

Free Pascal :  
Reference guide.

---

Reference guide for Free Pascal, version 2.4.2rc1  
Document version 2.4  
July 2012

Michaël Van Canneyt

---

# Contents

<b>1</b>	<b>Pascal Tokens</b>	<b>10</b>
1.1	Symbols . . . . .	10
1.2	Comments . . . . .	11
1.3	Reserved words . . . . .	12
1.3.1	Turbo Pascal reserved words . . . . .	12
1.3.2	Free Pascal reserved words . . . . .	13
1.3.3	Object Pascal reserved words . . . . .	13
1.3.4	Modifiers . . . . .	13
1.4	Identifiers . . . . .	13
1.5	Hint directives . . . . .	14
1.6	Numbers . . . . .	15
1.7	Labels . . . . .	16
1.8	Character strings . . . . .	17
<b>2</b>	<b>Constants</b>	<b>19</b>
2.1	Ordinary constants . . . . .	19
2.2	Typed constants . . . . .	20
2.3	Resource strings . . . . .	20
<b>3</b>	<b>Types</b>	<b>22</b>
3.1	Base types . . . . .	22
3.1.1	Ordinal types . . . . .	23
	Integers . . . . .	23
	Boolean types . . . . .	24
	Enumeration types . . . . .	25
	Subrange types . . . . .	26
3.1.2	Real types . . . . .	27
3.2	Character types . . . . .	27
3.2.1	Char . . . . .	27
3.2.2	Strings . . . . .	28
3.2.3	Short strings . . . . .	28

---

3.2.4	Ansistrings	29
3.2.5	UnicodeStrings	31
3.2.6	WideStrings	31
3.2.7	Constant strings	31
3.2.8	PChar - Null terminated strings	32
3.3	Structured Types	32
	Packed structured types	33
3.3.1	Arrays	34
	Static arrays	34
	Dynamic arrays	36
	Packing and unpacking an array	38
3.3.2	Record types	38
3.3.3	Set types	42
3.3.4	File types	43
3.4	Pointers	43
3.5	Forward type declarations	45
3.6	Procedural types	46
3.7	Variant types	47
3.7.1	Definition	47
3.7.2	Variants in assignments and expressions	48
3.7.3	Variants and interfaces	49
<b>4</b>	<b>Variables</b>	<b>51</b>
4.1	Definition	51
4.2	Declaration	51
4.3	Scope	53
4.4	Initialized variables	53
4.5	Thread Variables	54
4.6	Properties	54
<b>5</b>	<b>Objects</b>	<b>58</b>
5.1	Declaration	58
5.2	Fields	59
5.3	Static fields	60
5.4	Constructors and destructors	61
5.5	Methods	62
5.5.1	Declaration	62
5.5.2	Method invocation	63
	Static methods	63
	Virtual methods	64
	Abstract methods	65

---

5.6	Visibility	66
<b>6</b>	<b>Classes</b>	<b>67</b>
6.1	Class definitions	67
6.2	Class instantiation	71
6.3	Methods	71
6.3.1	Declaration	71
6.3.2	invocation	71
6.3.3	Virtual methods	72
6.3.4	Class methods	73
6.3.5	Message methods	73
6.3.6	Using inherited	75
6.4	Properties	76
6.4.1	Definition	76
6.4.2	Indexed properties	78
6.4.3	Array properties	79
6.4.4	Default properties	80
6.4.5	Storage information	80
6.4.6	Overriding properties	81
<b>7</b>	<b>Interfaces</b>	<b>83</b>
7.1	Definition	83
7.2	Interface identification: A GUID	84
7.3	Interface implementations	85
7.4	Interfaces and COM	86
7.5	CORBA and other Interfaces	86
7.6	Reference counting	86
<b>8</b>	<b>Generics</b>	<b>88</b>
8.1	Introduction	88
8.2	Generic class definition	88
8.3	Generic class specialization	90
8.4	A word about scope	91
<b>9</b>	<b>Expressions</b>	<b>94</b>
9.1	Expression syntax	95
9.2	Function calls	96
9.3	Set constructors	98
9.4	Value typecasts	98
9.5	Variable typecasts	99
9.6	Unaligned typecasts	100

9.7	The @ operator . . . . .	100
9.8	Operators . . . . .	101
9.8.1	Arithmetic operators . . . . .	101
9.8.2	Logical operators . . . . .	102
9.8.3	Boolean operators . . . . .	102
9.8.4	String operators . . . . .	103
9.8.5	Set operators . . . . .	103
9.8.6	Relational operators . . . . .	105
9.8.7	Class operators . . . . .	106
<b>10</b>	<b>Statements</b>	<b>108</b>
10.1	Simple statements . . . . .	108
10.1.1	Assignments . . . . .	108
10.1.2	Procedure statements . . . . .	109
10.1.3	Goto statements . . . . .	110
10.2	Structured statements . . . . .	111
10.2.1	Compound statements . . . . .	111
10.2.2	The Case statement . . . . .	112
10.2.3	The If..then..else statement . . . . .	113
10.2.4	The For..to/downto..do statement . . . . .	114
10.2.5	The For..in..do statement . . . . .	115
10.2.6	The Repeat..until statement . . . . .	122
10.2.7	The While..do statement . . . . .	123
10.2.8	The With statement . . . . .	124
10.2.9	Exception Statements . . . . .	125
10.3	Assembler statements . . . . .	125
<b>11</b>	<b>Using functions and procedures</b>	<b>127</b>
11.1	Procedure declaration . . . . .	127
11.2	Function declaration . . . . .	128
11.3	Function results . . . . .	128
11.4	Parameter lists . . . . .	129
11.4.1	Value parameters . . . . .	129
11.4.2	Variable parameters . . . . .	130
11.4.3	Out parameters . . . . .	131
11.4.4	Constant parameters . . . . .	132
11.4.5	Open array parameters . . . . .	133
11.4.6	Array of const . . . . .	133
11.5	Function overloading . . . . .	136
11.6	Forward defined functions . . . . .	136
11.7	External functions . . . . .	137

11.8 Assembler functions . . . . .	138
11.9 Modifiers . . . . .	138
11.9.1 alias . . . . .	139
11.9.2 cdecl . . . . .	140
11.9.3 export . . . . .	141
11.9.4 inline . . . . .	141
11.9.5 interrupt . . . . .	141
11.9.6 iocheck . . . . .	141
11.9.7 local . . . . .	142
11.9.8 nostackframe . . . . .	142
11.9.9 overload . . . . .	142
11.9.10 pascal . . . . .	143
11.9.11 public . . . . .	143
11.9.12 register . . . . .	144
11.9.13 safecall . . . . .	144
11.9.14 saveregisters . . . . .	144
11.9.15 softfloat . . . . .	145
11.9.16 stdcall . . . . .	145
11.9.17 varargs . . . . .	145
11.10 Unsupported Turbo Pascal modifiers . . . . .	145
<b>12 Operator overloading</b>	<b>146</b>
12.1 Introduction . . . . .	146
12.2 Operator declarations . . . . .	146
12.3 Assignment operators . . . . .	147
12.4 Arithmetic operators . . . . .	149
12.5 Comparision operator . . . . .	150
<b>13 Programs, units, blocks</b>	<b>152</b>
13.1 Programs . . . . .	152
13.2 Units . . . . .	153
13.3 Unit dependencies . . . . .	155
13.4 Blocks . . . . .	156
13.5 Scope . . . . .	157
13.5.1 Block scope . . . . .	157
13.5.2 Record scope . . . . .	158
13.5.3 Class scope . . . . .	158
13.5.4 Unit scope . . . . .	158
13.6 Libraries . . . . .	159
<b>14 Exceptions</b>	<b>161</b>

14.1 The raise statement . . . . .	161
14.2 The try...except statement . . . . .	162
14.3 The try...finally statement . . . . .	163
14.4 Exception handling nesting . . . . .	164
14.5 Exception classes . . . . .	164
<b>15 Using assembler</b>	<b>166</b>
15.1 Assembler statements . . . . .	166
15.2 Assembler procedures and functions . . . . .	166

# List of Tables

3.1	Predefined integer types . . . . .	23
3.2	Predefined integer types . . . . .	24
3.3	Boolean types . . . . .	24
3.4	Supported Real types . . . . .	27
3.5	PChar pointer arithmetic . . . . .	33
9.1	Precedence of operators . . . . .	94
9.2	Binary arithmetic operators . . . . .	101
9.3	Unary arithmetic operators . . . . .	102
9.4	Logical operators . . . . .	102
9.5	Boolean operators . . . . .	103
9.6	Set operators . . . . .	104
9.7	Relational operators . . . . .	105
9.8	Class operators . . . . .	106
10.1	Allowed C constructs in Free Pascal . . . . .	109
11.1	Unsupported modifiers . . . . .	145

## About this guide

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

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

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

► syntactical elements are like this —————►

Keywords which must be typed exactly as in the diagram:

► **keywords are like this** —————►

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

► this can be repeated —————►

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

► First possibility —————►  
Second possibility

Note, that one of the possibilities can be empty:

► First possibility —————►  
Second possibility

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

## About the Pascal language

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

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

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

# Chapter 1

## Pascal Tokens

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

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

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

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

**separators** This is usually white-space.

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

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

### 1.1 Symbols

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



The following characters have a special meaning:

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

and the following character pairs too:

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

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

## 1.2 Comments

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

The following piece of code demonstrates a comment:

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

The use of ( \* and \* ) as comment delimiters dates from the very first days of the Pascal language. It has been replaced mostly by the use of { and } as comment delimiters, as in the following example:

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

The comment can also span multiple lines:

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

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

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

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

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

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

The following are valid ways of nesting comments:

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

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

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

and

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

The compiler will react with a 'invalid character' error when it encounters such constructs, regardless of the `-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 as such, cannot be redefined by the programmer. Throughout the syntax diagrams they will be denoted using a **bold** typeface. Pascal is not case sensitive so the compiler will accept any combination of upper or lower case letters for reserved words.

