5.4 Unit scope . . . 139
13.6 Libraries . . . 140

**14 Exceptions** **141**
14.1 The raise statement . . . 141
14.2 The try...except statement . . . 142
14.3 The try...finally statement . . . 143
14.4 Exception handling nesting . . . 144
14.5 Exception classes . . . 144

**15 Using assembler** **145**
15.1 Assembler statements . . . 145
15.2 Assembler procedures and functions . . . 145

# List of Tables


3.1 Predefined integer types . . . . . 18
3.2 Predefined integer types . . . . . 19
3.3 Boolean types . . . . . 19
3.4 Supported Real types . . . . . 22
3.5 `PChar` pointer arithmetic . . . . . 27
3.6 Set Manipulation operators . . . . . 37

9.1 Precedence of operators . . . . . 84
9.2 Binary arithmetic operators . . . . . 91
9.3 Unary arithmetic operators . . . . . 91
9.4 Logical operators . . . . . 92
9.5 Boolean operators . . . . . 92
9.6 Set operators . . . . . 93
9.7 Relational operators . . . . . 95
9.8 Class operators . . . . . 95

10.1 Allowed C constructs in Free Pascal . . . . . 98

11.1 Unsupported modifiers . . . . . 126

# About this guide

This document serves as the reference for the Pascal langauge 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 Delphiimplementations.

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

## 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 2 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.

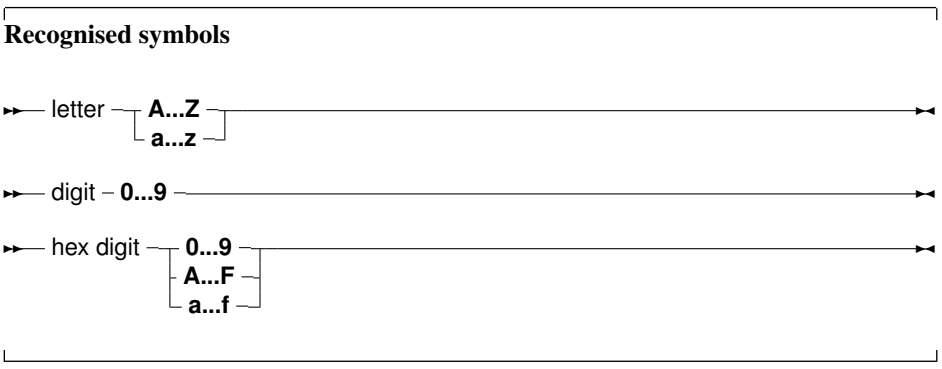
# Chapter 1

# Pascal Tokens

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 `]`. 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

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 `-Mturbo` switch.

**Remark:** In `TP` and `Delphi` mode, nested comments are not allowed, for maximum compatibility with existing code for those compilers.

## 1.3 Reserved words

Reserved words are part of the Pascal language, and cannot be redefined. They will be denoted as **this** throughout the syntax diagrams. Reserved words can be typed regardless of case, i.e. Pascal is case insensitive.

We make a distinction between Turbo Pascal and Delphireserved words, since in `TP` mode, only the Turbo Pascal reserved words are recognised, but the Delphiones can be redefined. By default, Free Pascal recognises the Delphireserved words.

### 1.3.1 Turbo Pascal reserved words

The following keywords exist in Turbo Pascal mode

```
absolute        do              in              on
and             downto          inherited       operator
array           else            inline          or
asm             end             interface       packed
begin           file            label           procedure
case            for             mod             program
const           function        nil             record
constructor     goto            not             reintroduce
destructor      if              object          repeat
div             implementation  of              self
```

```
set             then            until           with
shl             to              uses            xor
shr             type            var
string          unit            while
```

### 1.3.2 Delphi reserved words

The Delphireserved words are the same as the Turbo Pascal ones, plus the following keywords:

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

### 1.3.3 Free Pascal reserved words

On top of the Turbo Pascal and Delphireserved words, Free Pascal also considers the following as reserved words:

```
dispose         false           true
exit            new
```

### 1.3.4 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.

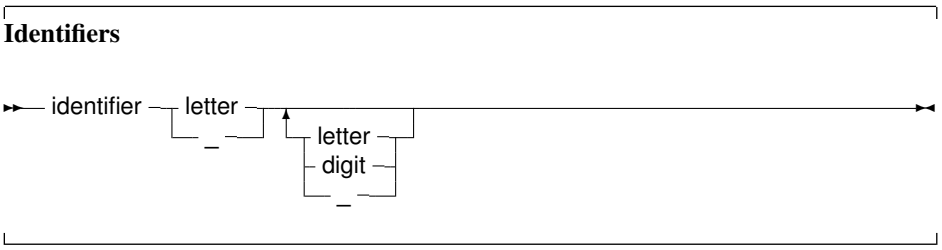
```
absolute        external        nostackframe    read
abstract        far             oldfpccall      register
alias           far16           override        safecall
assembler       forward         pascal          softfloat
cdecl           index           private         stdcall
cppdecl         local           protected       virtual
default         name            public          write
export          near            published
```

**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.

## 1.4 Identifiers

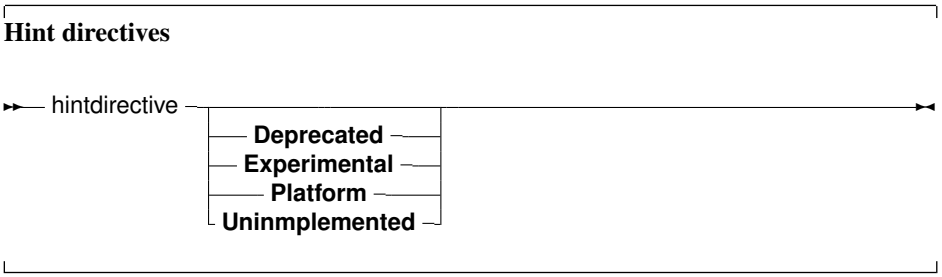
Identifiers denote constants, types, variables, procedures and functions, units, and programs. All names of things that are defined somehow in the source code are identifiers.

An identifier consists of at most 255 significant characters (letters, digits and the underscore character), from which the first must be an alphanumeric character, or an underscore (_) The following diagram gives the basic syntax for identifiers.




## 1.5 Hint directives

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




Whenever an identifier marked with a hint directive is used in the rest of the source code, then a warning will be displayed, corresponding to the hint that was specified.

**deprecated** The use of this identifier is deprecated, use an alternative instead.

**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.

**Uninmplemented** 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.

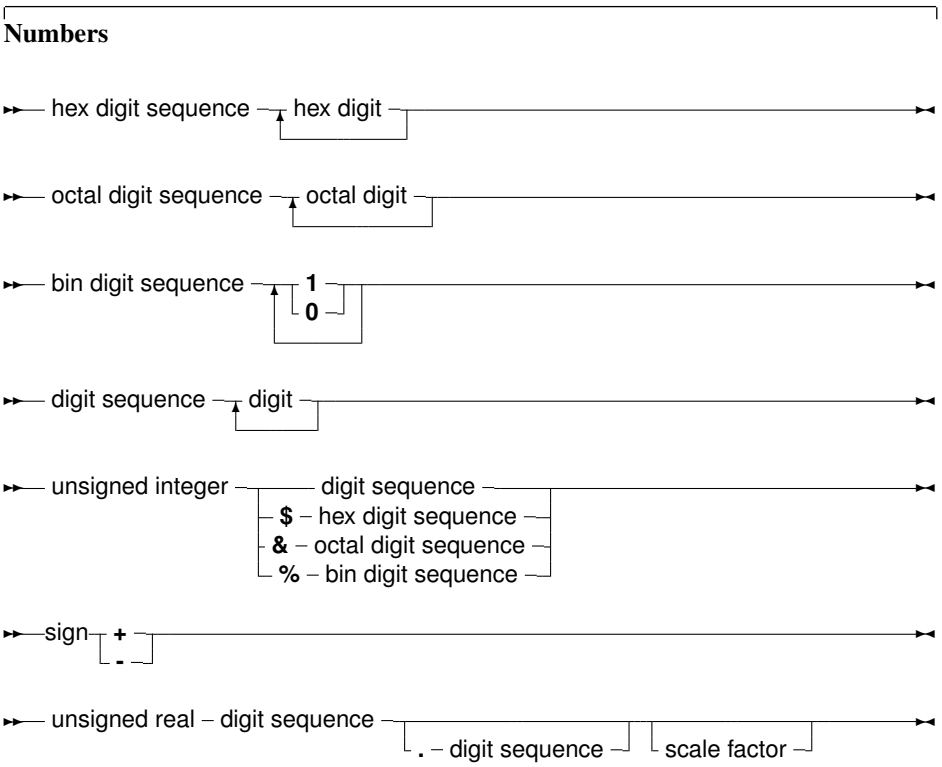
## 1.6 Numbers

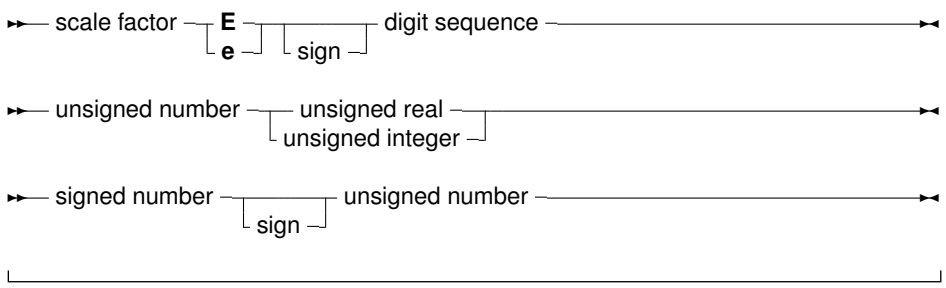
Numbers are by default denoted in decimal notation. Real (or decimal) numbers are written using engineering or scientific notation (e.g. `0.314E1`).

For integer type constants, Free Pascal supports 4 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 a 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:** All decimal constants which do no fit within the -2147483648..2147483647 range are silently and automatically parsed as 64-bit integer constants as of version 1.9.0. Earlier versions would convert it to a real-typed constant.

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

## 1.7 Labels

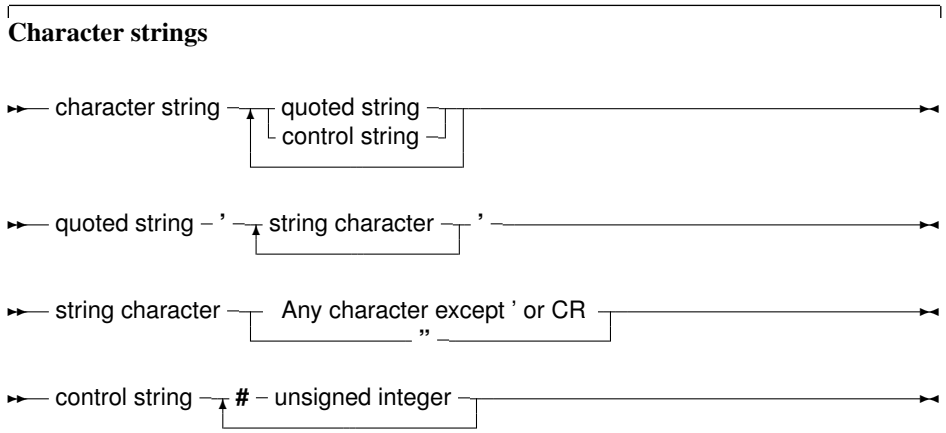
A label is a name for a location in the source code to which can be jumped from another location with a `Goto` statement. Labels can be digit sequences or identifiers.




**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.

## 1.8 Character strings

A character string (or string for short) is a sequence of zero or more characters (byte sized), enclosed in single quotes, and on 1 line of the program source. A character set with nothing between the quotes (`''`) is an empty string.



The string consists of standard, 8-bit ASCII 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.

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.

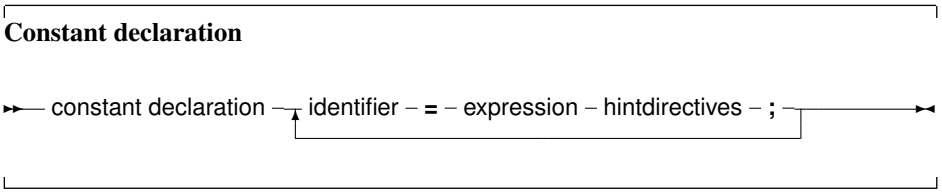
# Chapter 2

# Constants

Just as in Turbo Pascal, Free Pascal supports both normal and typed constants.

## 2.1 Ordinary constants

Ordinary constants declarations are not different from the Turbo Pascal or Delphi implementation.




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 9, page 84. Only constants of the following types can be declared: `Ordinal types`, `Real types`, `Char`, and `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.}
  s = chr(32)
  ls = SizeOf(Longint);
```

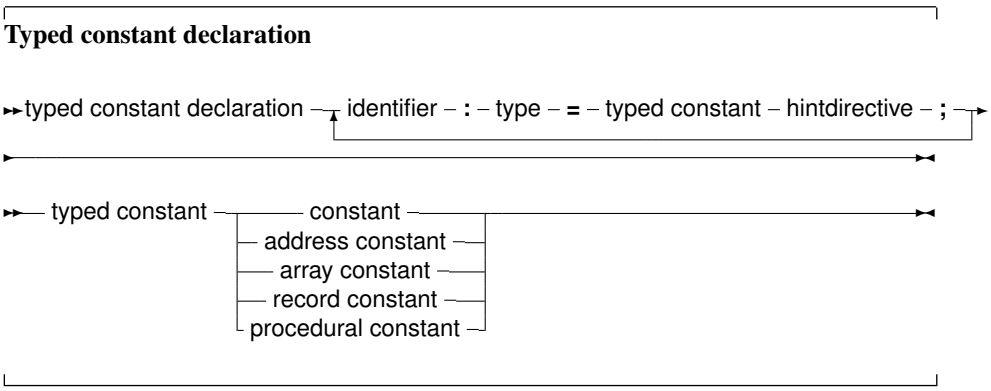
Assigning a value to an ordinary constant is not permitted. Thus, given the previous declaration, the following will result in a compiler error:

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

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

## 2.2 Typed constants

Typed constants serve to provide a program with initialised variables. Contrary to ordinary constants, a value can be assigned to them at run-time. Their definition is quite simple:




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 46.

Support for 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 initialized at program start. This is also true for *local* typed constants. 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 part is the `Resourestring` part. This part is like a `Const` section, but it only allows to declare constant of type string. This part is only available 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, allowing to manipulate the values of the strings at runtime with some special mechanisms.

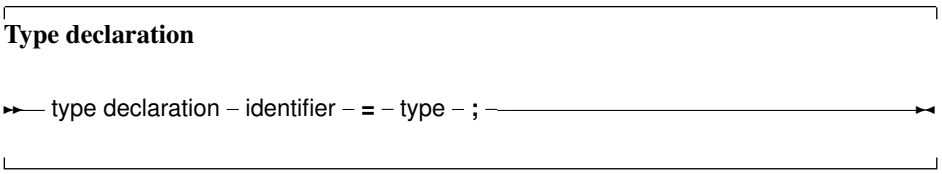
Semantically, the strings are like constants; Values can not be assigned to them, except through the special mechanisms in the objpas unit. However, they can be used in assignments or expressions as normal constants. The main use of the resourcestring section is to provide an easy means of internationalization.

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

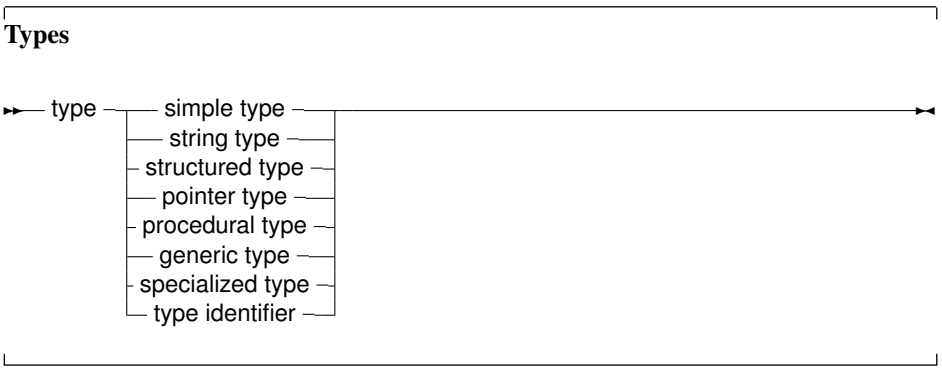
# Chapter 3

# Types

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



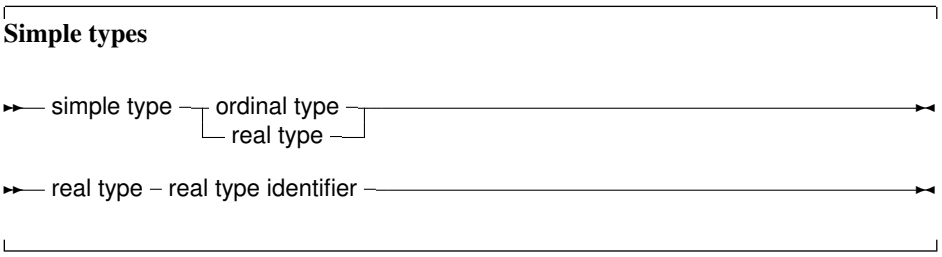

There are 7 major type classes :




The last case, type identifier, is just a means to give another name to a type. This presents a way to make types platform independent, by only using these types, and then defining these types for each platform individually. The programmer that uses these units doesn't have to worry about 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`.

## 3.1 Base types

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




### 3.1.1 Ordinal types

With the exception of `int64`, `qword` and Real 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 bye 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.

#### Integers

A list of pre-defined 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 |
| Boolean |
| ByteBool |
| LongBool |
| Char |

The integer types, and their ranges and sizes, that are predefined in Free Pascal are listed in table (3.2). Please note that the `qword` and `int64` types are not true ordinals, so some Pascal constructs will not work with these two integer types.

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

Table 3.2: Predefined integer types

| 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 |

Free Pascal does automatic type conversion in expressions where different kinds of integer types are used.

### Boolean types

Free Pascal supports the `Boolean` type, with its two pre-defined 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. Free Pascal also supports

Table 3.3: Boolean types

| Name | Size | Ord(True) |
|---|---|---|
| Boolean | 1 | 1 |
| ByteBool | 1 | Any nonzero value |
| WordBool | 2 | Any nonzero value |
| LongBool | 4 | Any nonzero value |

the `ByteBool`, `WordBool` and `LongBool` types. These are of type `Byte`, `Word` or `Longint`, but are assignment compatible with a `Boolean`: 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 -1 in case it is assigned to a variable of type `LongBool`.

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.