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

### 1.3.1 Turbo Pascal reserved words

The following keywords exist in Turbo Pascal mode

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

### 1.3.2 Free Pascal reserved words

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

dispose	false	true
exit	new	

### 1.3.3 Object Pascal reserved words

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

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

### 1.3.4 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	reintroduce
assembler	forward	pascal	safecall
cdecl	index	private	softfloat
cppdecl	local	protected	stdcall
default	name	public	virtual
export	near	published	write

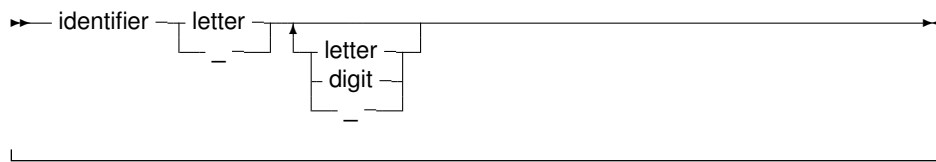
**Remark:** Predefined types such as `Byte`, `Boolean` and constants such as `maxint` are *not* reserved words. They are identifiers, declared in the system unit. This means that these types can be redefined in other units. The programmer is however not encouraged to do this, as it will cause a lot of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

## 1.4 Identifiers

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

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

**Identifiers**



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

```
myprocedure;
```

and

```
MyProcedure;
```

refer to the same procedure.

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

```
program testdo;
```

```
procedure &do;
```

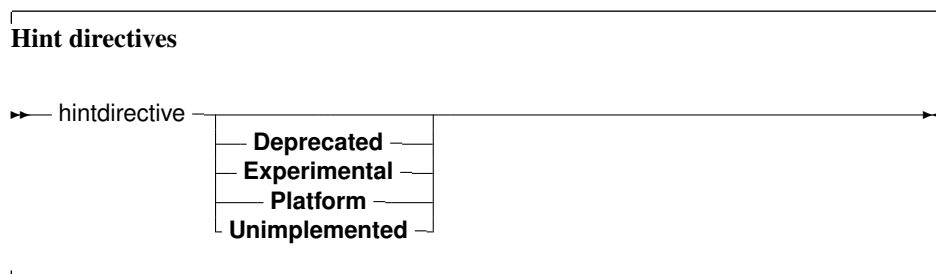
```
begin
end;
```

```
begin
  &do;
end.
```

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

## 1.5 Hint directives

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



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

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

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

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

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

The following are examples:

```
Const
  AConst = 12 deprecated;

var
  p : integer platform;

Function Something : Integer; experimental;

begin
  Something:=P+AConst;
end;

begin
  Something;
end.
```

This would result in the following output:

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

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

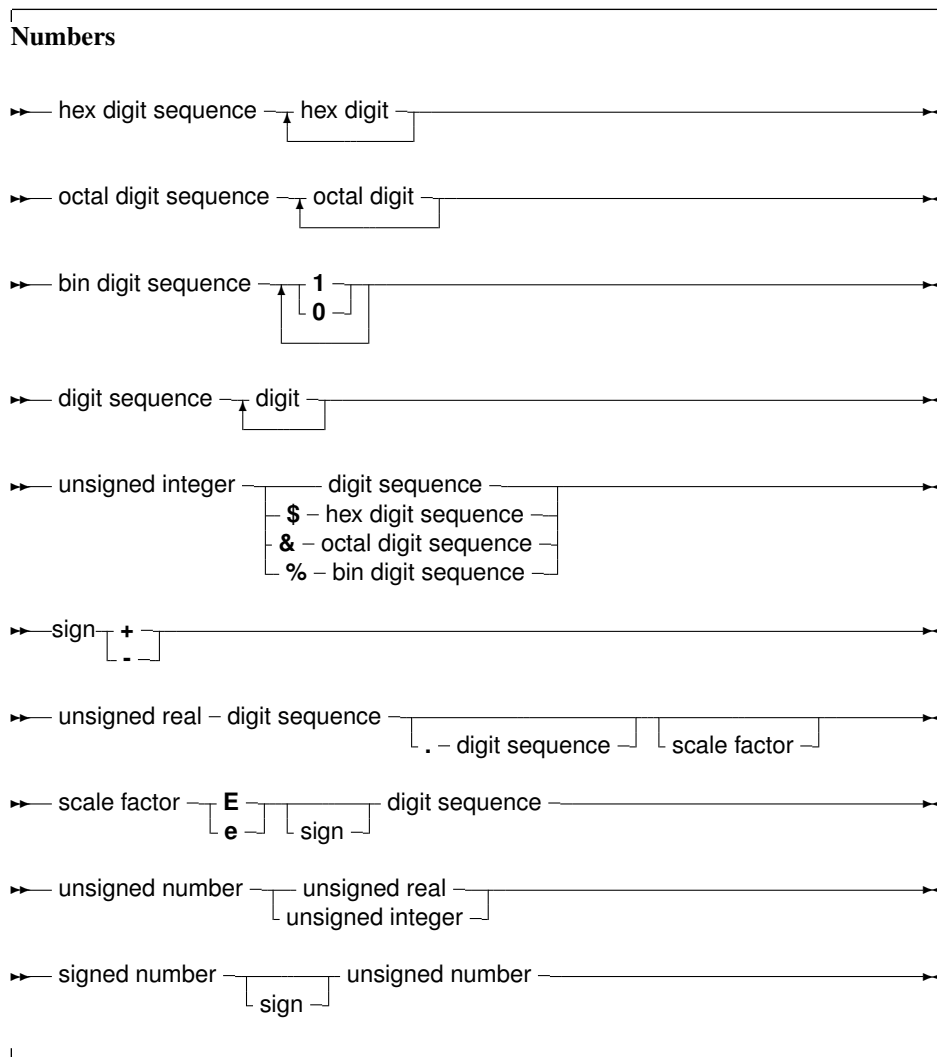
## 1.6 Numbers

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

For integer type constants, Free Pascal supports 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 an ampersand (&). For instance 15 is specified in octal notation as &17.
4. Binary notation (base 2). A binary number can be specified by preceding it with a percent sign (%). Thus, 255 can be specified in binary notation as %11111111.

The following diagrams show the syntax for numbers.



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

## 1.7 Labels

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

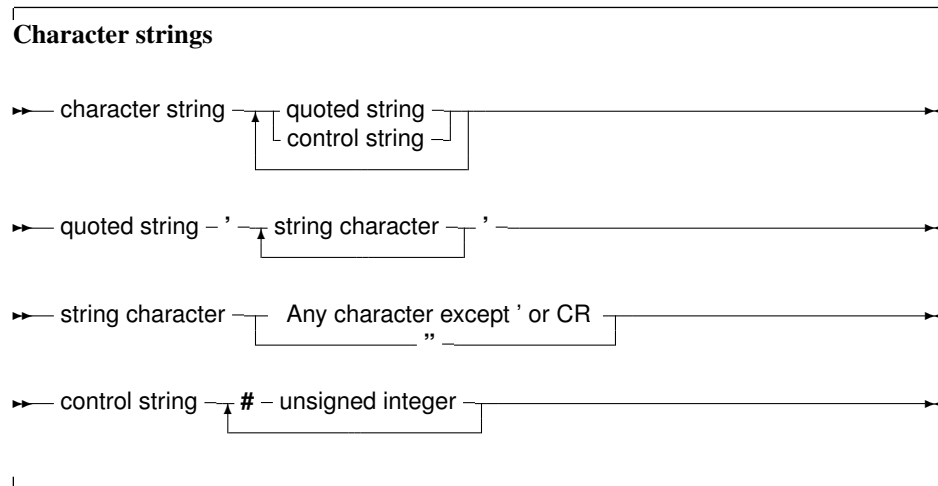


**Remark:** The `-Sg` or `-Mtp` switches must be specified before labels can be used. By default, Free Pascal doesn't support `label` and `goto` statements. The `{ $GOTO ON }` directive can also be used to allow use of labels and the `goto` statement.

## 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 a single line of the program source code: no literal carriage return or linefeed characters can appear in the string.

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



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

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

The following are valid string constants:

```

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

```

The following is an invalid string:

```

'the string starts here
and continues here'

```

The above string must be typed as:

```

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

```

or

```

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

```

on unices (including Mac OS X), and as

```

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

```

on a classic Mac-like operating system.

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

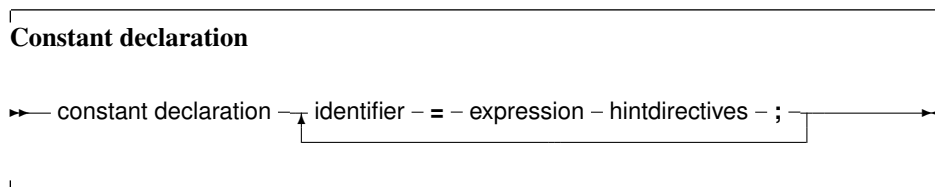
## Chapter 2

# Constants

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

### 2.1 Ordinary constants

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



The compiler must be able to evaluate the expression in a constant declaration at compile time. This means that most of the functions in the Run-Time library cannot be used in a constant declaration. Operators such as `+`, `-`, `*`, `/`, `not`, `and`, `or`, `div`, `mod`, `ord`, `chr`, `sizeof`, `pi`, `int`, `trunc`, `round`, `frac`, `odd` can be used, however. For more information on expressions, see chapter 9, page 94.

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:

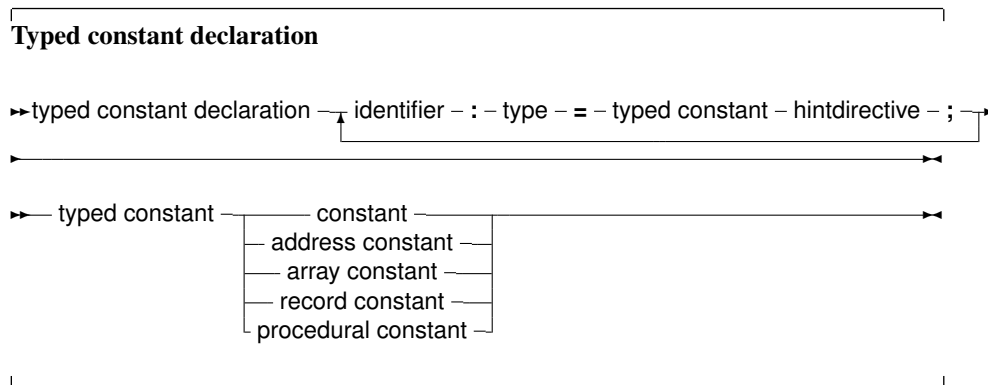
```
s := 'some other string';
```

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

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

## 2.2 Typed constants

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



Contrary to ordinary constants, a value can be assigned to them at run-time. This is an old concept from Turbo Pascal, which has been replaced with support for initialized variables: For a detailed description, see section 4.4, page 53.

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

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

## 2.3 Resource strings

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

The following is an example of a `resourcestring` definition:

```
Resourcestring
```

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

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

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

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

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

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

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

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

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

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

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

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

## Chapter 3

# Types

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

### Type declaration

→ type declaration – identifier – = – type – ; →

There are 7 major type classes :

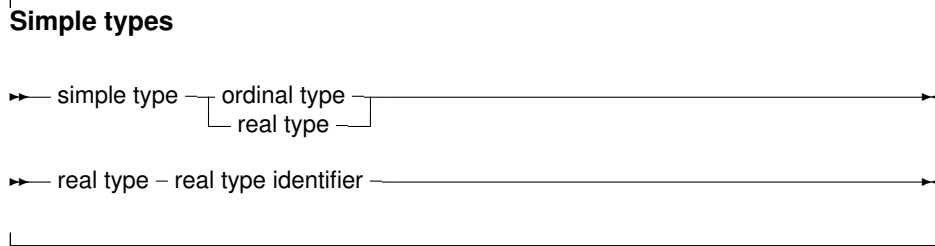
### Types



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. Any programmer who then uses these custom types doesn't have to worry about the underlying type size: it is opaque to him. It also allows to use shortcut names for fully qualified type names. e.g. define `system.longint` as `Olongint` and then redefine `longint`.

### 3.1 Base types

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



### 3.1.1 Ordinal types

With the exception of `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 by one, in a specified order. This property allows the operation of functions as `Inc`, `Ord`, `Dec` on ordinal types to be defined.
2. Ordinal values have a smallest possible value. Trying to apply the `Pred` function on the smallest possible value will generate a range check error if range checking is enabled.
3. Ordinal values have a largest possible value. Trying to apply the `Succ` function on the largest possible value will generate a range check error if range checking is enabled.

### 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
WordBool
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.

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

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.

**Remark:** All decimal constants which do not 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.

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 the `ByteBool`, `WordBool` and `LongBool` types. These are of

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

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:

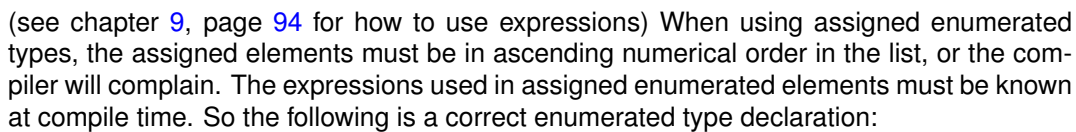
```
B := True;
B := False;
B := 1<>2; { Results in B := True }
```

Boolean expressions are also used in conditions.

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

Here `Func` is a function which returns a `Boolean` type.  
This behaviour is controllable by the `{ $B }` compiler directive.

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.



A C-style enumeration type looks as follows:

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.

25

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

→ subrange type – constant – .. – constant ←

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,

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	19-20	8

in effect, a 64-bit integer and is not available on all target platforms. To get more information on the supported types for each platform, refer to the [Programmer's Guide](#).

The currency type is a fixed-point real data type which is internally used as an 64-bit integer type (automatically scaled with a factor 10000), this 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'.

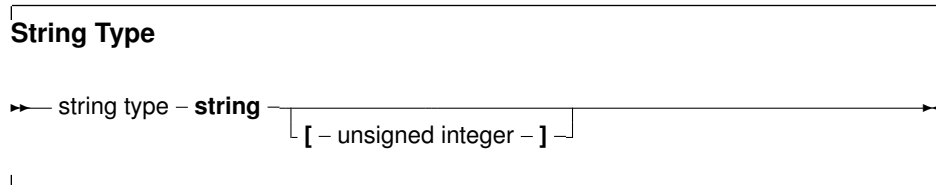
Also, the caret character (^) can be used in combination with a letter to specify a character with ASCII value less than 27. Thus ^G equals #7 - G is the seventh letter in the alphabet. The compiler is rather sloppy about the characters it allows after the caret, but in general one should assume only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

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

### 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 transparently, 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 guaranteed 1
```

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

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 will 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 UnicodeStrings

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

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

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

### 3.2.6 WideStrings

Widestrings (used to represent unicode character strings in COM applications) are implemented in much the same way as `unicodestrings`. Unlike the latter, they are *not* reference counted, and on Windows, they are allocated with a special windows function which allows them to be used for OLE automation. This means they are implemented as null-terminated arrays of `WideChars` instead of regular `Chars`. 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`. Similar to `unicodestrings`, the compiler transparently converts `WideStrings` to `AnsiStrings` and vice versa.

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

### 3.2.7 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 subtracted, however.

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

### 3.2.8 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`.

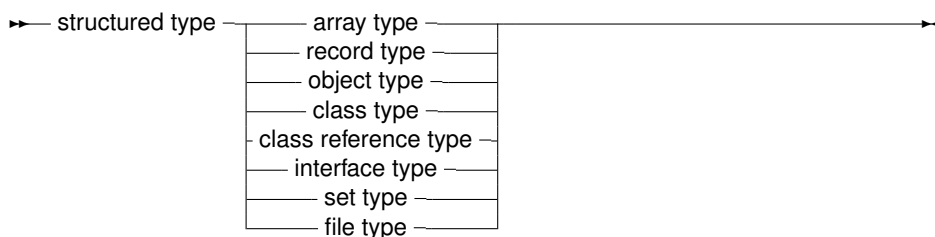
## 3.3 Structured Types

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

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 $P$ .
$P - I$	Subtracts $I$ 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 $Q$ )

## Structured Types



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

### Packed structured types

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

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

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

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

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

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

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

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

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

There are some more restrictions to elements of bitpacked structures:

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

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

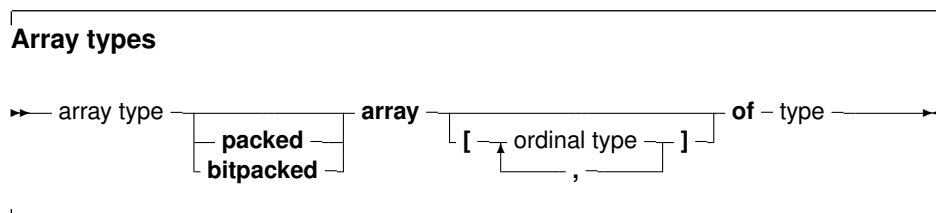
The size of bitpacked records and arrays is limited:

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

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

### 3.3.1 Arrays

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



#### Static arrays

When the range of the array is included in the array definition, it is called a static array. Trying to access an element with an index that is outside the declared range will generate

a run-time error (if range checking is on). The following is an example of a valid array declaration:

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

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

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

is equivalent to the declaration:

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

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

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

```
program testarray1;

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

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

```
    end;  
end.
```

The output of this program will be 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  
  TArray = 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 : TArray;  
  
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 `ansistring`s, 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 : TArray;  
  
begin  
  SetLength(A, 10);  
  A[0] := 33;  
  B := A;  
  A[0] := 31;
```

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

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

```
program testarray1;

Type
  TA = Array of array of Integer;

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

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

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

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

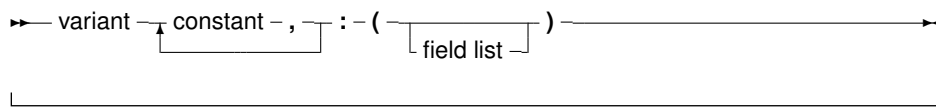
```
program testarray3;

Type
  TA = array of Integer;

var
  A,B : TA;
  I : Integer;

begin
  Setlength(A,10);
  For I:=0 to 9 do
    A[I]:=I;
```





So the following are valid record type declarations:

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

The variant part must be last in the record. The optional identifier in the case statement serves to access the tag field value, which otherwise would be invisible to the programmer. It can be used to see which variant is active at a certain time<sup>1</sup>. 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.

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.

<sup>1</sup>However, it is up to the programmer to maintain this field.

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 }
```

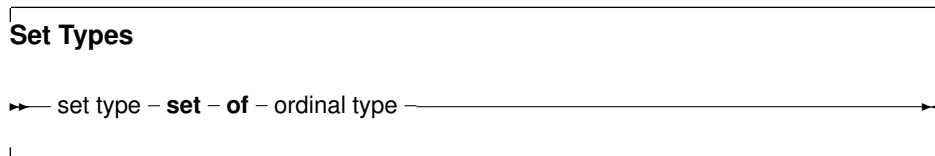
and