```
...
```

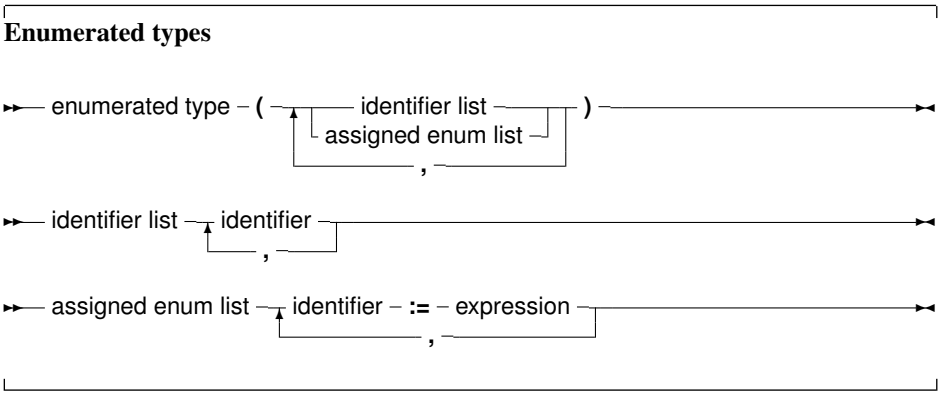
```
B := False;
A := B and Func;
```

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

This behaviour 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 9, page 84 for how to use expressions) When using assigned enumerated types, the assigned elements must be in ascending numerical order in the list, or the compiler will complain. 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);
```

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.

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, three, forty := 40, thirty := 30);
```

It is necessary to keep `forty` and `thirty` in the correct order. When using enumeration types it is important to keep the following points in mind:

1. The `Pred` and `Succ` functions cannot be used on this kind of enumeration types. Trying to do this anyhow will result in a compiler error.

2. Enumeration types are stored using a default, independent of the actual number of values: the compiler does not try to optimize for space. This behaviour 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
```

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;
```

### 3.1.2 Real 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 Single or 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 (3.4). The `Comp` type is, in effect, a 64-bit integer and

Table 3.4: Supported Real 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 | -2E64+1 .. 2E63-1 | 19-20 | 8 |
| Currency | -922337203685477.5808 | 922337203685477.5807 | 8 |

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 minimalizes rounding errors.

## 3.2 Character types

### 3.2.1 Char

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'`.

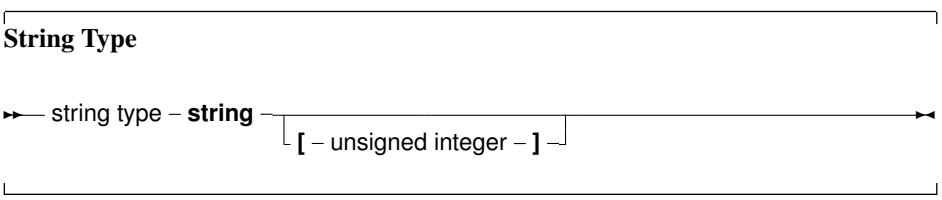
Also, the caret character (`^`) can be used in combination with a letter to specify a character with ASCII value less than 27. Thus `^G` equals `#7` - G is the seventh letter in the alphabet.

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

### 3.2.2 Strings

Free Pascal supports the `String` type as it is defined in Turbo Pascal: a sequence of characters with an optional size specification. It also supports ansistrings (with unlimited length) as in Delphi.

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



If there is a size specifier, then its maximum value - indicating the maximum size of the string - is 255.

The meaning of a string declaration statement without size indicator is interpreted differently depending on the `{$H}` switch. If no size indication is present, the above declaration can declare an ansistring or a short string.

Whatever the actual type, ansistrings and short 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.

### 3.2.3 Short strings

A string declaration declares a short string in the following cases:

1. If the switch is off: `{$H-}`, the string declaration will always be a short string declaration.
2. If the switch is on `{$H+}`, and there is a maximum length (the size) specifier, the declaration is a short string declaration.

The predefined type `ShortString` is defined as a string of size 255:

```
ShortString = String[255];
```

If the size of the string is not specified, `255` is taken as a default. The actual length of the string can be obtained with the `Length` standard runtime routine. For example in

```
{$H-}

Type
  NameString = String[10];
  StreetString = String;
```

`NameString` can contain a maximum of 10 characters. While `StreetString` can contain up to 255 characters.

**Remark:** Short strings have a maximum length of 255 characters: when specifying a maximum length, the maximum length may not exceed 255. If a length larger than 255 is attempted, then the compiler will give an error message:

```
Error: string length must be a value from 1 to 255
```

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. Despite this, to write portable code, it is best to set the length of a shortstring with the `SetLength` call, and to retrieve it with the `Length` call. These functions will always work, whatever the internal representation of the shortstrings or other strings in use: this allows easy switching between the various string types.

### 3.2.4 Ansistrings

Ansistrings are strings that have no length limit. 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.

This is all handled transparantly, i.e. they can be manipulated as a normal short string. Ansistrings can be defined using the predefined `AnsiString` type.

**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 `{$H}` switch is on, then a string definition using the regular `String` keyword and that doesn't contain a length specifier, will be regarded as an ansistring as well. If a length specifier is present, a short string will be used, regardless of the `{$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 typecasted 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 anstrings that are part of a structure (arrays, records or objects).

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 quaranteed 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.

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

```
L:=ord(S[0]);
```

which was valid for Turbo Pascal shortstrings, is no longer correct for Ansistrings. The compiler will warn if such a construct is encountered.

To set the length of an ansistring, the `SetLength` function must be used. Constant ansistrings have a reference count of -1 and are treated specially, The same remark as for `Length` must be given: The construct

```
L:=12;
S[0]:=Char(L);
```

which was valid for Turbo Pascal shortstrings, is no longer correct for Ansistrings. The compiler will warn if such a construct is encountered.

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 typecasted 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 typecasted to a pointer, the pointer wil be `Nil`. If an empty ansistring is typecasted 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.

### 3.2.5 WideStrings

Widestrings (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 `WideStrings` as for `AnsiStrings`. The compiler transparantly converts WideStrings to AnsiStrings and vice versa.

Similarly to the typecast of an Ansistring to a `PChar` null-terminated array of characters, a WideString can be converted to a `PWideChar` null-terminated array of characters. Note that the `PWideChar` 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 widestring manager record, which can be initialized with some OS-specific unicode handling routines. For more information, see the system unit reference.

### 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 is represented by 2 single-quote characters next to each other. 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 substracted, 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`. The strings unit contains procedures and functions that manipulate the `PChar` type as in the standard C library. 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[30];
    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. Null-terminated strings cannot be added as normal Pascal strings. If two `PChar` strings must be concatenated; the functions from the unit strings must be used.

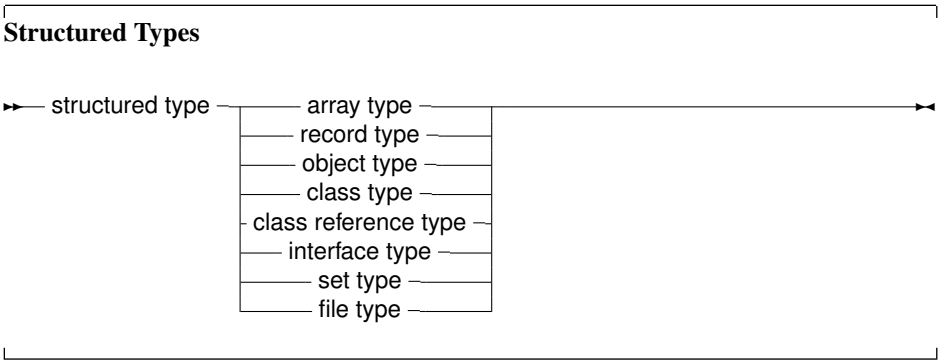
However, it is possible to do some pointer arithmetic. The operators + and − can be used to do operations on `PChar` pointers. In table (3.5), `P` and `Q` are of type `PChar`, and `I` is of type `Longint`.

Table 3.5: `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` | Substracts `I` from the address pointed to by `P`. |
| `P - Q` | Returns, as an integer, the distance between 2 addresses (or the number of characters between `P` and `Q`) |

## 3.3 Structured Types

A structured type is a type that can hold multiple values in one variable. Stuctured types can be nested to unlimited levels.




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 is 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 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 is 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:

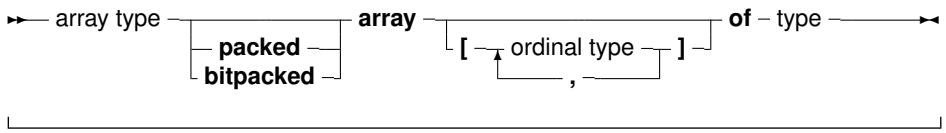
- On 32 bit systems the maximal size is $2^{29}$ bytes (512 MB).
- On 64 bit systems the maximal size is $2^{61}$ bytes.

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:

**Array types**




### 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 2 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.

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, mirrorred.

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.

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 6 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.

#### Packing and unpacking an array

Arrays can be packed and bitpacked. 2 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:

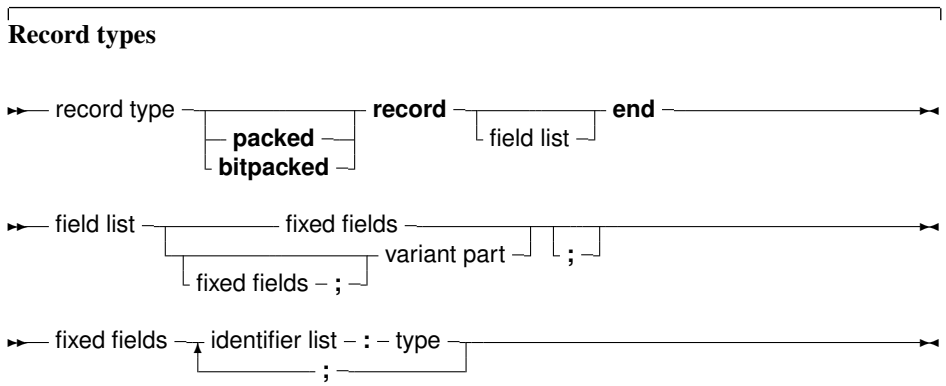
```
Var
  foo : array [ 'a'..'f' ] of Boolean
    = ( false, false, true, false, false, false );
  bar : packed array [ 42..47 ] of Boolean;
  baz : array [ '0'..'5' ] of Boolean;

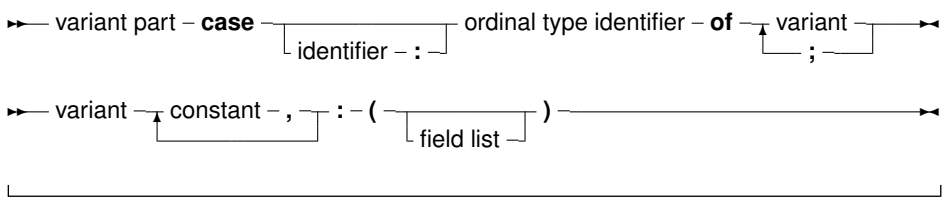
begin
  pack(foo,'a',bar);
  unpack(bar,baz,'0');
end.
```

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 types 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[1]. In effect, it introduces a new field in the record.

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

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

By default the size of a record is the sum of the sizes of its fields, each size of a field is rounded up to a power of two. If the record contains a variant part, the size of the variant part is the size of the biggest variant, plus the size of the tag field type *if an identifier was declared for it*. Here also, the size of each part is first rounded up to two. So in the above example:

- `SizeOf` would return 24 for `Point`,
- It would result in 24 for `RPoint`
- Finally, 26 would be the size of `BetterRPoint`.
- For `MyRec`, the value would be 12.

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

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 for this is that by default, elements of a record are aligned at 2-byte boundaries, for performance reasons.

This default behaviour can be changed with the `{$PACKRECORDS N}` switch. Possible values for `N` are 1, 2, 4, 16 or `Default`. This switch tells the compiler to align elements of a record or object or class that have size larger than `n` on `n` byte boundaries.

Elements that have size smaller or equal than `n` are aligned on natural boundaries, i.e. to the first power of two that is larger than or equal to the size of the record element.

The keyword `Default` selects the default value for the platform that the code is compiled for (currently, this is 2 on all platforms) Take a look at the following program:

```
Program PackRecordsDemo;
type
   {$PackRecords 2}
     Trec1 = Record
       A : byte;
       B : Word;
     end;

     {$PackRecords 1}
     Trec2 = Record
       A : Byte;
       B : Word;
       end;
   {$PackRecords 2}
     Trec3 = Record
       A,B : byte;
     end;

    {$PackRecords 1}
     Trec4 = Record
       A,B : Byte;
       end;
   {$PackRecords 4}
     Trec5 = Record
       A : Byte;
       B : Array[1..3] of byte;
       C : byte;
     end;

     {$PackRecords 8}
     Trec6 = Record
       A : Byte;
       B : Array[1..3] of byte;
       C : byte;
       end;
   {$PackRecords 4}
     Trec7 = Record
       A : Byte;
       B : Array[1..7] of byte;
       C : byte;
     end;
```

```
     {$PackRecords 8}
     Trec8 = Record
       A : Byte;
       B : Array[1..7] of byte;
       C : byte;
       end;
Var rec1 : Trec1;
    rec2 : Trec2;
    rec3 : TRec3;
    rec4 : TRec4;
    rec5 : Trec5;
    rec6 : TRec6;
    rec7 : TRec7;
    rec8 : TRec8;

begin
  Write ('Size Trec1 : ',SizeOf(Trec1));
  Writeln (' Offset B : ',Longint(@rec1.B)-Longint(@rec1));
  Write ('Size Trec2 : ',SizeOf(Trec2));
  Writeln (' Offset B : ',Longint(@rec2.B)-Longint(@rec2));
  Write ('Size Trec3 : ',SizeOf(Trec3));
  Writeln (' Offset B : ',Longint(@rec3.B)-Longint(@rec3));
  Write ('Size Trec4 : ',SizeOf(Trec4));
  Writeln (' Offset B : ',Longint(@rec4.B)-Longint(@rec4));
  Write ('Size Trec5 : ',SizeOf(Trec5));
  Writeln (' Offset B : ',Longint(@rec5.B)-Longint(@rec5),
           ' Offset C : ',Longint(@rec5.C)-Longint(@rec5));
  Write ('Size Trec6 : ',SizeOf(Trec6));
  Writeln (' Offset B : ',Longint(@rec6.B)-Longint(@rec6),
           ' Offset C : ',Longint(@rec6.C)-Longint(@rec6));
  Write ('Size Trec7 : ',SizeOf(Trec7));
  Writeln (' Offset B : ',Longint(@rec7.B)-Longint(@rec7),
           ' Offset C : ',Longint(@rec7.C)-Longint(@rec7));
  Write ('Size Trec8 : ',SizeOf(Trec8));
  Writeln (' Offset B : ',Longint(@rec8.B)-Longint(@rec8),
           ' Offset C : ',Longint(@rec8.C)-Longint(@rec8));
end.
```

The output of this program will be :