```
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];
```

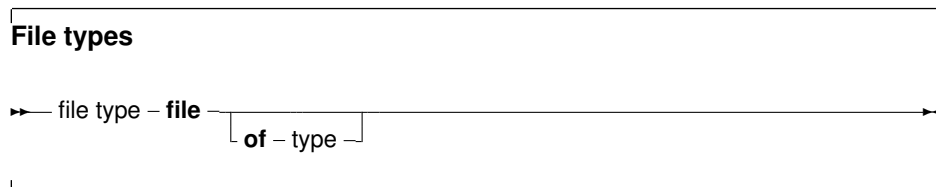
The compiler stores small sets (less than 32 elements) in a `Longint`, if the type range allows it. This allows for faster processing and decreases program size. Otherwise, sets are stored in 32 bytes.

Several operations can be done on sets: taking unions or differences, adding or removing elements, comparisons. These are documented in section 9.8.5, page 103

### 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:

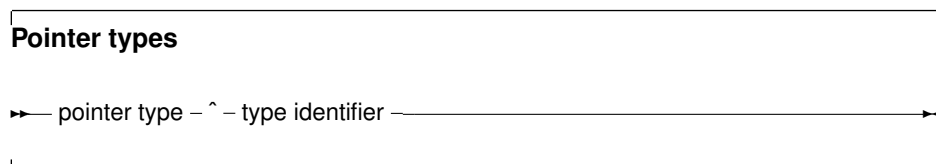
```
Type
  Point = Record
    X,Y,Z : real;
  end;
  PointFile = File of Point;
```

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. If the pointer is untyped, a size of 1 byte is assumed (i.e. as if the pointer were a pointer to a byte: `^byte`.)

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

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

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

### 3.5 Forward type declarations

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

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

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

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

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

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

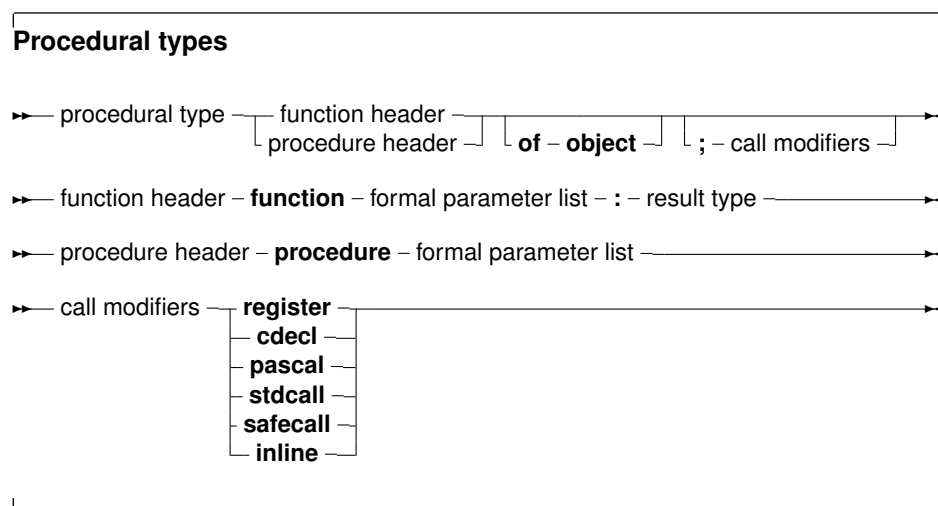
The referenced type should be introduced later in the same `Type` block. No other block may come between the definition of the pointer type and the referenced type. Indeed, even the word `Type` itself may not re-appear: in effect it would start a new type-block, causing the compiler to resolve all pending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

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

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

### 3.6 Procedural types

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



For a description of formal parameter lists, see chapter 11, page 127. 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 variant. Almost any simple type can be assigned to variants: ordinal types, string types, `int64` types.

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

This means that the following assignments are valid:

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

Var
```

```
V : Variant;  
I : Integer;  
B : Byte;  
W : Word;  
Q : Int64;  
E : Extended;  
D : Double;  
En : TMyEnum;  
AS : AnsiString;  
WS : WideString;  
  
begin  
  V:=I;  
  V:=B;  
  V:=W;  
  V:=Q;  
  V:=E;  
  V:=En;  
  V:=D;  
  V:=AS;  
  V:=WS;  
end;
```

And of course vice-versa as well.

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

```
Program testv;  
  
uses variants;  
  
Var  
  A : Variant;  
  I : integer;  
  
begin  
  A:=VarArrayCreate([1,10],varInteger);  
  For I:=1 to 10 do  
    A[I]:=I;  
  end.
```

For the explanation of `VarArrayCreate`, see [Unit Reference](#).

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

### 3.7.2 Variants in assignments and expressions

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

```
program testv3;

uses Variants;

Var
  V : Variant;
  I : Integer;

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

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

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

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

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

### 3.7.3 Variants and interfaces

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

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

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

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

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

The line

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

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

# Variables

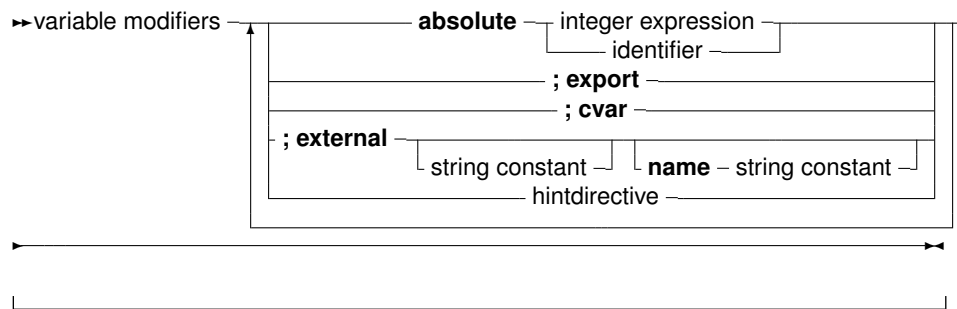
Variables are explicitly named memory locations with a certain type. When assigning values to variables, the Free Pascal compiler generates machine code to move the value to the memory location reserved for this variable. Where this variable is stored depends on where it is declared:

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

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.

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 eleventh 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 initialized elements must be specified, surrounded by round brackets, and separated by commas. The number of initialized elements must be exactly the same as the number of elements in the declaration of the type. As an example:

```

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

```

For constant records, each element of the record 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](#).

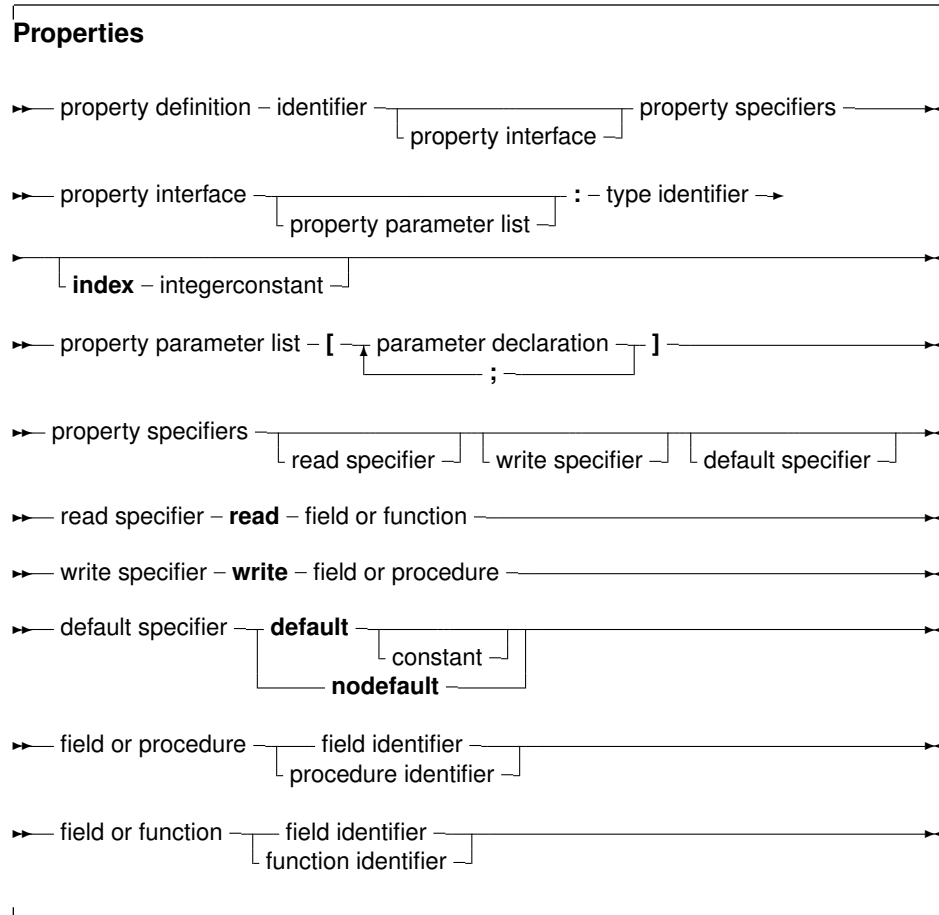
## 4.6 Properties

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 [67](#).

## 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.

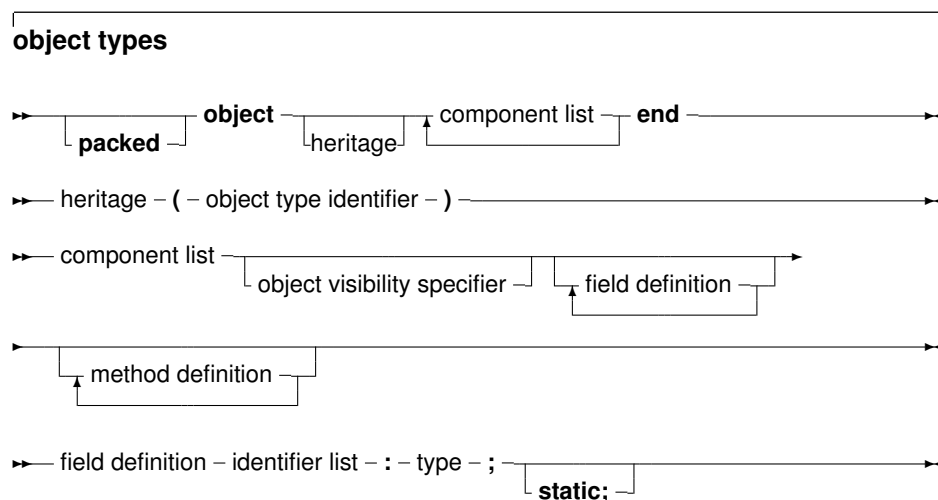
Objects should be treated as a special kind of record. The record contains all the fields that are declared in the objects definition, and pointers to the methods that are associated to the objects' type.

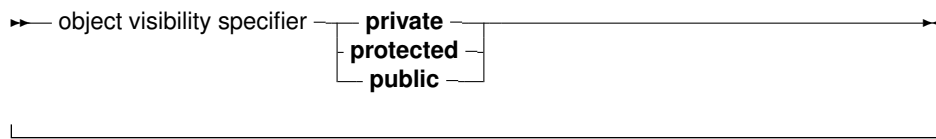
An object is declared just as a record would be declared; except that now, procedures and functions can be declared as if they were part of the record. Objects can "inherit" fields and methods from "parent" objects. This means that these fields and methods can be used as if they were included in the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

The following is a valid definition of an object:

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

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

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

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

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

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

## 5.2 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 10.2.8](#), page 124.

## 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(c1.l);
end.