```
Size Trec1 : 4 Offset B : 2
Size Trec2 : 3 Offset B : 1
Size Trec3 : 2 Offset B : 1
Size Trec4 : 2 Offset B : 1
Size Trec5 : 8 Offset B : 4 Offset C : 7
Size Trec6 : 8 Offset B : 4 Offset C : 7
Size Trec7 : 12 Offset B : 4 Offset C : 11
Size Trec8 : 16 Offset B : 8 Offset C : 15
```

And this is as expected:

- In `Trec1`, since `B` has size 2, it is aligned on a 2 byte boundary, thus leaving an empty byte between `A` and `B`, and making the total size 4. In `Trec2`, `B` is aligned on a 1-byte boundary, right after `A`, hence, the total size of the record is 3.

- For `Trec3`, the sizes of `A,B` are 1, and hence they are aligned on 1 byte boundaries. The same is true for `Trec4`.
- For `Trec5`, since the size of B – 3 – is smaller than 4, `B` will be on a 4-byte boundary, as this is the first power of two that is larger than its size. The same holds for `Trec6`.
- For `Trec7`, `B` is aligned on a 4 byte boundary, since its size – 7 – is larger than 4. However, in `Trec8`, it is aligned on a 8-byte boundary, since 8 is the first power of two that is greater than 7, thus making the total size of the record 16.

Free Pascal supports also the 'packed record', this is a record where all the elements are byte-aligned. Thus the two following declarations are equivalent:

```
{$PackRecords 1}
Trec2 = Record
  A : Byte;
  B : Word;
  end;
{$PackRecords 2}
```

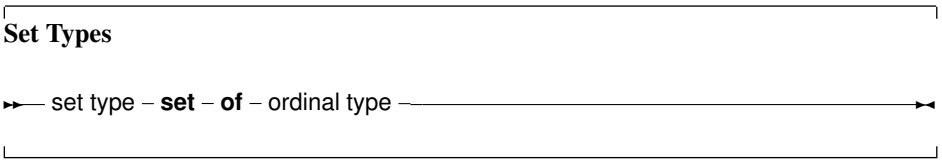
and

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

Note the `{$PackRecords 2}` after the first declaration !

### 3.3.3 Set types

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




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 `255` elements. The following are valid set declaration:

```
Type
  Junk = Set of Char;

  Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  WorkDays : Set of days;
```

Given these declarations, the following assignment is legal:

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

The operators and functions for manipulations of sets are listed in table (3.6).

Two sets can be compared with the <> and = operators, but not (yet) with the < and > operators.

The compiler stores small sets (less than 32 elements) in a Longint, if the type range allows it. This allows for faster processing and decreases program size. Otherwise, sets are stored in 32 bytes.

Table 3.6: Set Manipulation operators

| Operation | Operator |
|---|---|
| Union | + |
| Difference | - |
| Intersection | * |
| Add element | `include` |
| Delete element | `exclude` |

### 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:




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:

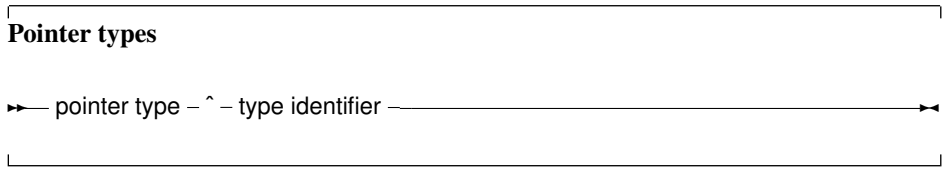
```
Type
  Point = Record
    X,Y,Z : real;
    end;
  PointFile = File of Point;
```

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:



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 23-rd 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` with 4, because 4 is the size of a longint.

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 substracted *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'.

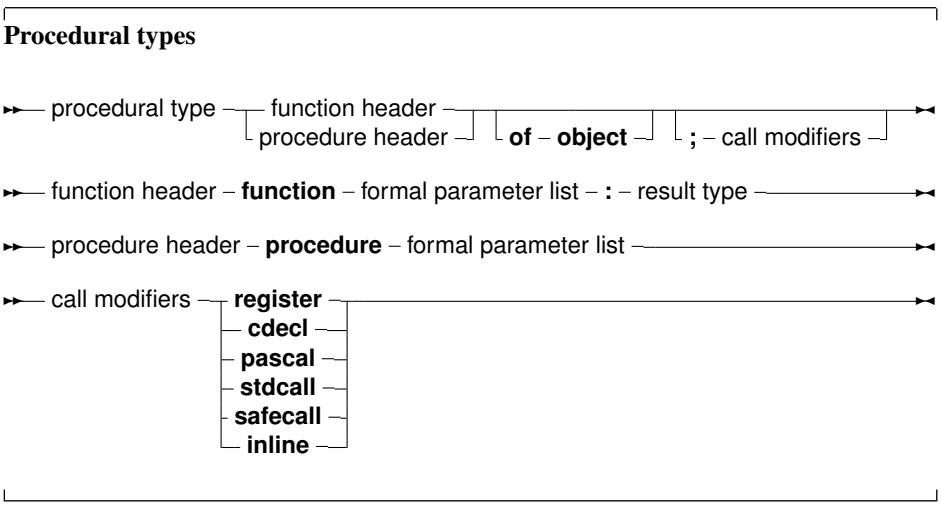
The referenced type should be introduced later in the same `Type` block. No other block may come between the definition of the pointer type and the referenced type. Indeed, even the word `Type` itself may not re-appear: in effect it would start a new type-block, causing the compiler to resolve all pending declarations in the current block.

In most cases, the definition of the referenced type will follow immediatly 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 Delphiimplementation 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 11, page 109. 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.

## 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 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 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.

# 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.

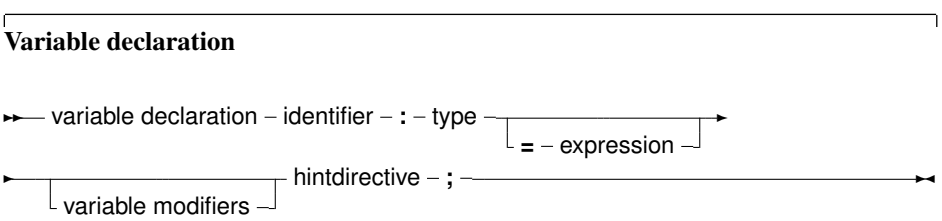
The Free Pascal compiler handles the allocation of these memory locations transparantly, although this location can be influenced in the declaration.

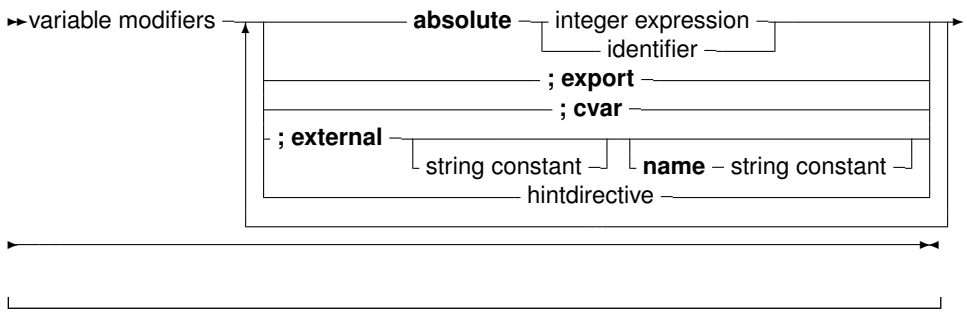
The Free Pascal compiler also handles reading values from or writing values to the variables transparantly. But even this can be explicitly handled by the programmer when using properties.

Variables must be explicitly declared when they are needed. No memory is allocated unless a variable is declared. Using an 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 section of a unit or a procedure or function. 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 ;
```

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.
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 the name of the variable (case sensitive) and 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 elevents form declares a variable (`curterm11`) and initializes it with a value (1 in the above case).

Note that assembler names must be unique. It's not possible to declare or export 2 variables with the same assembler name.

## 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 behaviour 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, variables in pascal are not initialized after their declaration. Any assumption that they contain 0 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.

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 initial elements must be specified, surrounded by round brackets, and separated by commas. The number of 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 should be specified, in the form `Field: Value`, separated by semicolons, and surrounded by round brackets. As an example:

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

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 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.

## 4.5 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.

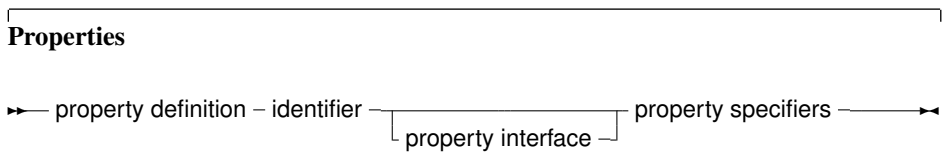
## 4.6 Properties

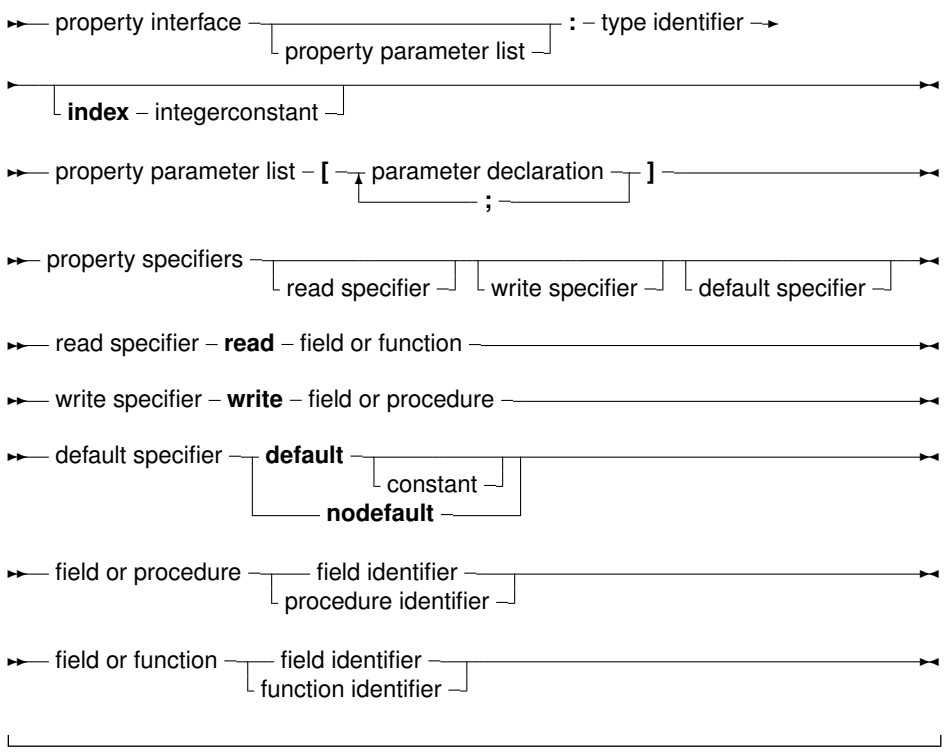
A global block can declare properties, just as they could be defined in a class. The difference is that the global property does not need a class instance: there is only 1 instance of this property. Other than that, a global property behaves like a class property. The read/write specifiers for the global 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:






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 can be hidden by declaring them in another unit which must be in the `uses` clause of the unit. This can be used to hide the read/write access specifiers for programmers, just as if they were in a `private` section of a class (discussed below). For the previous example, this could look as follows:

```
{$mode objfpc}
unit testrw;

Interface

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

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('Only even values are allowed');
  FMyInt:=Value;
end;

end.
```

The unit testprop would then look like:

```
{$mode objfpc}
unit testprop;

Interface

uses testrw;

Property
  MyProp : Integer Read GetMyInt Write SetMyInt;
```

```
Implementation

end.
```

More information about properties can be found in chapter 6, page 60.

# 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.

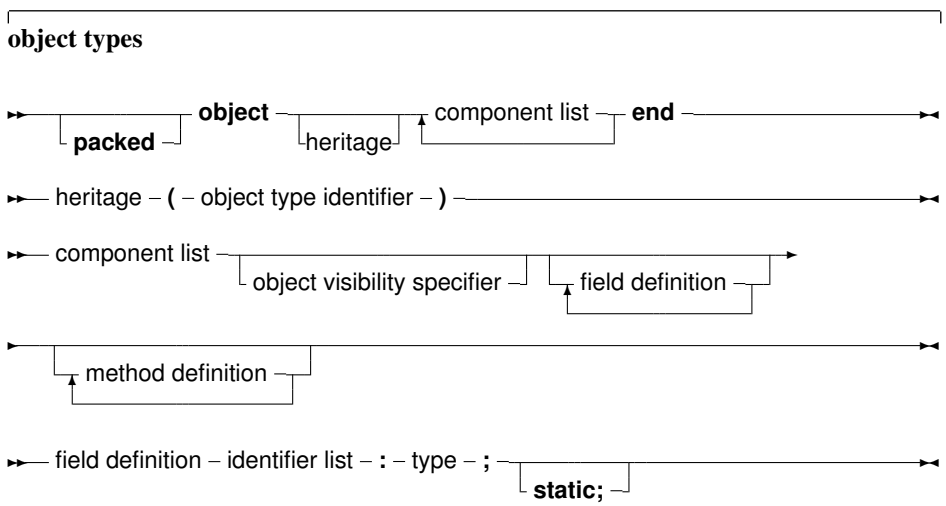
Objects should be treated as a special kind of record. The record contains all the fields that are declared in the objects definition, and pointers to the methods that are associated to the objects' type.

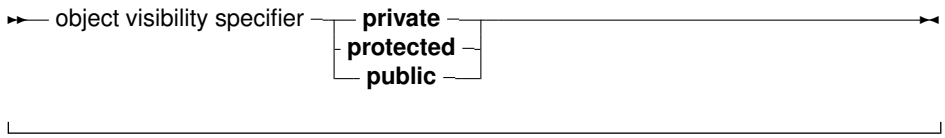
An object is declared just as a record would be declared; except that now, procedures and functions can be declared as if they were part of the record. Objects can ”inherit” fields and methods from ”parent” objects. This means that these fields and methods can be used as if they were included in the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

**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 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 10.2.7, page 105.

## 5.3 Static fields

When the `{$STATIC ON}` directive is active, then an object can contain static fields: these fields are global to the object type, and act like global variables, but are known only as part of the object. 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:

```
{$static on}
type
  cl=object
    l : longint;static;
  end;
var
  c1,c2 : cl;
begin
  c1.l:=2;
  writeln(c2.l);
  c2.l:=3;
  writeln(c1.l);
  Writeln(cl.l);
end.
```

will be the following

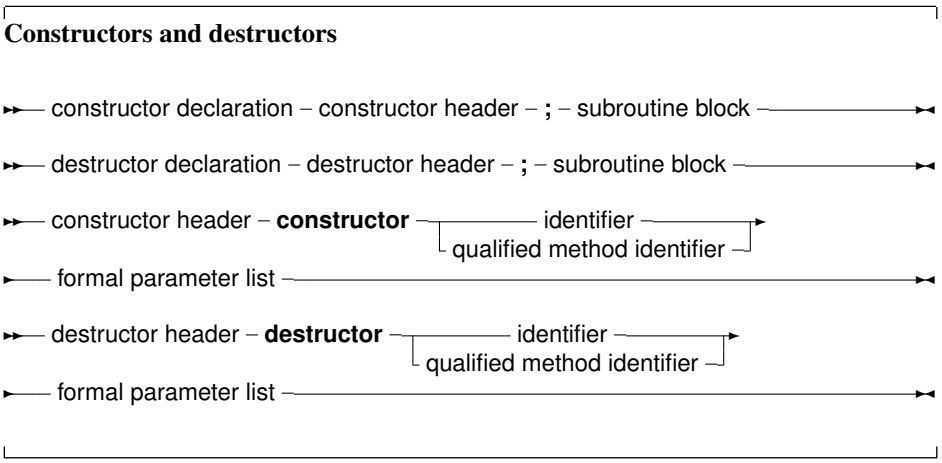
```
2
3
3
```

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

## 5.4 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 constuctor or destructor. When the constructor or destructor is implemented, A qualified method identifier should be used, 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 3 calls are equivalent:

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

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

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

`Dispose` procedure accepts the name of a destructor. The destructor will then be called, before removing the object from the heap.

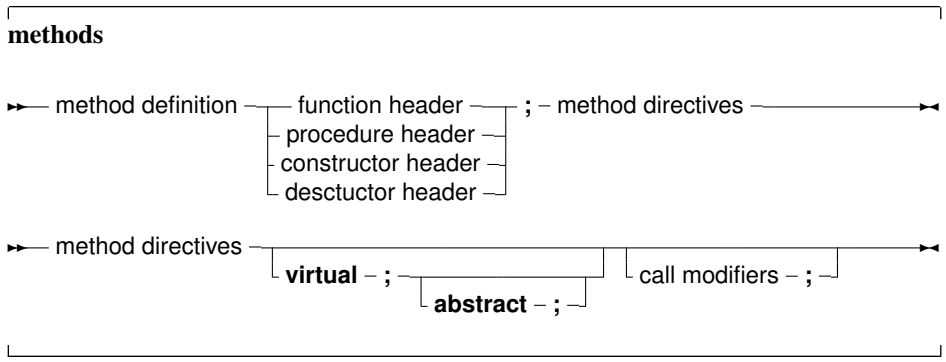
In view of the compiler warning remark, the following chapter presents the Delphi approach to object-oriented programming, and may be considered a more natural way of object-oriented programming.

## 5.5 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.

### 5.5.1 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
    Procedure Doit;
    Field : Longint;
  end;
```

### 5.5.2 Method invocation

Methods are called just as normal procedures are called, only they have an object instance identifier prepended to them (see also chapter 10, page 97). To determine which method is called, it is necessary to know the type of the method. We treat the different types in what follows.

#### Static methods

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 acces 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)

## 5.6 Visibility

For objects, 3 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.

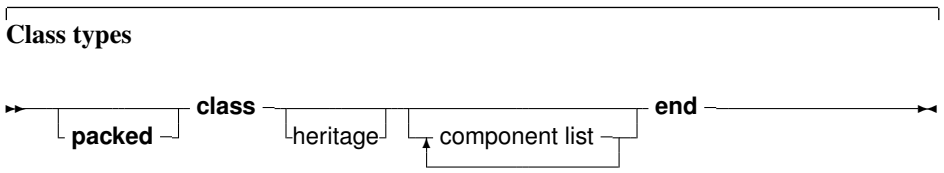
**Remark:** In earlier versions of Free Pascal it was necessary, in order to use classes, to put the objpas unit in the uses clause of a unit or program. *This is no longer needed* as of version 0.99.12. As of this version, the unit will be loaded automatically when the `-MObjfpc` or `-MDelphi` options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

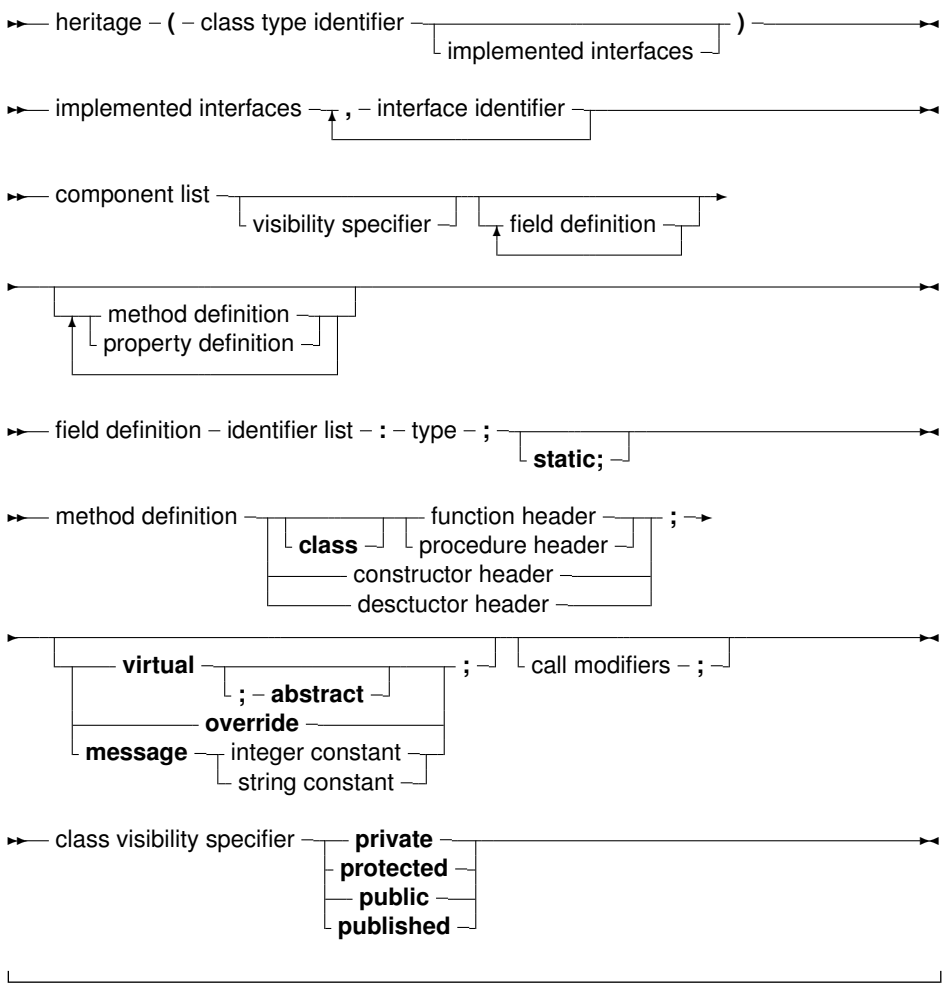
```
{$mode objfpc}
{$mode delphi}
```

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:






As many `private`, `protected`, `published` and `public` blocks as needed can be repeated. 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)

**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** sections are always accessible.

**Published** Is the same as a `Public` section, but the compiler generates also type information that is needed for automatic streaming of these classes. 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).

Similar to objects, if the `{$STATIC ON}` directive is active, then a class can contain static fields: these fields 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.

For instance, the output of the following program:

```
{$mode objfpc}
{$static on}
type
  cl=class
    l : longint;static;
  end;
var
  c1,c2 : cl;
begin
  c1:=cl.create;
  c2:=cl.create;
  c1.l:=2;
  writeln(c2.l);
  c2.l:=3;
  writeln(c1.l);
  Writeln(cl.l);
end.
```

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`).

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 speficied 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 if `TMaxClass`.

More about instantiating a class can be found in the next section.

## 6.2 Class instantiation

Classes 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 object 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 `getmem`, to allocate enough space to hold the class instance data. After that, the constuctor's code is executed. The constructor has a pointer to its data, in `Self`.

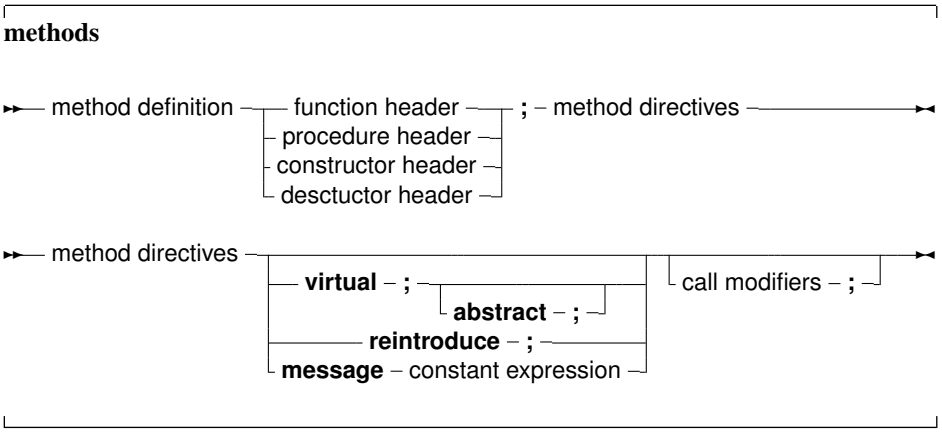
**Remark:**

- 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 but a pointer to an object. To get the size of the class instance data, use the `TObject.InstanceSize` method.

## 6.3 Methods

### 6.3.1 Declaration

Declaration of methods in classes follows the same rules as method declarations in objects:




### 6.3.2 invocation

Method invocation for classes is no different than for objects. The following is a valid method invocation:

```
Var  AnObject : TAnObject;
begin
  AnObject := TAnObject.Create;
  ANobject.AMethod;
```

### 6.3.3 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
  ObjParent = Class
    Procedure MyProc; virtual;
  end;
  ObjChild  = Class(ObjPArent)
    Procedure MyProc; virtual;
  end;
```

The compiler will produce a warning:

```
Warning: An inherited method is hidden by OBJCHILD.MYPROC
```

The compiler will compile it, but using `Inherited` can produce strange effects.

The correct declaration is as follows:

```
Type
  ObjParent = Class
    Procedure MyProc; virtual;
  end;
  ObjChild  = Class(ObjPArent)
    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
  ObjParent = Class
    Procedure MyProc; virtual;
  end;
  ObjChild  = Class(ObjPArent)
    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.

### 6.3.4 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;

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 cannot be used as read or write specifiers for a property.

### 6.3.5 Message methods

New in classes are `message` methods. Pointers to message methods are stored in a special table, together with the integer or string cnstant 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;
```

In addition to this mechanism, a string message method accepts a self parameter:

```
Procedure StrMsgHandler(Data: Pointer;
                        Self: TMyObject); Message 'OnClick';
```

When encountering such a method, the compiler will generate code that loads the `Self` parameter into the object instance pointer. The result of this is that it is possible to pass `Self` as a parameter to such a method.

**Remark:** The type of the `Self` parameter must be of the same class as the class the method is defined in.

### 6.3.6 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.

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);

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`.

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.

## 6.4 Properties

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 for 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.

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 2 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 paramater 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).

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.

Array properties also exist. These are properties that accept an index, just as an array does. Only now the index 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 arguent 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 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, and descendent classes cannot redeclare the default property.

The *stored specifier* should be either a boolean constant, a boolean field of the class, or a parameterless function which returns a boolean result. This specifier has no result on the class behaviour. It is an aid for the streaming system: the stored specifier is specified in the RTTI generated for a class (it can only be streamed if RTTI is generated), and is used to determine whether a property should be streamed or not: it saves space in a stream. It is not possible to specify the 'Stored' directive for array properties.

The *default specifier* can be specified for ordinal types and sets. It serves the same purpose as the *stored specifier*: Properties that have as value their default value, will not be written to the stream by

the streaming system. The default value is stored in the RTTI that is generated for the class. Note that

1. When the class is instantiated, the default value is not automatically applied to the property, it is the responsability 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.

# Chapter 7

# Interfaces

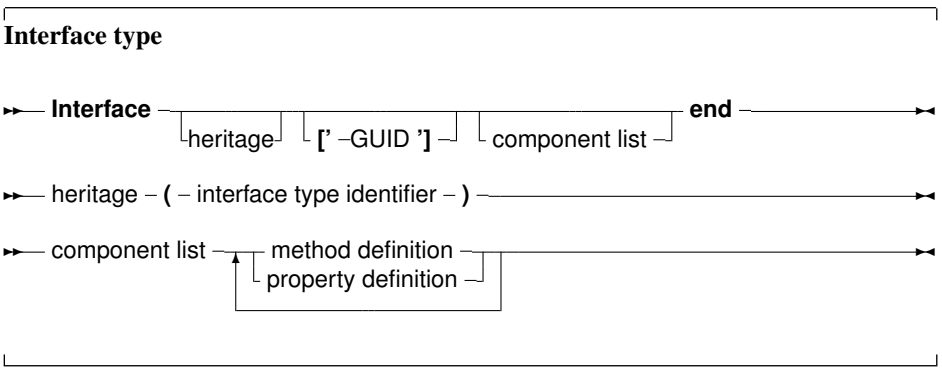
## 7.1 Definition

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 definition of a 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;
          );
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
```

At the moment of writing, the compiler does not yet support providing aliases for an interface as in Delphi. i.e. the following will not yet compile:

```
ype
  IMyInterface = Interface
    Function MyFunc : Integer;
  end;
```

```
TMyClass = Class(TInterfacedObject,IMyInterface)
  Function MyOtherFunction : Integer;
  // The following fails in FPC.
  Function IMyInterface.MyFunc = MyOtherFunction;
end;
```

This declaration should tell the compiler that the `MyFunc` method of the `IMyInterface` interface is implemented in the `MyOtherFunction` method of the `TMyClass` class.

## 7.4 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.

## 7.5 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.

# Chapter 8

# Generics

## 8.1 Introduction

Generics are templates for generating classes. 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.

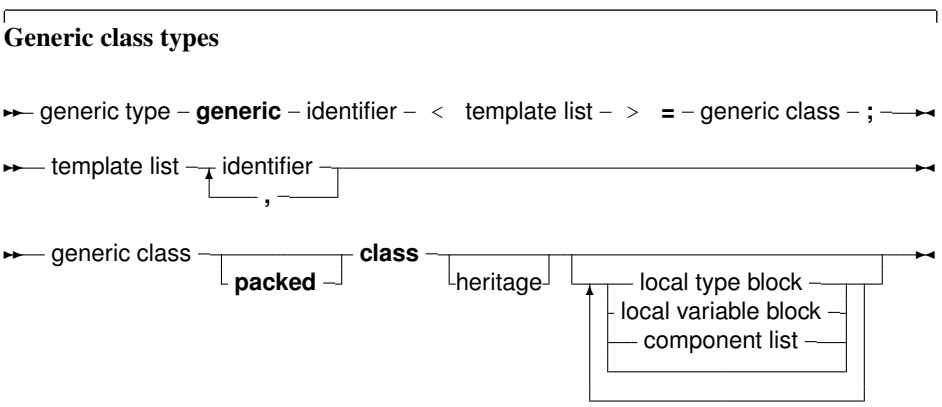
Currently, only generic classes can be defined. Later, support for generic records, functions and arrays may be introduced.

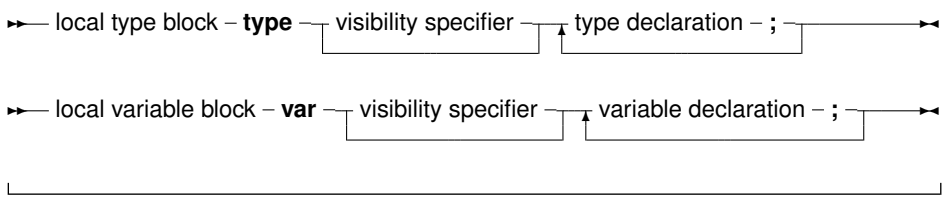
Creating and using generics is a 2-phase process.

1. The definition of the generic class 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 class is specialized: this defines a second class, which is a specific implementation of the generic class: the compiler replays the macro which was stored when the generic class was defined.

## 8.2 Generic class definition

A generic class definition is much like a class definition, with the exception that it contains a list of placeholders for types, and can contain a series of local variable blocks or local type blocks, as can be seen in the following syntax diagram:






The generic class declaration should be followed by a class 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.

The generic class declaration is much like a normal class declaration, except for the local variable and local type block. The local type block defines types that are type placeholders: they are not actualized until the class is specialized.

The local variable block is just an alternate syntax for ordinary class fields. The reason for introducing is the introduction of the `Type` block: just as in a unit or function declaration, a class declaration can now have a local type and variable block definition.

The following is a valid generic class definition:

```
Type
  generic TList<_T>=class(TObject)
    type public
       TCompareFunc = function(const Item1, Item2: _T): Integer;
    var public
      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 placehoder. 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:

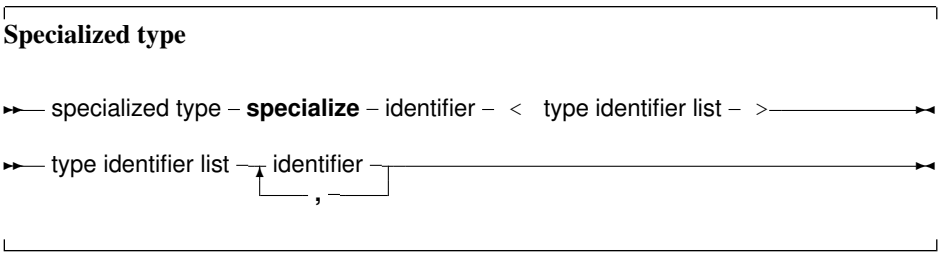
```
generic TList<_T>=class(TObject)
  type public
     TCompareFunc = function(const Item1, Item2: _T): Integer;
Public
  data : _T;
  procedure Add(item: _T);
  procedure Sort(compare: TCompareFunc);
end;
```

4. Both the local variable block and local type block have a visibility specifier. This is optional; if it is omitted, the current visibility is used.

## 8.3 Generic class specialization

Once a generic class is defined, it can be used to generate other classes: this is like replaying the definition of the class, 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:




Which is a very simple definition. 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>;
```

The following is not allowed:

```
Var
  P : specialize TList<Pointer>;
```

that is, a variable cannot be directly declared using a specialization.

The type in the specialize statement must be known. Given the 2 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. However, the following is allowed:

```
type
  TA = specialize TMyFirstType<Atype>;
  TB = specialize TMySecondType<TA>;
```

because `TA` is already fully defined when `TB` is specialized.

Note that 2 specializations of a generic type with the same types in a placeholder are not assignment compatible. In the following example:

```
type
  TA = specialize TList<Pointer>;
  TB = specialize TList<Pointer>;
```

variables of types `TA` and `TB` cannot be assigned to each other, i.e the following assignment will be invalid:

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

begin
  A:=B;
```

**Remark:** It is not possible to make a forward definition of a generic class. The compiler will generate an error if a forward declaration of a class is later defined as a generic specialization.