```

will be the following

```

2
3
3

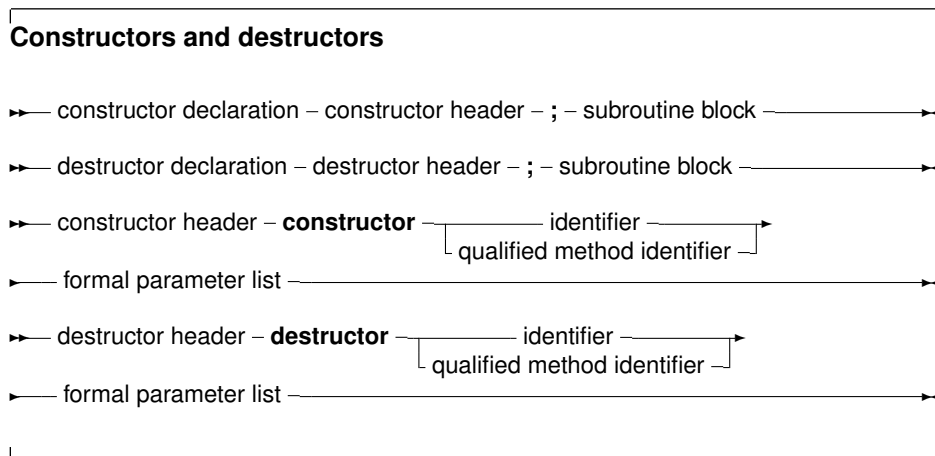
```

Note that the last line of code references the object type itself (`c1`), and not an instance of the object (`c11` or `c12`).

## 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<sup>1</sup>.

In the declaration of the object type, a simple identifier should be used for the name of the constructor or destructor. When the constructor or destructor is implemented, 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:

<sup>1</sup>A pointer to the VMT must be set up.

```

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

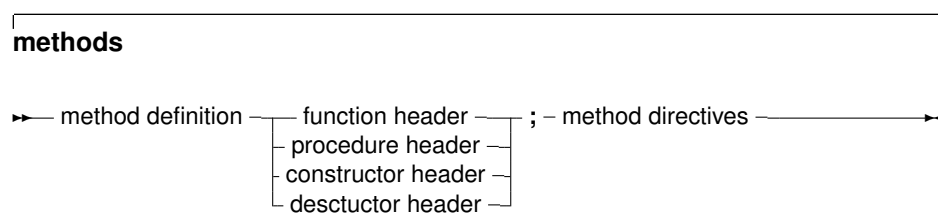
In view of the compiler warning remark, the following chapter presents the Delphi approach to object-oriented programming, and may be considered a more natural way of object-oriented programming.

## 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:





```

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

```

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

### Virtual methods

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

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

```

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

```

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

```

Var
  ParentA, ParentB : PParent;
  Child           : PChild;

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

```

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

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

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

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

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

To be able to do this, the compiler keeps - per object type - a table with virtual methods: the VMT (Virtual Method Table). This is simply a table with pointers to each of the virtual methods: each virtual method has its fixed location in this table (an index). The compiler uses this table to look up the actual method that must be used. When a descendent object overrides a method, the entry of the parent method is overwritten in the VMT. More information about the VMT can be found in [Programmer's Guide](#).

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

### Abstract methods

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

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

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

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

```
    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.

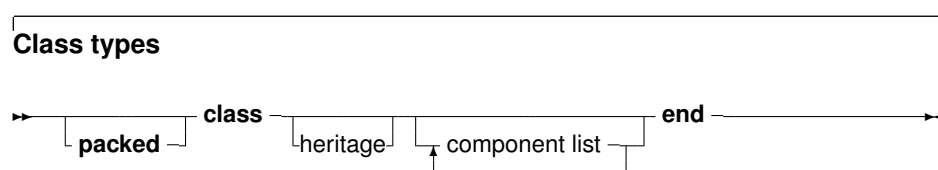
**Remark:** In earlier versions of Free Pascal it was necessary, in order to use classes, to put the `objpas` unit in the `uses` clause of a unit or program. *This is no longer needed* as of version 0.99.12. As of this version, the unit will be loaded automatically when the `-MObjfpc` or `-MDelphi` options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

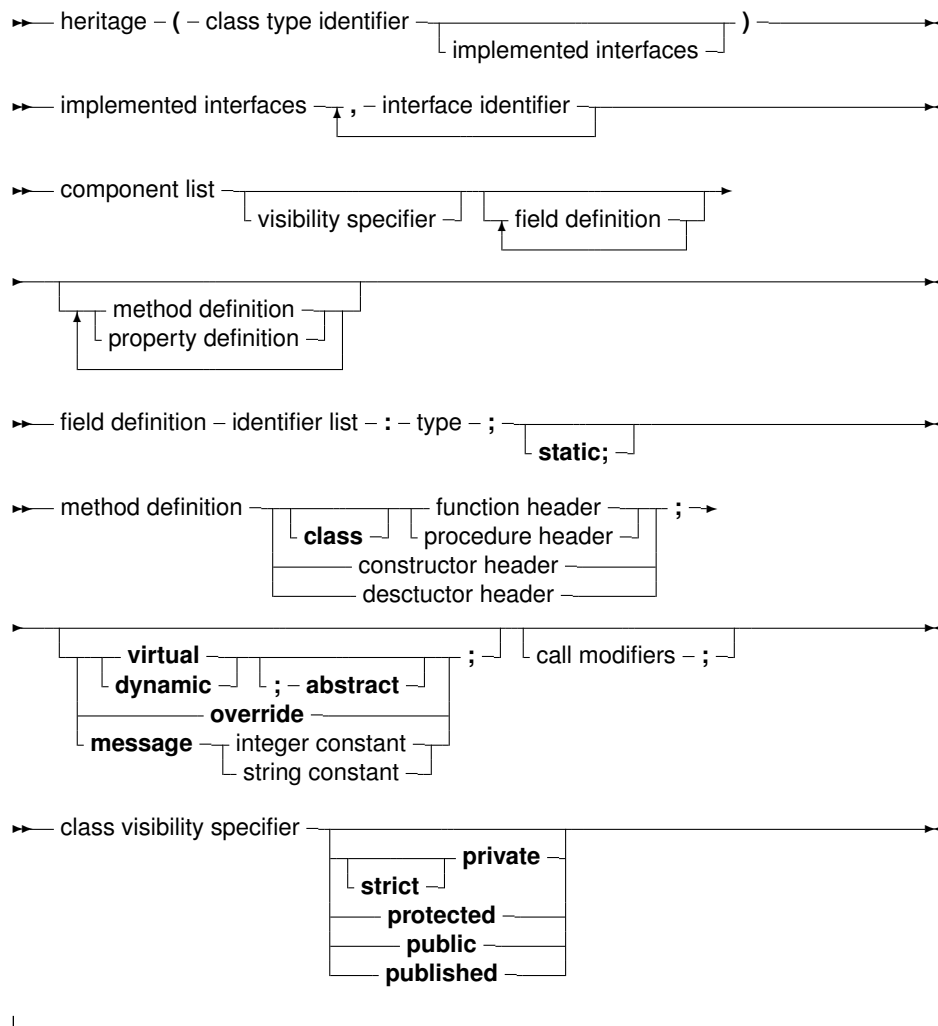
```
{ $mode objfpc }
{ $mode delphi }
```

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:





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

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

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

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

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

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

**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 if the compiler is in the `{$M+}` state. Fields defined in a `published` section must be of class type. Array properties cannot be in a `published` section.

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

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

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(c1.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 of** – classtype →

Class reference types are used to create instances of a certain class, which is not yet known at compile time, but which is specified at run time. Essentially, a variable of a class

reference type contains a pointer to the definition of the specified class. This can be used to construct an instance of the class corresponding to the definition, or to check inheritance. The following example shows how it works:

```
Type
  TComponentClass = Class of TComponent;

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

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

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

```
Var
  C : TComponent;

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

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

```
Var
  C : TComponent;

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

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

References to classes can also be used to check inheritance:

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

Function CheckObjectBetween (Instance : TObject) : boolean;

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

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

More about instantiating a class can be found in 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 constructor's code is executed. The constructor has a pointer to its data, in `Self`.

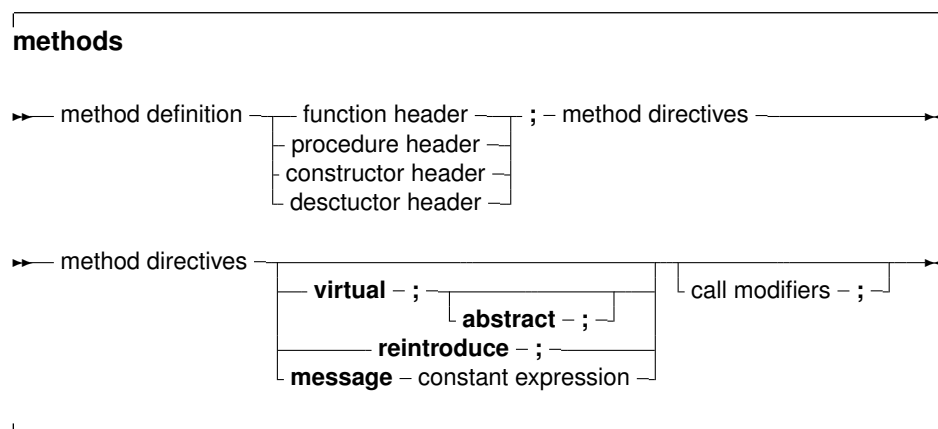
### Remark:

- The `{ $PackRecords }` directive also affects classes. i.e. the alignment in memory of the different fields depends on the value of the `{ $PackRecords }` directive.
- Just as for objects and records, a packed class can be declared. This has the same effect as on an object, or record, namely that the elements are aligned on 1-byte boundaries. i.e. as close as possible.
- `SizeOf(class)` will return the same as `SizeOf(Pointer)`, since a class is 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](#).

**Remark:** The keyword 'virtual' can be replaced with the 'dynamic' keyword: dynamic methods behave the same as virtual methods. Unlike in Delphi, in FPC the implementation of dynamic methods is equal to the implementation of virtual methods.

### 6.3.4 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 constant that they were declared with. They are

primarily intended to ease programming of callback functions in several GUI toolkits, such as Win32 or GTK. In difference with Delphi, Free Pascal also accepts strings as message identifiers. Message methods are always virtual.

As can be seen in the class declaration diagram, message methods are declared with a `Message` keyword, followed by an integer constant expression.

Additionally, they can take only one var argument (typed or not):

```
Procedure TMyObject.MyHandler(Var Msg); Message 1;
```

The method implementation of a message function is not different from an ordinary method. It is also possible to call a message method directly, but this should not be done. Instead, the `TObject.Dispatch` method should be used. Message methods are automatically virtual, i.e. they can be overridden in descendent classes.

The `TObject.Dispatch` method can be used to call a message handler. It is declared in the system unit and will accept a var parameter which must have at the first position a cardinal with the message ID that should be called. For example:

```
Type
  TMsg = Record
    MSGID : Cardinal
    Data : Pointer;
Var
  Msg : TMsg;

MyObject.Dispatch (Msg);
```

In this example, the `Dispatch` method will look at the object and all its ancestors (starting at the object, and searching up the inheritance class tree), to see if a message method with message `MSGID` has been declared. If such a method is found, it is called, and passed the `Msg` parameter.

If no such method is found, `DefaultHandler` is called. `DefaultHandler` is a virtual method of `TObject` that doesn't do anything, but which can be overridden to provide any processing that might be needed. `DefaultHandler` is declared as follows:

```
procedure defaulthandler(var message);virtual;
```

In addition to the message method with a `Integer` identifier, Free Pascal also supports a message method with a string identifier:

```
Procedure TMyObject.MyStrHandler(Var Msg); Message 'OnClick';
```

The working of the string message handler is the same as the ordinary integer message handler:

The `TObject.DispatchStr` method can be used to call a message handler. It is declared in the system unit and will accept one parameter which must have at the first position a short string with the message ID that should be called. For example:

```
Type
  TMsg = Record
    MsgStr : String[10]; // Arbitrary length up to 255 characters.
    Data : Pointer;
Var
```

```
Msg : TMsg;

MyObject.DispatchStr (Msg);
```

In this example, the `DispatchStr` method will look at the object and all its ancestors (starting at the object, and searching up the inheritance class tree), to see if a message method with message `MsgStr` has been declared. If such a method is found, it is called, and passed the `Msg` parameter.

If no such method is found, `DefaultHandlerStr` is called. `DefaultHandlerStr` is a virtual method of `TObject` that doesn't do anything, but which can be overridden to provide any processing that might be needed. `DefaultHandlerStr` is declared as follows:

```
procedure DefaultHandlerStr(var message);virtual;
```

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;

```

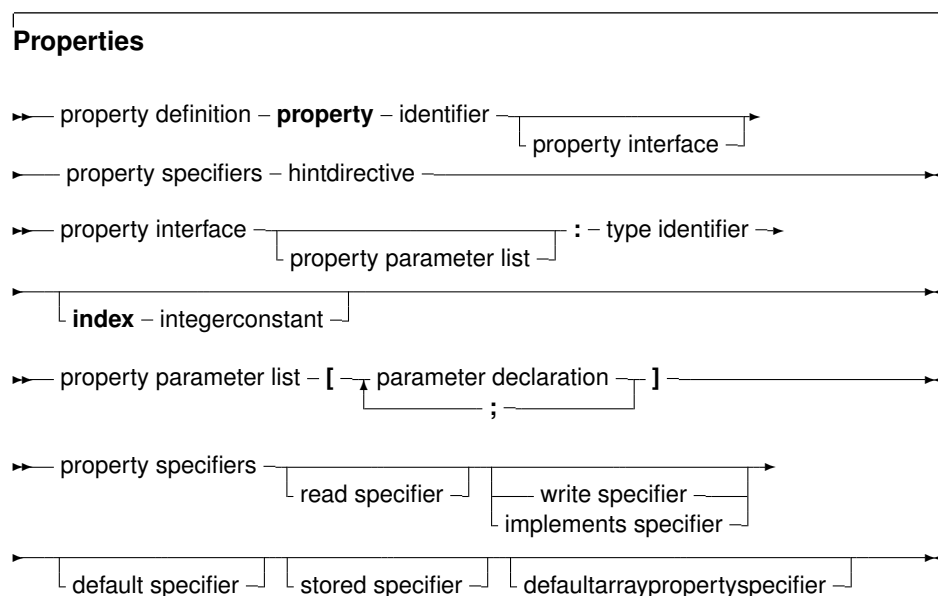
The `CreateNew` method will first call `TComponent.Create` and will pass it `AOwner` as a parameter. It will not call `TMyClass.Create`.

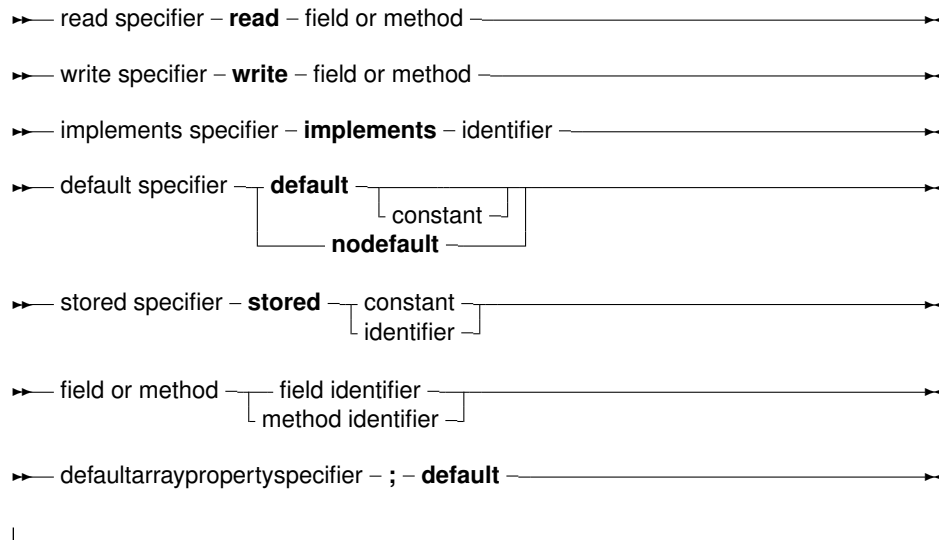
Although the examples were given using constructors, the use of `inherited` is not restricted to constructors, it can be used for any procedure or function or destructor as well.

## 6.4 Properties

### 6.4.1 Definition

Classes can contain properties as part of their fields list. A property acts like a normal field, i.e. its value can be retrieved or set, but it allows to redirect the access of the field through functions and procedures. They provide a means to associate an action with an assignment of or a reading from a class 'field'. This allows 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 parameter to the various `setNNN` methods.

Because of this mechanism, properties cannot be passed as var arguments to a function or procedure, since there is no known address of the property (at least, not always).

### 6.4.2 Indexed properties

If the property definition contains an index, then the read and write specifiers must be a function and a procedure. Moreover, these functions require an additional parameter : An integer parameter. This allows to read or write several properties with the same function. For this, the properties must have the same type. The following is an example of a property with an index:

```
{ $mode objfpc }
Type
  TPoint = Class(TObject)
  Private
    FX, FY : Longint;
    Function GetCoord (Index : Integer) : Longint;
    Procedure SetCoord (Index : Integer; Value : longint);
  Public
    Property X : Longint index 1 read GetCoord Write SetCoord;
    Property Y : Longint index 2 read GetCoord Write SetCoord;
    Property Coords[Index : Integer] : Longint Read GetCoord;
  end;

Procedure TPoint.SetCoord (Index : Integer; Value : Longint);
begin
  Case Index of
    1 : FX := Value;
    2 : FY := Value;
  end;
end;

Function TPoint.GetCoord (INdex : Integer) : Longint;
begin
  Case Index of
```

```

    1 : Result := FX;
    2 : Result := FY;
end;
end;

Var
    P : TPoint;

begin
    P := TPoint.create;
    P.X := 2;
    P.Y := 3;
    With P do
        WriteLn ('X=', X, ' Y=', Y);
    end.

```

When the compiler encounters an assignment to `X`, then `SetCoord` is called with as first parameter the index (1 in the above case) and with as a second parameter the value to be set. Conversely, when reading the value of `X`, the compiler calls `GetCoord` and passes it index 1. Indexes can only be integer values.

### 6.4.3 Array properties

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 argument a variable of the same type as the index type. For an array property, one cannot specify fields as `read specifiers`.

A `write specifier` for an array property is the name of a method procedure that accepts two arguments: The first argument has the same type as the index, and the second argument is a parameter of the same type as the property type. As an example, see the following declaration:

```

Type
    TIntList = Class
    Private
        Function GetInt (I : Longint) : longint;
        Function GetAsString (A : String) : String;
        Procedure SetInt (I : Longint; Value : Longint);
        Procedure SetAsString (A : String; Value : String);
    Public
        Property Items [i : Longint] : Longint Read GetInt
                                                    Write SetInt;
        Property StrItems [S : String] : String Read GetAsString
                                                    Write SetAsString;
    end;

Var
    AIntList : TIntList;

```

Then the following statements would be valid:

```

AIntList.Items[26] := 1;
AIntList.StrItems['twenty-five'] := 'zero';
WriteLn ('Item 26 : ', AIntList.Items[26]);
WriteLn ('Item 25 : ', AIntList.StrItems['twenty-five']);

```

While the following statements would generate errors:

```

AIntList.Items['twenty-five'] := 1;
AIntList.StrItems[26] := 'zero';

```

Because the index types are wrong.

#### 6.4.4 Default properties

Array properties can be declared as `default` properties. This means that it is not necessary to specify the property name when assigning or reading it. In the previous example, if the definition of the `items` property would have been

```

Property Items[i : Longint] : Longint Read GetInt
                                     Write SetInt; Default;

```

Then the assignment

```
AIntList.Items[26] := 1;
```

Would be equivalent to the following abbreviation.

```
AIntList[26] := 1;
```

Only one default property per class is allowed, and descendent classes cannot redeclare the default property.

#### 6.4.5 Storage information

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 responsibility of the programmer to do this in the constructor of the class.
2. The value 2147483648 cannot be used as a default value, as it is used internally to denote `nodefault`.
3. It is not possible to specify a default for array properties.

The *nodefault specifier* (`nodefault`) must be used to indicate that a property has no default value. The effect is that the value of this property is always written to the stream when streaming the property.

### 6.4.6 Overriding properties

Properties can be overridden in descendent classes, just like methods. The difference is that for properties, the overriding can always be done: properties should not be marked 'virtual' so they can be overridden, they are always overridable (in this sense, properties are always 'virtual'). The type of the overridden property does not have to be the same as the parents class property type.