## 8.4 A word about scope

It should be stressed that all identifiers other than the template placeholders should be known when the generic class is declared. This works in 2 ways. First, all types must be known, that is, a type identifier with the same name must exist. The following unit will produce an error:

```
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
```

The second way in which this is visible, is the following. Assume a unit

```
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.
```

and a program

```
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 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.

# Chapter 9

# Expressions

Expressions occur in assignments or in tests. Expressions produce a value of a certain type. Expressions are built with two components: Operators and their operands. Usually an operator is binary, i.e. it requires 2 operands. Binary operators occur always between the operands (as in `X/Y`). Sometimes an operator is unary, i.e. it requires only one argument. A unary operator occurs always before the operand, as in `-X`.

When using multiple operands in an expression, the precedence rules of table (9.1) are used. When

Table 9.1: Precedence of operators

| Operator | Precedence | Category |
|---|---|---|
| `Not, @` | Highest (first) | Unary operators |
| `* / div mod and shl shr as` | Second | Multiplying operators |
| `+ - or xor` | Third | Adding operators |
| `< <> < > <= >= in is` | Lowest (Last) | relational operators |

determining the precedence, the compiler uses the following rules:

1. In operations with unequal precedences the operands belong to the operater with the highest precedence. For example, in `5*3+7`, the multiplication is higher in precedence than the addition, so it is executed 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.

**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 expression is evaluated first should be made in such a case. The compiler will decide which expression to evaluate first based on optimization rules. Thus, in the following expression:

```
a := g(3) + f(2);
```

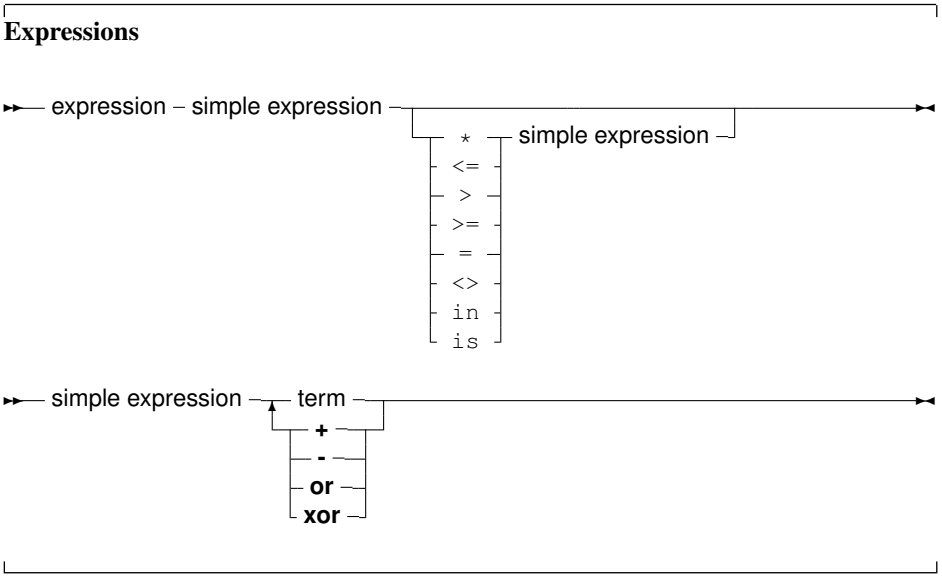
`f(2)` may be executed before `g(3)`. This behaviour is distinctly different from Delphi or Turbo Pascal.

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);
```

## 9.1 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.



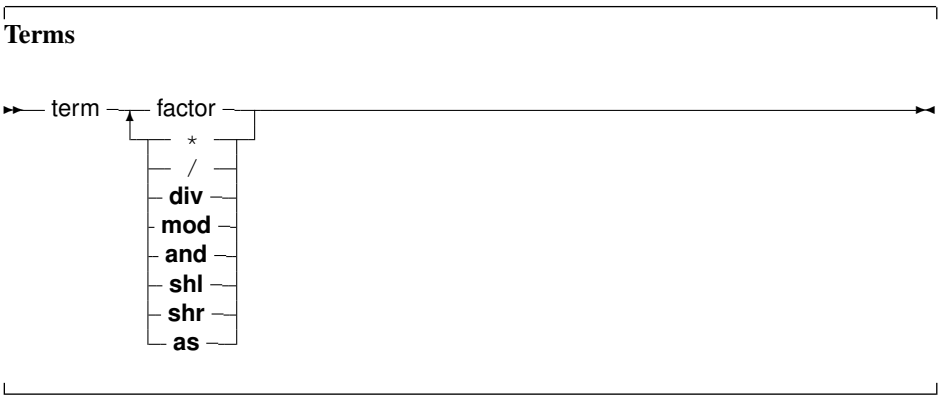

The following are valid expressions:

```
GraphResult<>grError
(DoItToday=Yes) and (DoItTomorrow=No);
Day in Weekend
```

And here are some simple expressions:

```
A + B
-Pi
ToBe or NotToBe
```

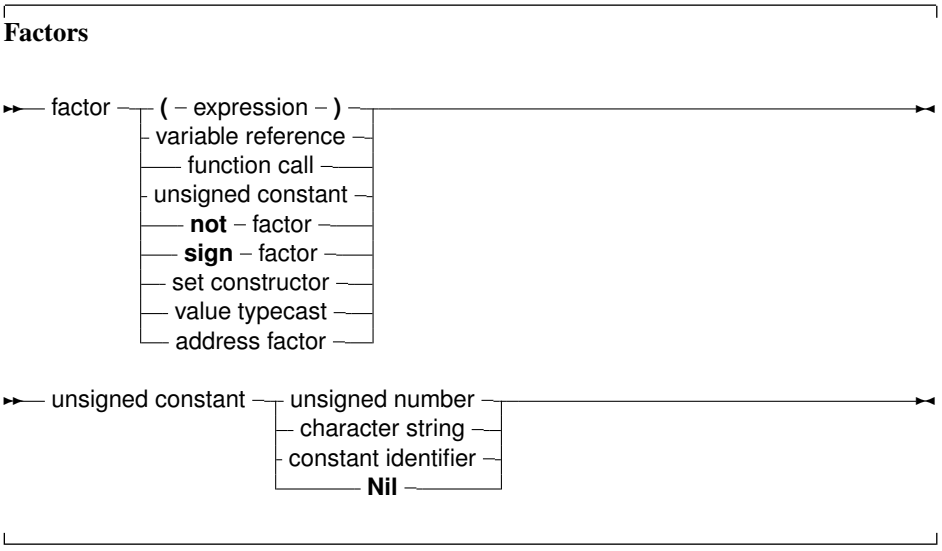
Terms consist of factors, connected by multiplication operators.




Here are some valid terms:

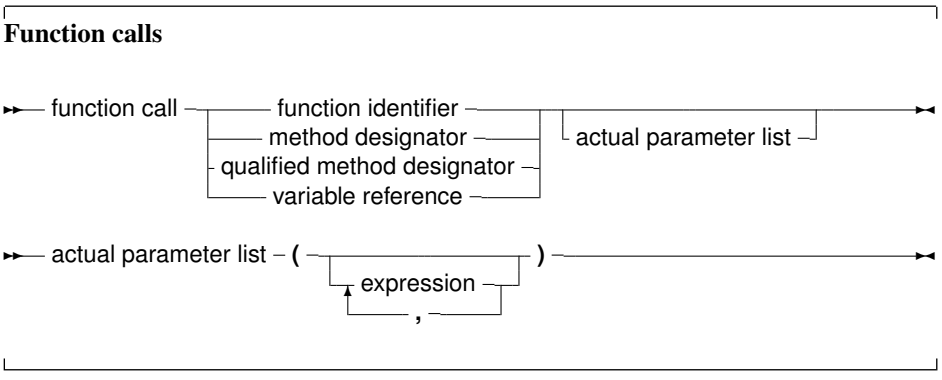
```
2 * Pi
A Div B
(DoItToday=Yes) and (DoItTomorrow=No);
```

Factors are all other constructions:




## 9.2 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:

```
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 assigment to `F` will not cause the function `AddOne` to be called. The assignment to `N`, however, will call `AddOne`.

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? Free Pascal solves this by deciding that a procedural variable is 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, hence a type mismatch occurs.

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');
```

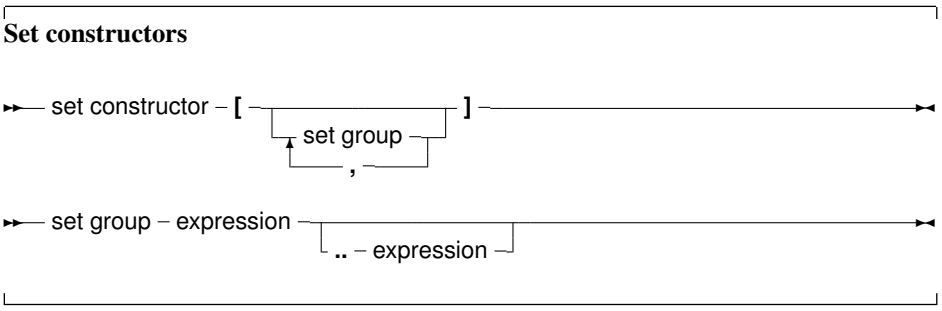
The left hand side of the boolean expression is an address. The right hand side also, and so the compiler compares 2 addresses. How to compare the values that both functions return ? By adding an empty parameter list:

```
  If F()=Addone then
    WriteLn ('Functions return same values ');
```

Remark that this behaviour is not compatible with Delphisyntax. Switching on `Delphi` mode will allow you to use Delphisyntax.

## 9.3 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.



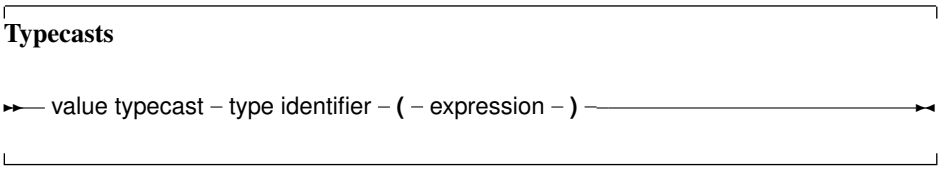

All set groups and set elements must be of the same ordinal type. The empty set is denoted by `[]`, and it can be assigned to any type of set. A set group with a range `[A..Z]` makes all values in the range a set element. The following are valid set constructors:

```
[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.

## 9.4 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:




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, enumerateds) 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 Delphior Turbo Pascal behaviour.

## 9.5 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;
```

Pointer variables can be typecasted 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;
```

## 9.6 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 therefor 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.

Example:

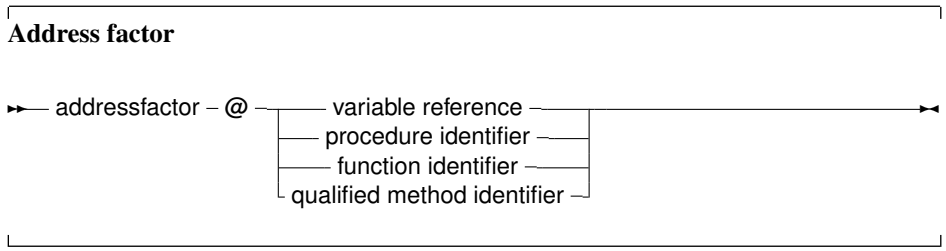
```
program me;

Var
  A : packed Array[1..20] of Byte;
  I : LongInt;
```

```
begin
  For I:=1 to 20 do
    A[i]:=I;
  I:=PInteger(Unaligned(@A[13]))^;
end.
```

## 9.7 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 assigment 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 behaviour can be avoided by using the `-Mtp` or `-MDelphi` switches, which result in a more compatible Delphior Turbo Pascal syntax.

## 9.8 Operators

Operators can be classified according to the type of expression they operate on. We will discuss them type by type.

### 9.8.1 Arithmetic operators

Arithmetic operators occur in arithmetic operations, i.e. in expressions that contain integers or reals. There are 2 kinds of operators : Binary and unary arithmetic operators. Binary operators are listed in table (9.2), unary operators are listed in table (9.3). With the exception of `Div` and `Mod`, which

Table 9.2: Binary arithmetic operators

| Operator | Operation |
|---|---|
| `+` | Addition |
| `-` | Subtraction |
| `*` | Multiplication |
| `/` | Division |
| `Div` | Integer division |
| `Mod` | Remainder |

accept only integer expressions as operands, all operators accept real and integer expressions as operands.

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 9.3: 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.

### 9.8.2 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 (9.4). The following are valid logical expressions:

Table 9.4: 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 |

```
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 }
```

### 9.8.3 Boolean operators

Boolean operators can be considered 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 (9.5)

Table 9.5: 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)

### 9.8.4 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).

### 9.8.5 Set operators

The following operations on sets can be performed with operators: Union, difference, symmetric difference, inclusion and intersection. Elements can be aded or removed from the set with the `Include` or `Exclude` operators. The operators needed for this are listed in table (9.6). The set type of the

Table 9.6: Set operators

| Operator | Action |
|---|---|
| + | Union |
| - | Difference |
| * | Intersection |
| >< | Symmetric difference |
| <= | Contains |
| include | include an element in the set |
| exclude | exclude an element from the set |

operands 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;

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);
end.
```

As can be seen, the union is equivalent to a binary OR, while the intersection is equivalent to a binary AND, and the summetric difference equals a XOR operation.

The `Include` and `Exclude` operations are equivalent to a union or a difference with a set of 1 element. Thus,

```
Include(W,wed);
```

is equivalent to

```
W:=W+[wed];
```

and

```
Exclude(W,wed);
```

is equivalent to

```
W:=W-[wed];
```

### 9.8.6 Relational operators

The relational operators are listed in table (9.7) Normally, left and right operands must be of the same type. There are some notable exceptions, where the compiler can handle mixed expressions:

Table 9.7: Relational operators

| Operator | Action |
|---|---|
| = | Equal |
| <> | Not equal |
| < | Stricty less than |
| > | Strictly greater than |
| <= | Less than or equal |
| >= | Greater than or equal |
| in | Element of |

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`

### 9.8.7 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 2 class operators, as can be seen in table (9.8). An expression containing the `is` operator results in a boolean type. The `is` operator can only

Table 9.8: Class operators

| Operator | Action |
|---|---|
| is | Checks class type |
| as | Conditional typecast |

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:

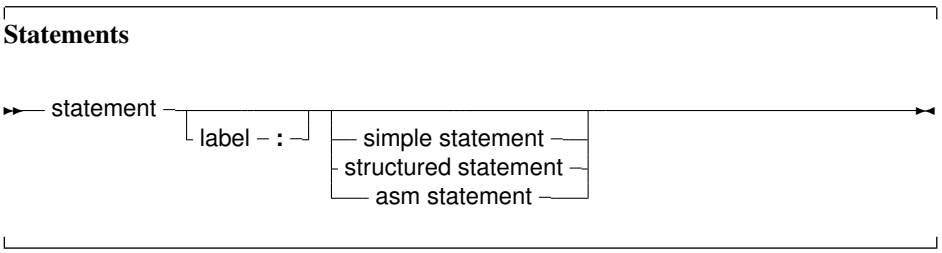
```
Var
  C : TComponent;
  O : TObject;

begin
  (C as TEdit).Text:='Some text';
  C:=O as TComponent;
end;
```

# Chapter 10

# Statements

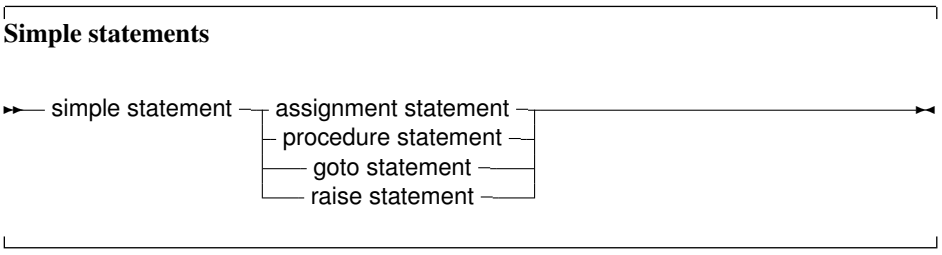
The heart of each algorithm are the actions it takes. These actions are contained in the statements of a program or unit. Each statement can be labeled and jumped to (within certain limits) with `Goto` statements. This can be seen in the following syntax diagram:




A label can be an identifier or an integer digit.

## 10.1 Simple statements

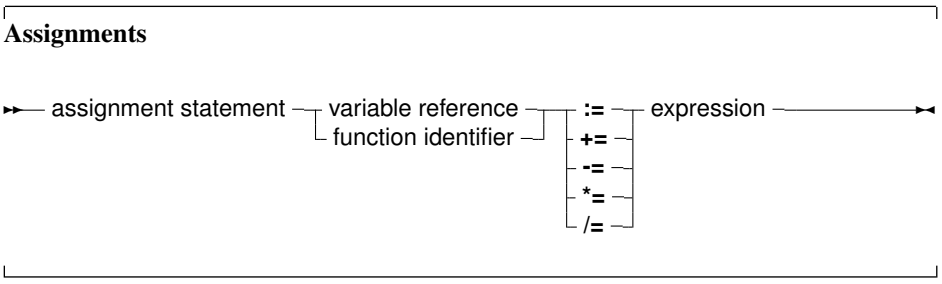
A simple statement cannot be decomposed in separate statements. There are basically 4 kinds of simple statements:




Of these statements, the *raise statement* will be explained in the chapter on Exceptions (chapter 14, page 141)

### 10.1.1 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 varable 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 (10.1).

Table 10.1: Allowed C constructs in Free Pascal

| Assignment | Result |
|---|---|
| a += b | Adds b to a, and stores the result in a. |
| a -= b | Substracts 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. |

For these constructs to work, the `-Sc` command-line switch must be specified.

**Remark:** These constructions are just for typing convenience, they don't generate different code. Here are some examples of valid assignment statements:

```
X := X+Y;
X+=Y;       { Same as X := X+Y, needs -Sc command line switch}
X/=2;       { Same as X := X/2, needs -Sc command line switch}
Done := False;
Weather := Good;
MyPi := 4* Tan(1);
```

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';
```

Note the double dereferencing in the second assignment.

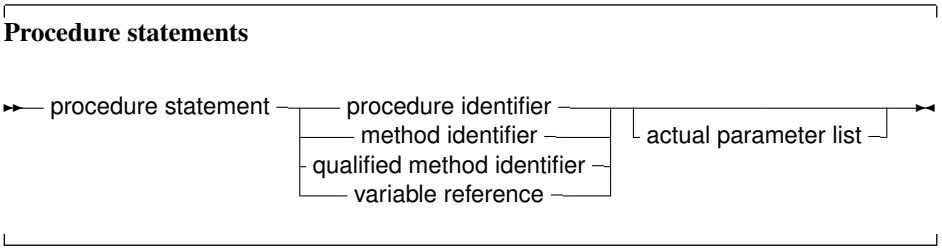
### 10.1.2 Procedure statements

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:



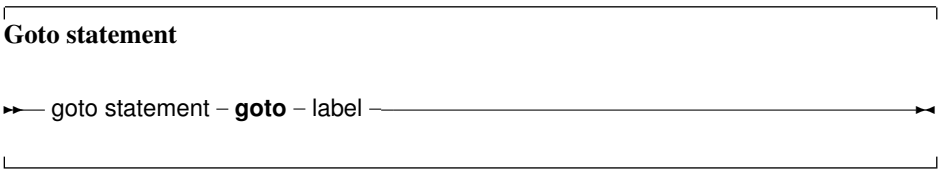

The Free Pascal compiler will look for a procedure with the same name as given in the procedure statement, and with a declared parameter list that matches the actual parameter list. The following are valid procedure statements:

```
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.

### 10.1.3 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.

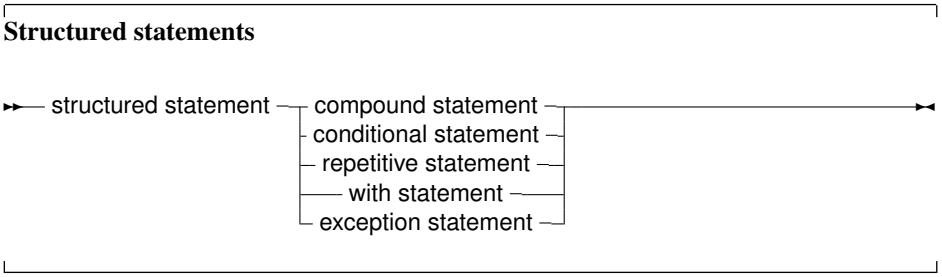
`Goto` statements are considered bad practice and should be avoided as much as possible. It is always possible to replace a `goto` statement by a construction that doesn't need a `goto`, although this construction may not be as clear as a goto statement. For instance, the following is an allowed goto statement:

```
label
  jumpto;
...
```

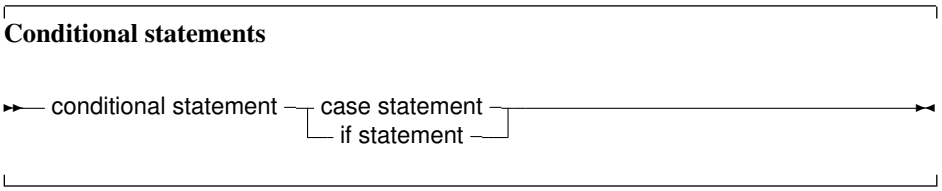
```
Jumpto :
  Statement;
...
Goto jumpto;
...
```

## 10.2 Structured statements

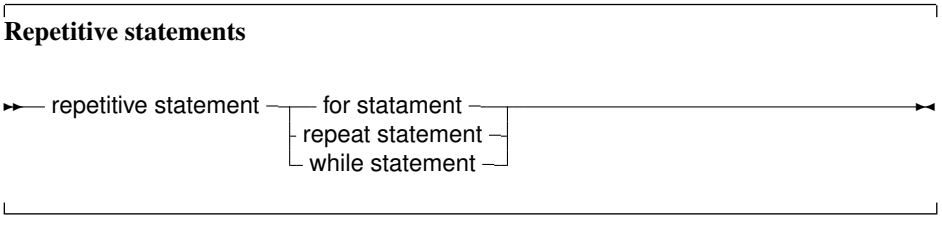
Structured statements can be broken into smaller simple statements, which should be executed repeatedly, conditionally or sequentially:




Conditional statements come in 2 flavours :




Repetitive statements come in 3 flavours:

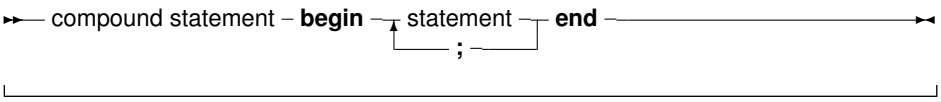



The following sections deal with each of these statements.

### 10.2.1 Compound statements

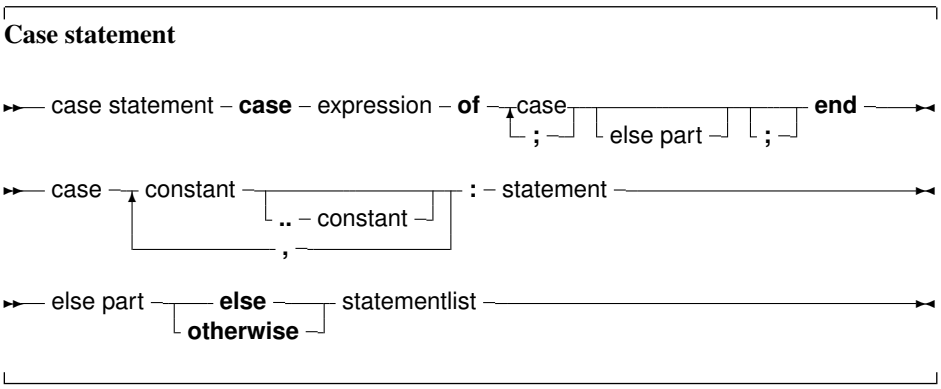
Compound statements are a group of statements, separated by semicolons, that are surrounded by the keywords `Begin` and `End`. The last statement - before the `End` keyword - doesn't need to be followed by a semicolon, although it is allowed. A compound statement is a way of grouping statements together, executing the statements sequentially. They are treated as one statement in cases where Pascal syntax expects 1 statement, such as in `if...then...else` statements.






### 10.2.2 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 (except boolean), and chars. 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).

**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;
```

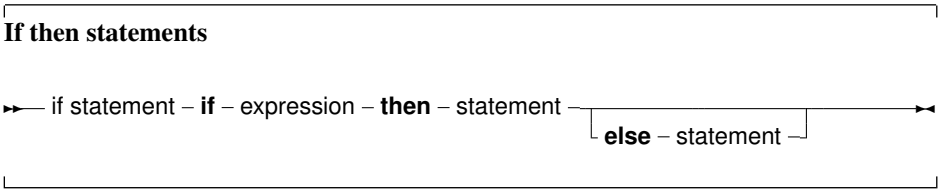
Or

```
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;
```

### 10.2.3 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 araise 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;
```

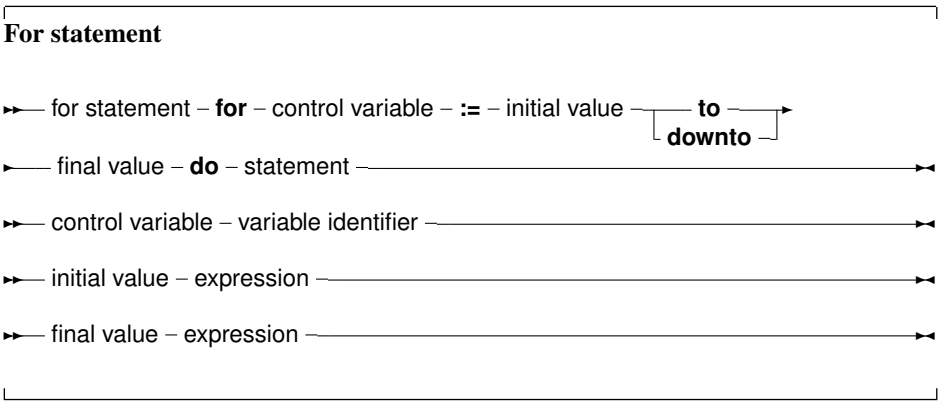
If it is this latter construct which is needed, the `begin` and `end` keywords must be present. When in doubt, it is better to add them.

The following is a valid statement:

```
If Today in [Monday..Friday] then
  WriteLn ('Must work harder')
else
  WriteLn ('Take a day off.');
```

### 10.2.4 The `For..to/downto..do` statement

Free Pascal supports the `For` loop construction. A for loop is used in case one wants to calculated 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 with 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 before initializing the counter variable with the initial value.

**Remark:** It is not allowed to change (i.e. assign a value to) the value of a loop variable inside the loop.

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:

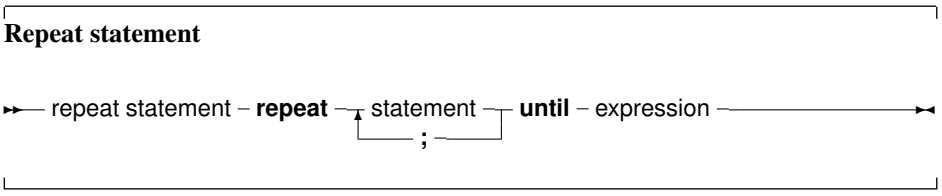
```
For I:=0 to 100 do
  begin
  DoSomething;
  I:=I*2;
  end;
```

because the loop variable `I` cannot be assigned to inside the loop.

If the statement is a compound statement, then the `Break` and `Continue` reserved words can be used to jump to the end or just after the end of the `For` statement.

### 10.2.5 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




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.

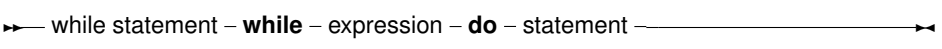
The `Break` and `Continue` reserved words can be used to jump to the end or just after the end of the `repeat .. until` statement.

### 10.2.6 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;
```

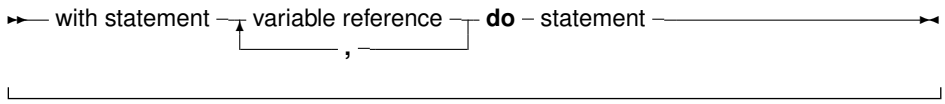
They correspond to the example loops for the `repeat` statements.

If the statement is a compound statement, then the `Break` and `Continue` reserved words can be used to jump to the end or just after the end of the `While` statement.

### 10.2.7 The `With` statement

The `with` statement serves to access the elements of a record or object or class, without having to specify the name of the 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;
```

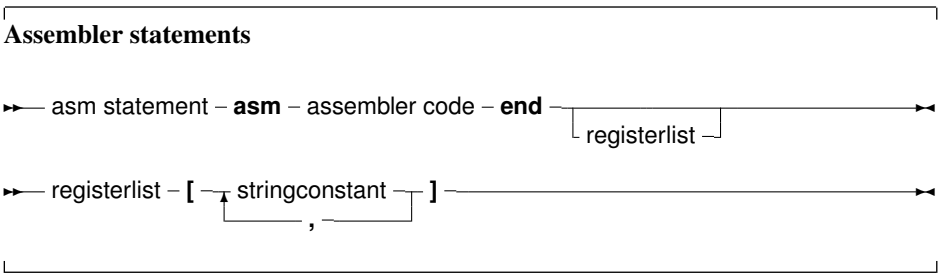
The reason the pointer cannot be changed is that the address is stored by the compiler in a temporary register. Changing the pointer won't change the temporary address. The same is true for classes.

### 10.2.8 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 14, page 141

## 10.3 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 11

# Using functions and procedures

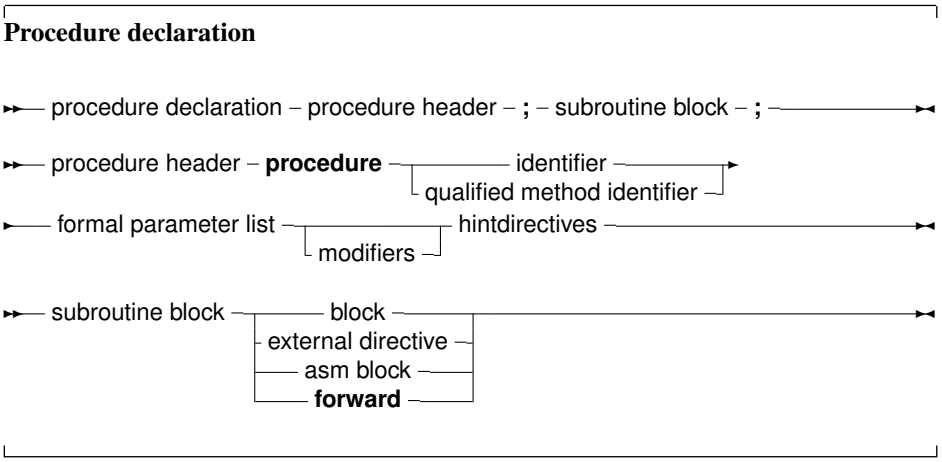
Free Pascal supports the use of functions and procedures. It supports

- 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.

## 11.1 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.



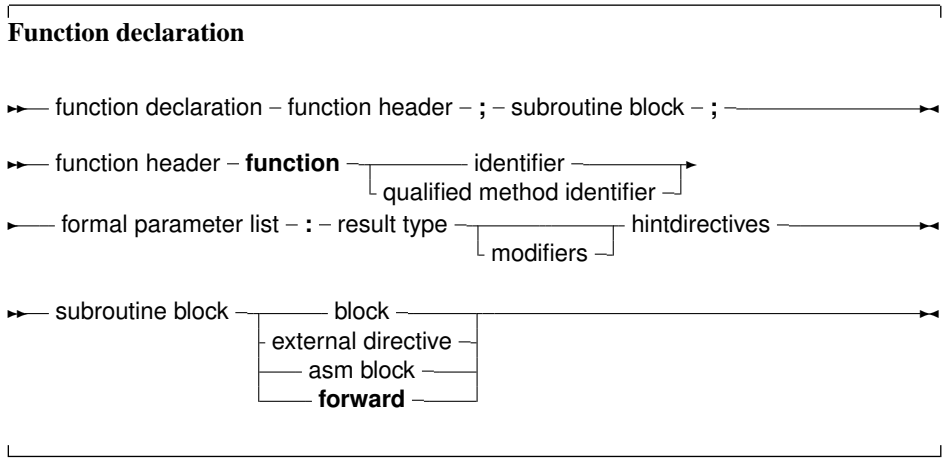

See section 11.4, page 111 for the list of parameters. A procedure declaration that is followed by a block implements the action of the procedure in that block. The following is a valid procedure :

```
Procedure DoSomething (Para : String);
begin
  Writeln ('Got parameter : ',Para);
  Writeln ('Parameter in upper case : ',Upper(Para));
end;
```

Note that it is possible that a procedure calls itself.

## 11.2 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.

## 11.3 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 `ObjPas` mode, the above can also be coded as:

```
Function MyFunction : Integer;

begin
  Result:=12;
end;
```

As an extension to Delphisyntax, the ObjFPC mode also supports a special extension of the `Exit` procedure:

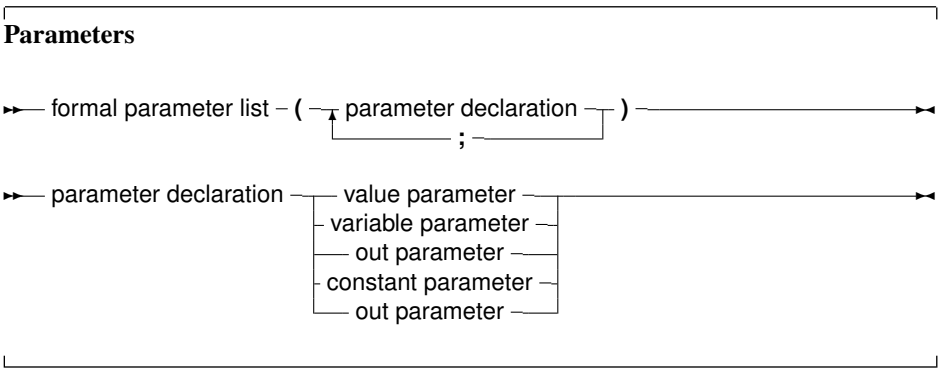
```
Function MyFunction : Integer;

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.

## 11.4 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.

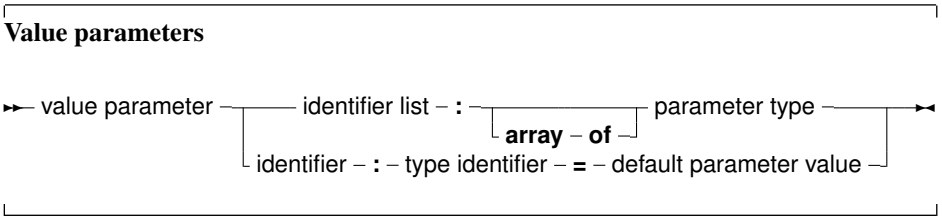



Constant parameters, out parameters and variable parameters can also be `untyped` parameters if they have no type identifier.

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.

### 11.4.1 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 11.4.5, page 114 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;

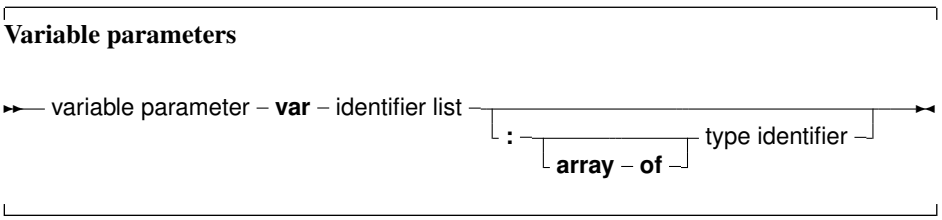
Procedure MyRealFunc(I : Integer = MyConst);

begin
  Writeln('Function received : ',I);
end;

begin
  MyRealFunc;
end.
```

### 11.4.2 Variable parameters

Variable parameters are declared as follows:




When parameters are declared as variable parameters, the procedure or function accesses immediatly 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 can passed to a function that has also an untyped parameter. If an untyped parameter is used in an assigment, or a value must be assigned to it, a typecast must be used.