Since they can be overridden, the keyword 'inherited' can also be used to refer to the parent definition of the property. For example consider the following code:

```
type
  TAncestor = class
    private
      FP1 : Integer;
    public
      property P: integer Read FP1 write FP1;
    end;

  TClassA = class(TAncestor)
    private
      procedure SetP(const AValue: char);
      function getP : Char;
    public
      constructor Create;
      property P: char Read GetP write SetP;
    end;

procedure TClassA.SetP(const AValue: char);

begin
  Inherited P:=Ord(AValue);
end;

procedure TClassA.GetP : char;

begin
  Result:=Char((Inherited P) and $FF);
end;
```

TClassA redefines P as a character property instead of an integer property, but uses the parents P property to store the value.

Care must be taken when using virtual get/set routines for a property: setting the inherited propert still observes the normal rules of inheritance for methods. Consider the following example:

```
type
  TAncestor = class
    private
```

```
    procedure SetP1(const AValue: integer); virtual;
public
    property P: integer write SetP1;
end;

TClassA = class(TAncestor)
private
    procedure SetP1(const AValue: integer); override;
    procedure SetP2(const AValue: char);
public
    constructor Create;
    property P: char write SetP2;
end;

constructor TClassA.Create;
begin
    inherited P:=3;
end;
```

In this case, when setting the inherited property `P`, the implementation `TClassA.SetP1` will be called, because the `SetP1` method is overridden.

If the parent class implementation of `SetP1` must be called, then this must be called explicitly:

```
constructor TClassA.Create;
begin
    inherited SetP1(3);
end;
```

## Chapter 7

# Interfaces

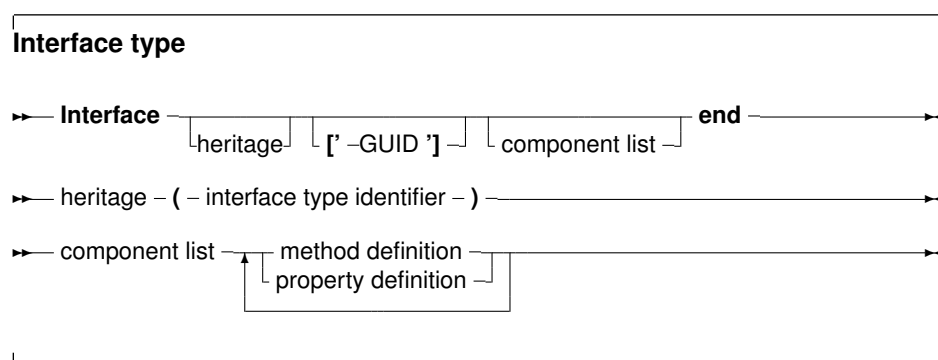
### 7.1 Definition

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<sup>1</sup>. 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).

<sup>1</sup>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):

```
['{HHHHHHHHH-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
```

Normally, the names of the methods that implement an interface, must equal the names of the methods in the interface definition.

However, it is possible to provide aliases for methods that make up an interface: that is, the compiler can be told that a method of an interface is implemented by an existing method with a different name. This is done as follows:

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

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

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

## 7.4 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.

## 7.6 Reference counting

All COM interfaces use reference counting. This means that whenever an interface is assigned to a variable, its reference count is updated. Whenever the variable goes out of scope, the reference count is automatically decreased. When the reference count reaches zero, usually the instance of the class that implements the interface, is freed.

Care must be taken with this mechanism. The compiler may or may not create temporary variables when evaluating expressions, and assign the interface to a temporary variable,

and only then assign the temporary variable to the actual result variable. No assumptions should be made about the number of temporary variables or the time when they are finalized - this may (and indeed does) differ from the way other compilers (e.g. Delphi) handle expressions with interfaces. e.g. a type cast is also an expression:

```
Var
  B : AClass;

begin
  // ...
  AInterface(B.intf).testproc;
  // ...
end;
```

Assume the interface `intf` is reference counted. When the compiler evaluates `B.Intf`, it creates a temporary variable. This variable may be released only when the procedure exits: it is therefore invalid to e.g. free the instance `B` prior to the exit of the procedure, since when the temporary variable is finalized, it will attempt to free `B` again.

## 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.

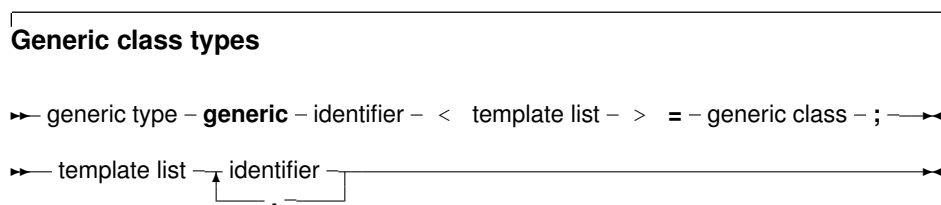
Currently, only generic classes can be defined. Later, support for generic records, functions and arrays may be introduced.

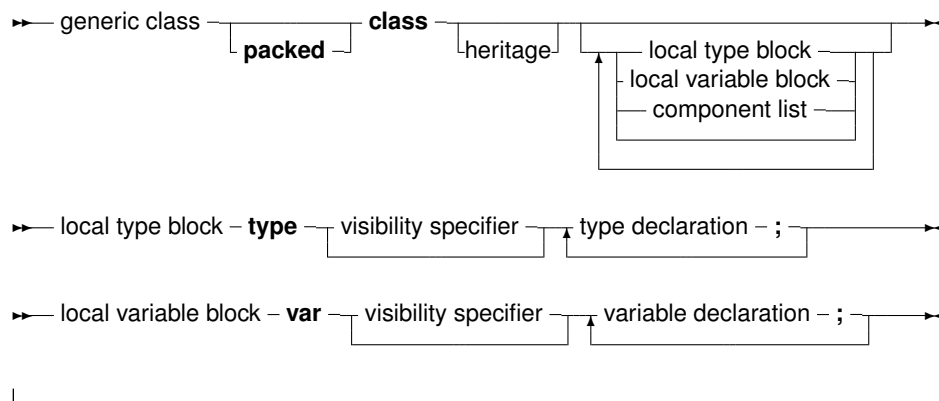
Creating and using generics is a 2-phase process.

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 placeholder. This means that the following would be invalid:

```

procedure TList.Sort(compare: TCompareFunc);

Var
  _t : integer;

begin
  // do something.
end;

```

2. The local type block contains a single type `TCompareFunc`. Note that the actual type is not yet known inside the generic class definition: the definition contains a reference to the placeholder `_T`. All other identifier references must be known when the generic class is defined, *not* when the generic class is specialized.
3. The local variable block is equivalent to the following:

```

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

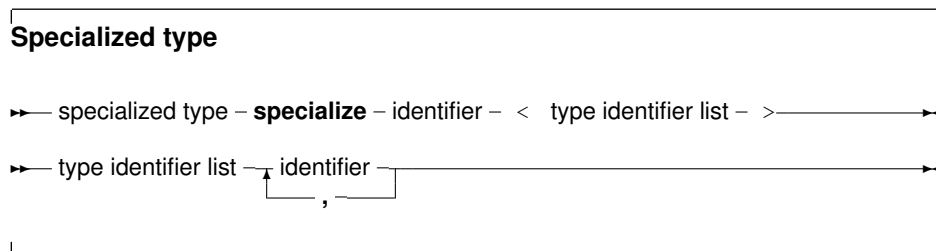
```

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 re-playing 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 determining the precedence, the compiler uses the following rules:

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

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

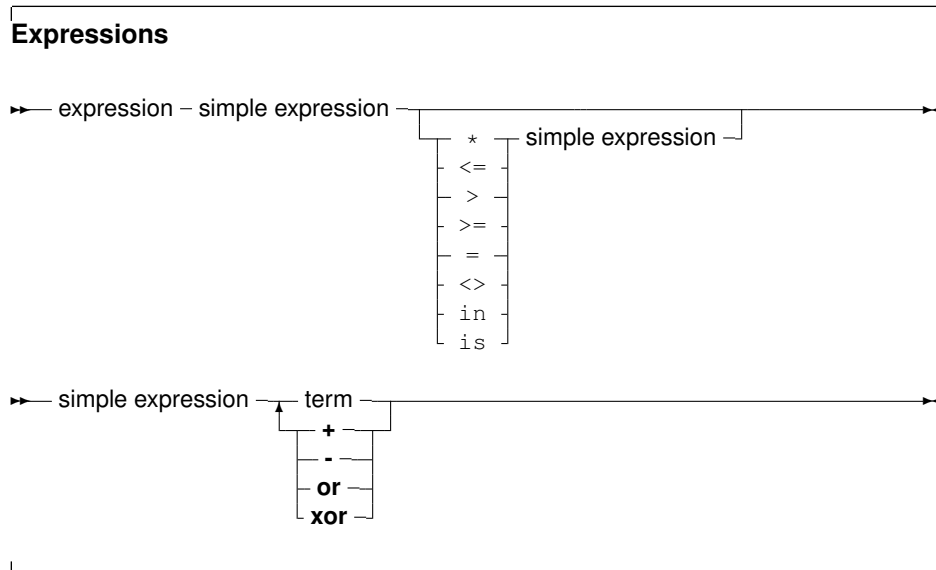
If one expression *must* be executed before the other, it is necessary to split up the statement using temporary results:

```
e1 := g(3);  
a := e1 + f(2);
```

**Remark:** The exponentiation operator (`**`) is available for overloading, but is not defined on any of the standard Pascal types (floats and/or integers).