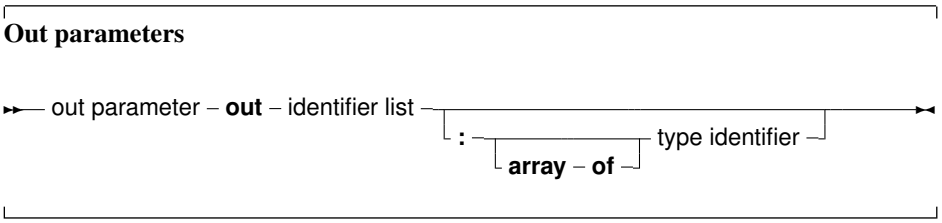
File type variables must always be passed as variable parameters.

Open arrays can be passed as variable parameters. See section 11.4.5, page 114 for more information on using open arrays.

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.

### 11.4.3 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: 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`:

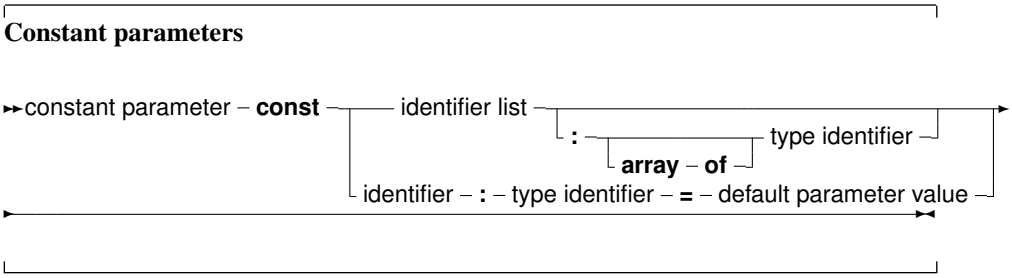
```
home: >fpc -S2 -vwhn testo.pp
testo.pp(19,8) Hint: Variable "C" does not seem to be initialized
```

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.

### 11.4.4 Constant parameters

In addition to variable parameters and value parameters Free Pascal also supports Constant parameters. A constant parameter as can be specified as follows:




A constant argument is passed by reference if its size is larger than a pointer. It is passed by value if the size is equal or is less then the size of a native pointer. This means that the function or procedure receives a pointer to the passed argument, but it cannot be assigned to, this will result in a compiler error. Furthermore a const parameter cannot be passed on to another function that requires a variable parameter. The main use for this is reducing the stack size, hence improving performance, and still retaining the semantics of passing by value...

Constant parameters can also be untyped. See section 11.4.2, page 112 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 11.4.5, page 114 for more information on using open arrays.

### 11.4.5 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(paremeter)`. 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)`. 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]));
```

### 11.4.6 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.

The elements of the array of const are converted to a 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;
```

Therefor, 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].VAnsiStr
    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 that this is not true for Delphi, so code relying on this feature will not be portable.

## 11.5 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 the RTL, the `Dec` procedure could be defined as:

```
...
Dec(Var I : Longint;decrement : Longint);
Dec(Var I : Longint);
```

```
Dec(Var I : Byte;decrement : Longint);
Dec(Var I : Byte);
...
```

When the compiler encounters a call to the `Dec` 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.

The compiler accepts the presence of the `overload` modifier as in Delphi, but it is not required, unless in Delphi mode.

## 11.6 Forward defined 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 defined as forward 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:

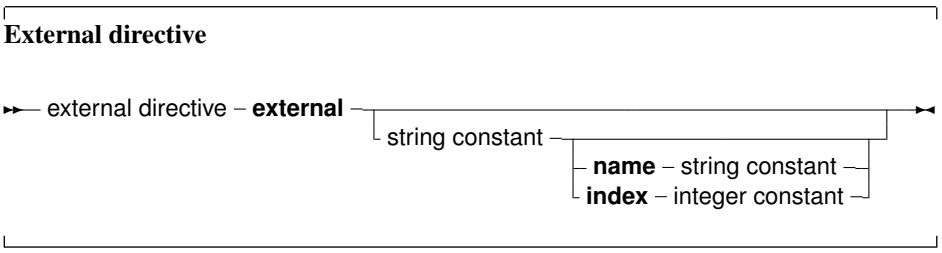
```
Unit testforward;
interface
Procedure First (n : longint);
Procedure Second;
implementation
Procedure First (n : longint); forward;
Procedure Second;
begin
  WriteLn ('In second. Calling first...');
  First (1);
```

```
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.

## 11.7 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.




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 :

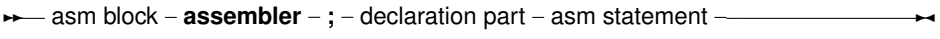
```
external name 'Fname';
{$L myfunc.o}
```

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.

## 11.8 Assembler functions

Functions and procedures can be completely implemented in assembly language. To indicate this, use the `assembler` keyword:

**Assembler functions**

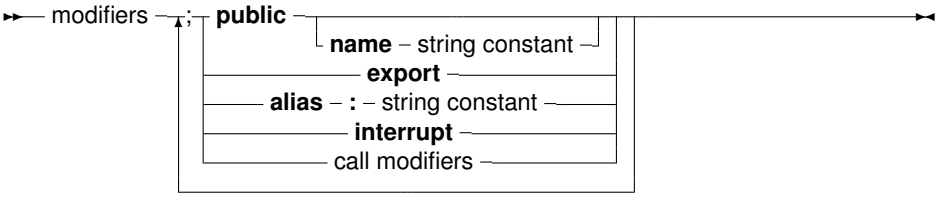


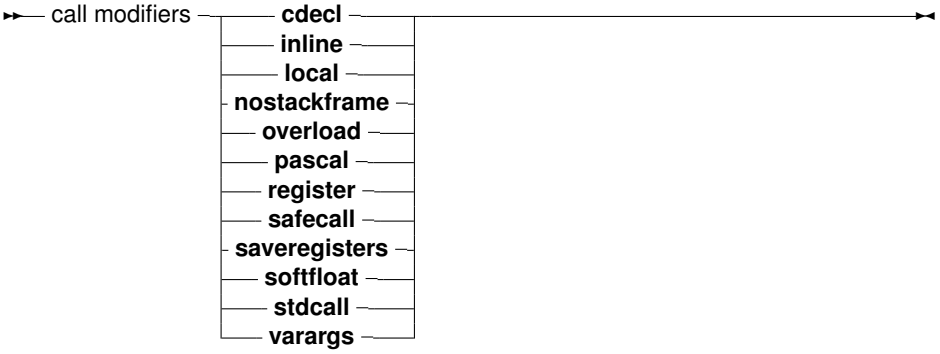

Contrary to Delphi, the assembler keyword must be present to indicate an assembler function. For more information about assembler functions, see the chapter on using assembler in the Programmer's Guide.

## 11.9 Modifiers

A function or procedure declaration can contain modifiers. Here we list the various possibilities:

**Modifiers**



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.

### 11.9.1 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;alias : 'DOIT';
begin
  WriteLn ('In Printit (alias : "DOIT")');
end;
begin
  asm
  call DOIT
  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`.

```
The \var{alias} directive is considered deprecated. Please use the
\var{public name} directive. See \sees{public}.
```

### 11.9.2 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 11.7).

**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 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.

### 11.9.3 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 Programming dynamic libraries in the Programmer's Guide.

### 11.9.4 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 accross units.

3. Recursive inline functions are not allowed. i.e. an inline function that calls itself is not allowed.

### 11.9.5 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.

### 11.9.6 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.

### 11.9.7 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.

One should be extremely careful when using this modifier: most procedures or functions need a stack frame. Particularly for debugging they are needed.

### 11.9.8 overload

The `overload` modifier tells the compiler that this function is overloaded. It is mainly for Delphicompatibility, as in Free Pascal, all functions and procedures can be overloaded without this modifier.

There is only one case 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.

The following example illustrates this. 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.

### 11.9.9 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.

### 11.9.10 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.

### 11.9.11 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. The first three arguments are passed in registers EAX,ECX and EDX.

### 11.9.12 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.

### 11.9.13 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.

### 11.9.14 softfloat

The `softfloat` modifier makes sense only on the ARM architecture.

### 11.9.15 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.

### 11.9.16 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 2 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]);
```

## 11.10 Unsupported Turbo Pascal modifiers

The modifiers that exist in Turbo Pascal, but aren't supported by Free Pascal, are listed in table (11.1). The compiler will give a warning when it encounters these modifiers, but will otherwise completely

Table 11.1: Unsupported modifiers

| Modifier | Why not supported ? |
|---|---|
| Near | Free Pascal is a 32-bit compiler. |
| Far | Free Pascal is a 32-bit compiler. |

ignore them.

# Chapter 12

# Operator overloading

## 12.1 Introduction

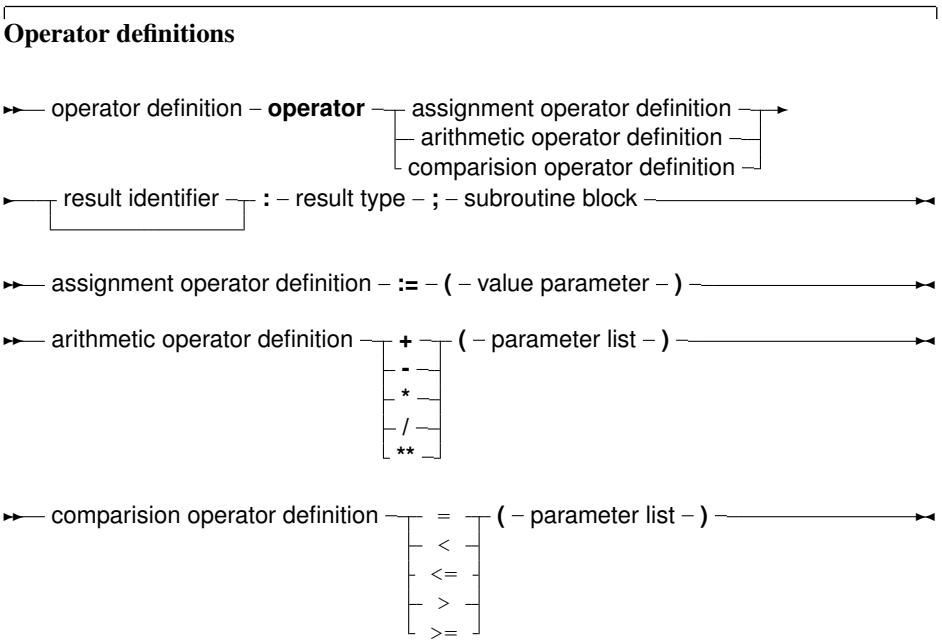
Free Pascal supports operator overloading. This means that it is possible to define the action of some operators on self-defined types, and thus allow the use of these types in mathematical expressions.

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.

## 12.2 Operator declarations

To define the action of an operator is much like defining a function:



The parameter list for a comparision operator or an arithmetic operator must always contain 2 parameters, with the exception of the unary minus, where only 1 parameters is needed. The result type of the comparision operator must be `Boolean`.

**Remark:** When compiling in `Delphi` mode or `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 2 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.

## 12.3 Assignment operators

The assignment operator defines the action of a assignent 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 assigment 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);
```

An experimental solution for this problem exists in the compiler, but is not enabled by default. Maybe someday it will be.

## 12.4 Arithmetic operators

Arithmetic operators define the action of a binary operator. Possible operations are:

**multiplication** To multiply two types, the $\star$ 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.

**substraction** To substract two types, the − substraction operator must be overloaded.

**exponentiation** To exponentiate two types, the $\star\star$ exponentiation operator must be overloaded.

**Unary minus** is used to take the negative of the argument following it.

The definition of an arithmetic operator takes two parameters, except for unary minus, which needs only 1 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 behaviour 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 mutiplication of a `(m,n)` with a `(n,m)` matrix is a `(m,m)` matrix, which needn't be the same in all cases.

## 12.5 Comparision operator

The comparision operator can be overloaded to compare two different types or to compare two equal types that are not basic types. The result type of a comparision operator is always a boolean.

The comparision operators that can be overloaded are:

**equal to** (=) To determine if two variables are equal.

**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.

There is no separate operator for *unequal to* (<>). To evaluate a statement that contans the *unequal to* operator, the compiler uses the *equal to* operator (=), and negates the result.

As an example, the following opetrator allows to compare two complex numbers:

```
operator = (z1, z2 : complex) b : boolean;
```

the above definition allows comparisions of the following form:

```
Var
  C1,C2 : Complex;

begin
  If C1=C2 then
    Writeln('C1 and C2 are equal');
end;
```

The comparision operator definition needs 2 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 2 comparision operators.

In the case of complex numbers, it is, for instance necessary to define 2 comparsions: 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 comparisions 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 comparisions is reversed.

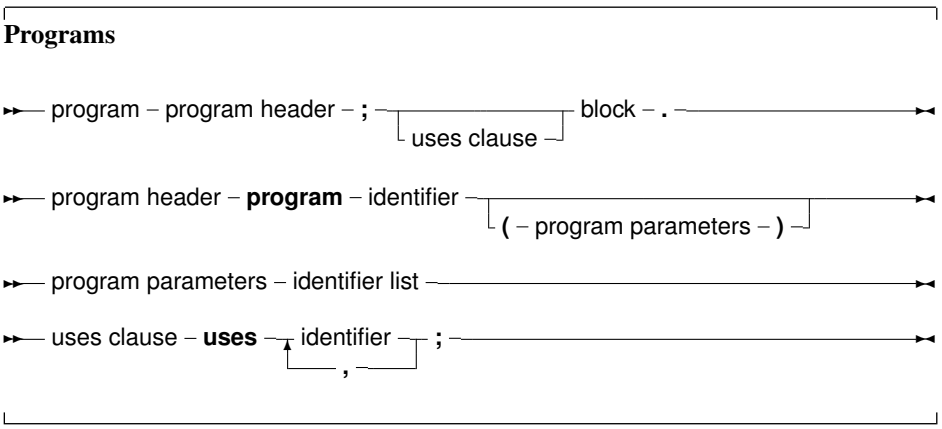
# Chapter 13

# 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.

## 13.1 Programs

A pascal program consists of the program header, followed possibly by a 'uses' clause, and a block.




The program header is provided for backwards compatibility, and is ignored by the compiler.

The uses clause serves to identify all units that are needed by the program. All identifiers which are declared in the 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.

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.

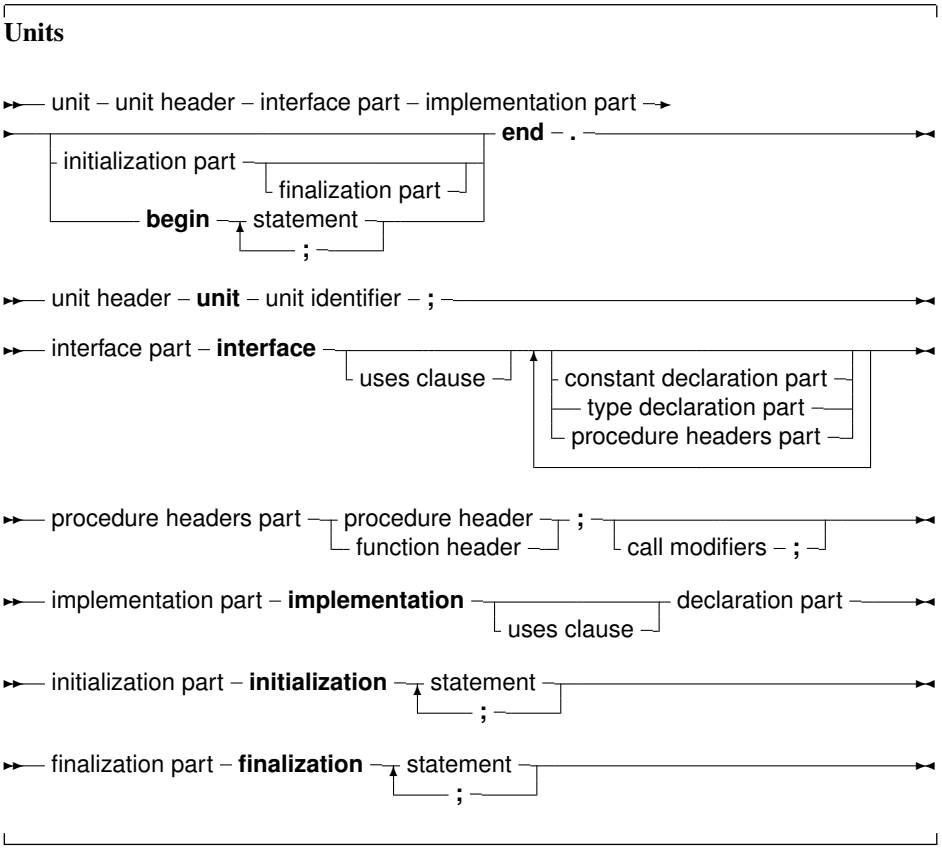
Note that the above search is performed in each directory in the search path.

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.

## 13.2 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.

Both the interface part or implementation part can be empty, but the keywords `Interface` and `implementation` must be specified. 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 it's 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 if a `finalization` block is present, an `Initialization` block must be present, but it can be empty:

```
Initialization

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.
```

## 13.3 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 :

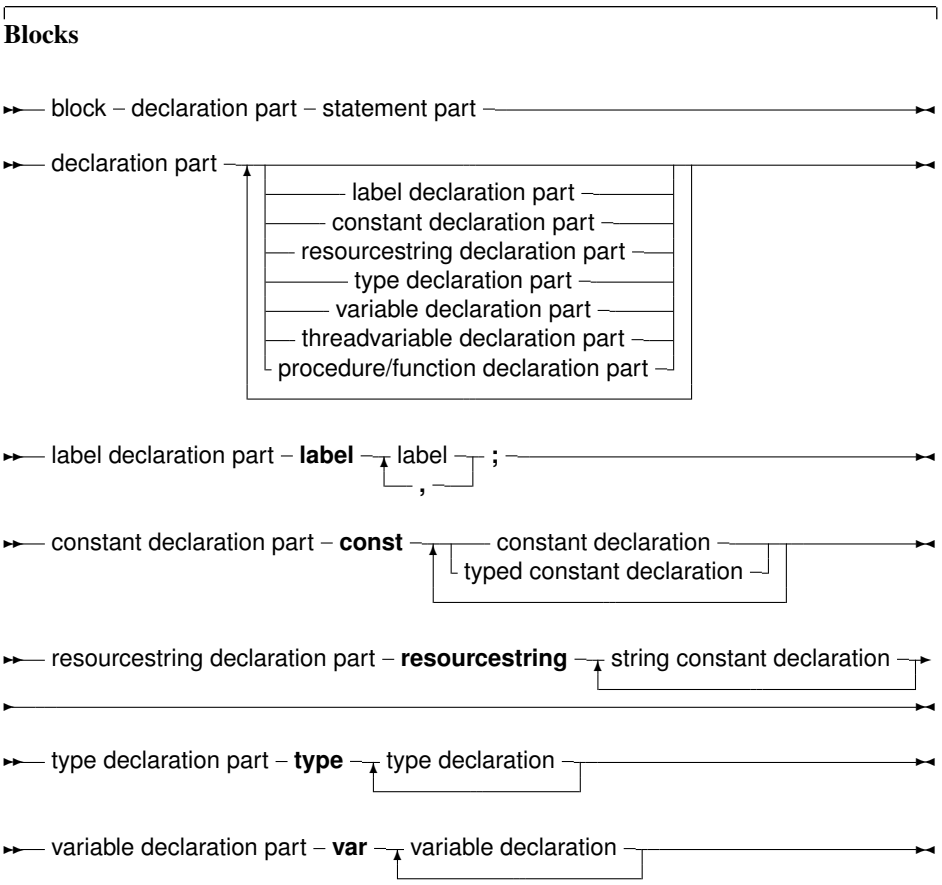
```
Unit UnitA;
interface
Uses UnitB;
implementation
end.
Unit UnitB
implementation
Uses UnitA;
end.
```

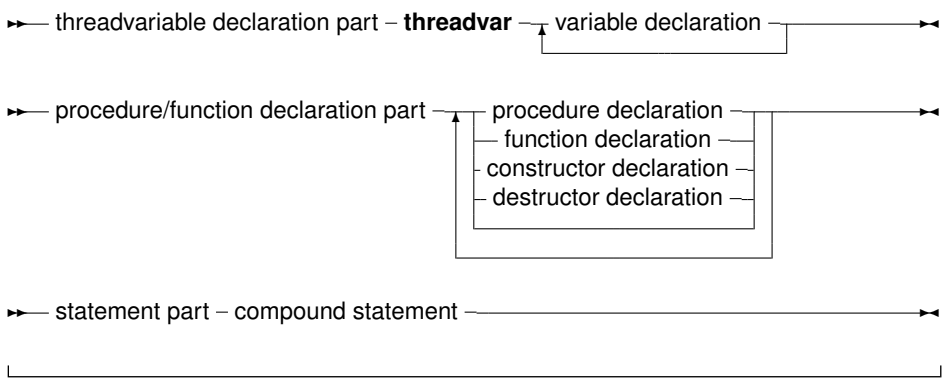
Because UnitB uses UnitA only in its implentation section.

In general, it is a bad idea to have unit interdependencies, even if it is only in implementation sections.

## 13.4 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 4 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.

## 13.5 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.

### 13.5.1 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 identfier 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 assigment}
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.

### 13.5.2 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.

### 13.5.3 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.

### 13.5.4 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;
```

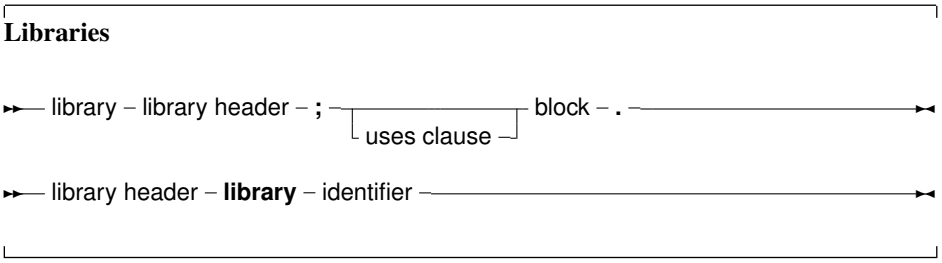
```
{ Redeclaration of MyType}
Type MyType = Integer;
Var A : Mytype;       { Will be Integer }
    B : UnitA.MyType { Will be real }
begin
end.
```

This is especially useful when redeclaring the system unit's identifiers.

## 13.6 Libraries

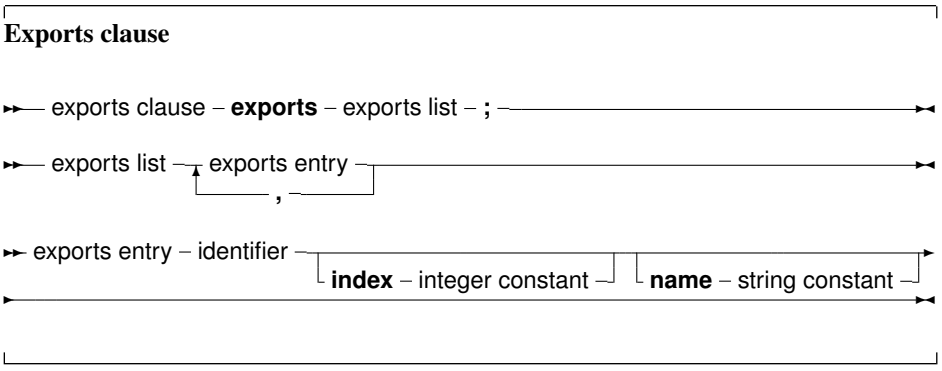
Free Pascal supports making of dynamic libraries (DLLs under Win32 and OS/2) trough the use of the `Library` keyword.

A Library is just like a unit or a program:




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:




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 14

# Exceptions

Exceptions provide a convenient way to program error and error-recovery mechanisms, and are closely related to classes. Exception support is based on 3 constructs:

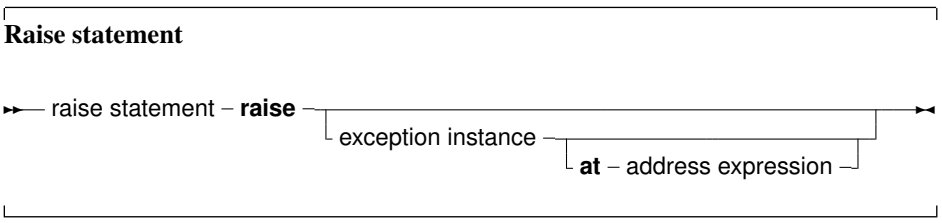
**Raise** statements. To raise an exeption. This is usually done to signal an error condition. It is however also usable to abort execution and immediatly return to a well-known point in the executable.

**Try ... Except** blocks. These block serve to catch exceptions raised within the scope of the block, and to provide exception-recovery code.

**Try ... Finally** blocks. These block serve to 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.

## 14.1 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 is optional. If it is 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 14.5, page 144). 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;
```

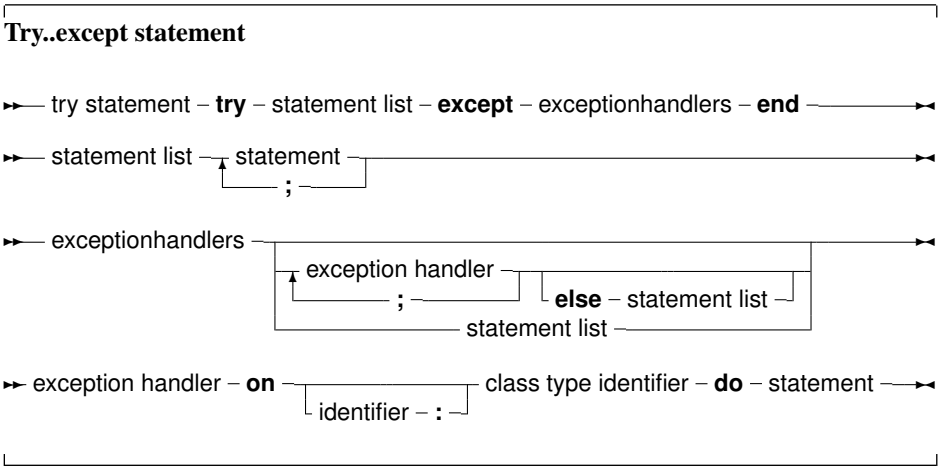
The class `Exception` is defined in the Sysutils unit of the rtl. (section 14.5, page 144)

**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`.

## 14.2 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 foillowing 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.

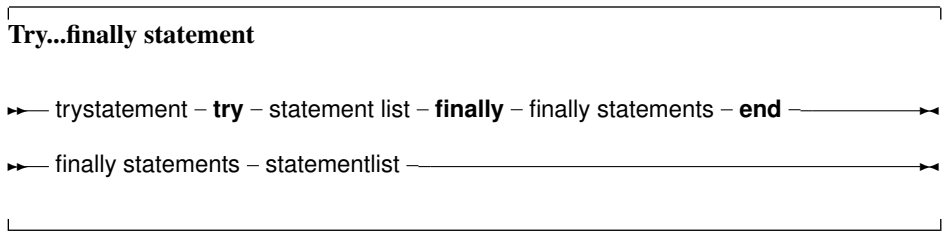
As an example, given the previous declaration of the `DoDiv` function, consider the following

```
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.

## 14.3 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.

If, however, an exception occurs, the program flow is immediatly transferred from the point where the excepion 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
  Try
    Assign (F,Name);
    Rewrite (name);
    ... File handling ...
  Finally
    Close(F);
  end;
```

If during the execution of the file handling an execption 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.

## 14.4 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.

## 14.5 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 a 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 15

# 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.

## 15.1 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 recognise them, and treat them as any other conditionals.

## 15.2 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.

# Index


Abstract, 57
Address, 90
Alias, 121
Ansistring, 23, 25
Array, 28, 114, 115
    Dynamic, 30
    Of const, 115
    Static, 29
array, 42
Asm, 107
Assembler, 107, 120, 145

block, 137
Boolean, 19

Case, 101
cdecl, 122
Char, 22
Class, 60, 65
Classes, 60
COM, 41, 77
Comments, 8
Comp, 22
Const, 16
    String, 16
Constants, 15
    Ordinary, 15
    String, 13, 15, 26
    Typed, 16
Constructor, 53, 63, 87
CORBA, 41, 77
Currency, 22

Destructor, 53
Directives
    Hint, 11
Dispatch, 66
DispatchStr, 67
Double, 22

else, 101, 102
except, 142, 144
Exception, 141
Exceptions, 141
    Catching, 141, 142
    Classes, 144
    Handling, 143, 144
    Raising, 141
export, 122
Expression, 104, 105
Expressions, 84
Extended, 22
External, 119
external, 45, 120

Fields, 32, 52
File, 37
finally, 143, 144
For, 103
Forward, 39, 118
Function, 110
Functions, 109
    Assembler, 120, 145
    External, 119
    Forward, 118
    Modifiers, 120
    Overloaded, 117

Generics, 78

Hint directives, 11

Identifiers, 10
If, 102
index, 70, 120
Inherited, 65
inline, 122
interface, 74
Interfaces, 41, 43, 74
    COM, 77
    CORBA, 77
    Implementations, 76
interrupt, 123

Labels, 13
Libraries, 140
library, 140
local, 123

Message, 66, 67
message, 66
Methods, 55, 64

Abstract, 57
Class, 65
Message, 66
Static, 55
Virtual, 56, 57, 64
Modifiers, 10, 120, 126
Alias, 121
cdecl, 122
export, 122
inline, 122
nostackframe, 123
overload, 123
pascal, 124
public, 124
register, 125
safecall, 125
saveregisters, 126
softfloat, 126
stdcall, 126
varargs, 126
Mofidiers
interrupt, 123
local, 123

name, 119
nostackframe, 123
Numbers, 12
Binary, 12
Decimal, 12
Hexadecimal, 12
Real, 12

object, 51
Objects, 51
Operators, 15, 27, 36, 39, 84, 90, 91
Arithmetic, 91, 130
Assignment, 128
Binary, 130
Boolean, 92
Comparison, 131
Logical, 92
Relational, 94
Set, 93
String, 93
Unary, 91
operators, 127
otherwise, 101
overload, 118, 123
overloading
operators, 127
Override, 65
override, 57

Packed, 33, 34, 52, 64
Parameters, 111
Constant, 111, 114
Open Array, 114, 115
Out, 113
Untypes, 111
Value, 111
Var, 70, 111, 112
pascal, 124
PChar, 25
Pointer, 37
Private, 58, 61, 70
private, 52
Procedural, 40
Procedure, 40, 109
Procedures, 109
program, 133
Properties, 47, 69
Array, 71
Indexed, 70
Property, 66, 69
Protected, 58, 61
Public, 58, 61
public, 52, 124
Published, 61, 70
PWideChar, 25

Raise, 141
Read, 69
Real, 22
Record, 32
Constant, 46
register, 125
reintroduce, 65
Repeat, 104
Reserved words, 9
Delphi, 10
Free Pascal, 10
Modifiers, 10
Turbo Pascal, 9
Resourcestring, 16

safecall, 125
saveregisters, 126
Scope, 24, 31, 46, 51, 58, 61, 138
block, 138
Class, 139
record, 139
unit, 139
Self, 55, 66, 68
Set, 36
Shortstring, 23
Single, 22
softfloat, 126
Statements, 97

Assembler, 107, 145
Assignment, 97
Case, 101
Compound, 100
Exception, 107
For, 103
Goto, 99
if, 102
Loop, 103–105
Procedure, 98
Repeat, 104
Simple, 97
Structured, 100
While, 105
With, 105
stdcall, 126
String, 13
Symbols, 8
Syntax diagrams, 7

Text, 37
then, 102
Thread Variables, 47
Threadvar, 47
Tokens, 8
Identifiers, 10
Numbers, 12
Reserved words, 9
Strings, 13
Symbols, 8
try, 143, 144
Type, 17
Typecast, 24, 25, 88, 89
Unaligned, 89
Value, 88
Variable, 89
Types, 17
Ansistring, 23
Array, 28–30
Base, 17
Boolean, 19
Char, 22
Class, 60
Enumeration, 20
File, 37
Forward declaration, 39
Integer, 18
Object, 51
Ordinal, 18
PChar, 25, 26
Pointer, 26, 37
Procedural, 40
Real, 22
Record, 32
Reference counted, 23, 25, 30, 31, 77
Set, 36
String, 22
Structured, 27
Subrange, 21
Variant, 41
Widestring, 25

unit, 134, 139
uses, 133

Var, 44
varargs, 126
Variable, 44
Variables, 44
Initialized, 16, 46
Variant, 41
Virtual, 53, 54, 56, 64, 66
Visibility, 51, 58, 74
Private, 51
Protected, 61
Public, 51, 61
Published, 61

While, 105
Widestring, 25
With, 105
Write, 70