## 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:

```

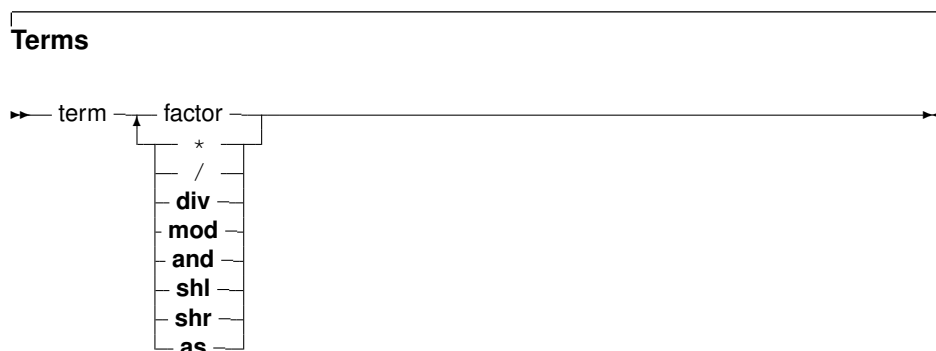
GraphResult<>grError
(DoItToday=Yes) and (DoItTomorrow=No);
Day in Weekend
  
```

And here are some simple expressions:

```

A + B
-Pi
ToBe or NotToBe
  
```

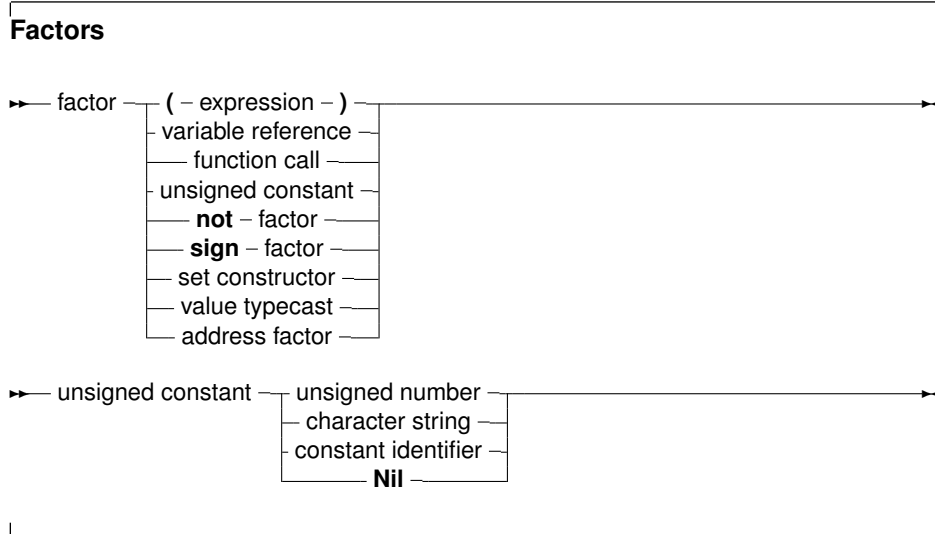
Terms consist of factors, connected by multiplication operators.



Here are some valid terms:

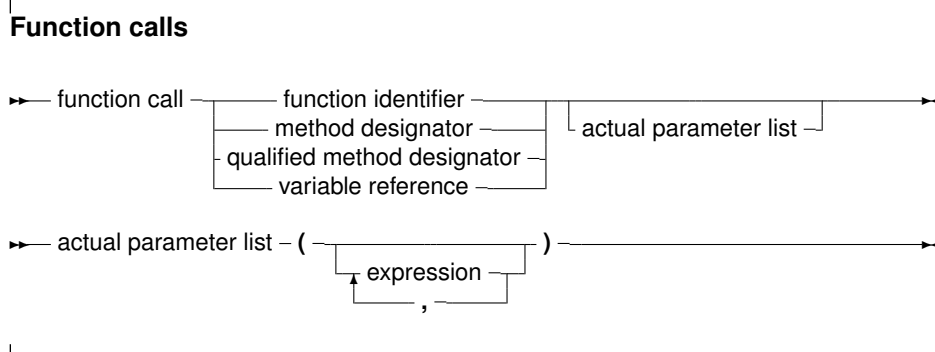
```
2 * Pi
A Div B
(DoItToday=Yes) and (DoItTomorrow=No);
```

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 in Delphi or Turbo Pascal mode:

```
Type
  FuncType = Function: Integer;
Var A : Integer;
Function AddOne : Integer;
begin
  A := A+1;
  AddOne := A;
end;
Var F : FuncType;
    N : Integer;
begin
  A := 0;
  F := AddOne; { Assign AddOne to F, Don't call AddOne}
  N := AddOne; { N := 1 !!}
end.
```

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

A problem with this syntax is the following construction:

```
If F = AddOne Then
  DoSomethingHorrible;
```

Should the compiler compare the addresses of `F` and `AddOne`, or should it call both functions, and compare the result? In `fpc` and `objfpc` mode this is solved by considering a procedural variable as equivalent to a pointer. Thus the compiler will give a type mismatch error, since `AddOne` is considered a call to a function with integer result, and `F` is a pointer.

How then, should one check whether `F` points to the function `AddOne`? To do this, one should use the address operator `@`:

```
If F = @AddOne Then
  WriteLn ('Functions are equal');
```

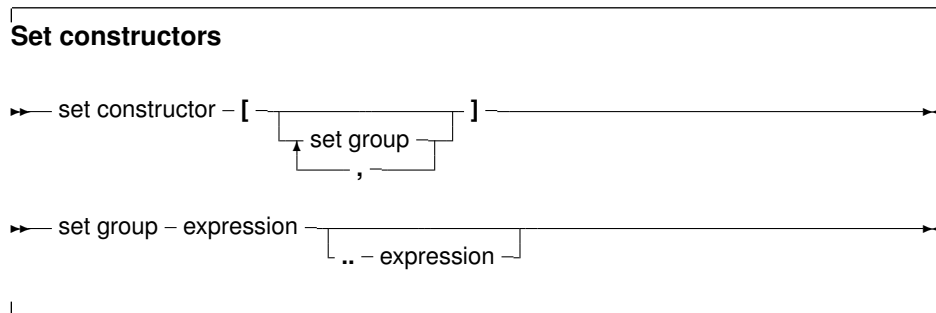
The left hand side of the boolean expression is an address. The right hand side also, and so the compiler compares 2 addresses. How to compare the values that both functions return ? By adding an empty parameter list:

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

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

### 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:

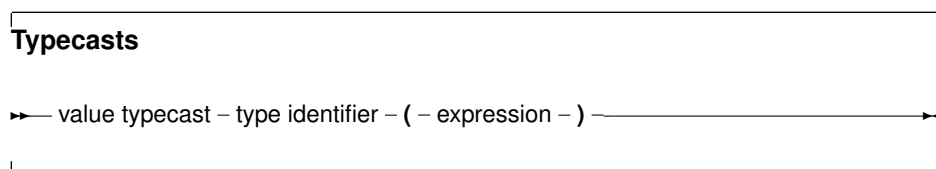
```

[ 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 Delphi or 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:

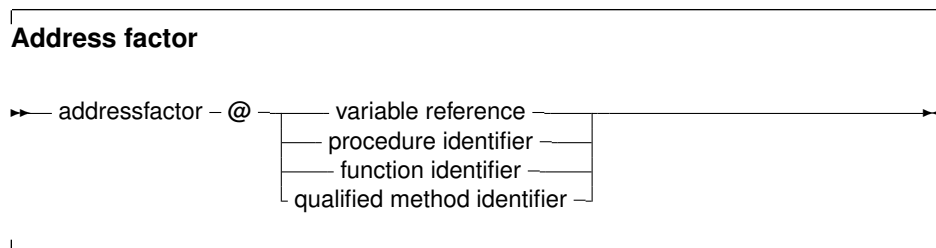
```
program me;

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 assignment compatible with all pointer types. The type of the pointer is  $^T$ , where  $T$  is the type of the variable reference. For example, the following will compile

```
Program tcast;
{$T-} { @ returns untyped pointer }

Type art = Array[1..100] of byte;
Var Buffer : longint;
    PLargeBuffer : ^art;

begin
  PLargeBuffer := @Buffer;
end.
```

Changing the `{T-}` to `{T+}` will prevent the compiler from compiling this. It will give a type mismatch error.

By default, the address operator returns an untyped pointer: applying the address operator to a function, method, or procedure identifier will give a pointer to the entry point of that function. The result is an untyped pointer.

This means that the following will work:

```
Procedure MyProc;

begin
end;

Var
  P : PChar;

begin
  P:=@MyProc;
end;
```

By default, the address operator must be used if a value must be assigned to a procedural type variable. This behaviour can be avoided by using the `-Mtp` or `-MDelphi` switches, which result in a more compatible Delphi or 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

Table 9.2: Binary arithmetic operators

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

of `Div` and `Mod`, which 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 :

$$I \bmod J = I - (I \operatorname{div} J) * J$$

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
<code>not</code>	Bitwise negation (unary)
<code>and</code>	Bitwise and
<code>or</code>	Bitwise or
<code>xor</code>	Bitwise xor
<code>shl</code>	Bitwise shift to the left
<code>shr</code>	Bitwise shift to the right
<code>«</code>	Bitwise shift to the left (same as <code>shl</code> )
<code>»</code>	Bitwise shift to the right (same as <code>shr</code> )

```
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
<code>not</code>	logical negation (unary)
<code>and</code>	logical and
<code>or</code>	logical or
<code>xor</code>	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 added 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 operands must be the same, or an error will be generated by the compiler.

The following program gives some valid examples of set operations:

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
in	check whether an element is in a set

```

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] ;
```

The `In` operation results in a `True` if the left operand (an element) is included of the right operand (a set), the result will be `False` otherwise.

### 9.8.6 Relational operators

The relational operators are listed in table (9.7) Normally, left and right operands must be of

Table 9.7: Relational operators

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

the same type. There are some notable exceptions, where the compiler can handle mixed expressions:

1. Integer and real types can be mixed in relational expressions.
2. If the operator is overloaded, and an overloaded version exists whose arguments types match the types in the expression.
3. Short-, Ansi- and widestring types can be mixed.

Comparing strings is done on the basis of their character code representation.

When comparing pointers, the addresses to which they point are compared. This also is true for `PChar` type pointers. To compare the strings the `Pchar` point to, the `StrComp` function from the strings unit must be used. The `in` returns `True` if the left operand (which must have the same ordinal type as the set type, and which must be in the range 0..255) is an element of the set which is the right operand, otherwise it returns `False`

### 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

Table 9.8: Class operators

Operator	Action
<code>is</code>	Checks class type
<code>as</code>	Conditional typecast

boolean type. The `is` operator can only be used with a class reference or a class instance. The usage of this operator is as follows:

```
Object is Class
```

This expression is completely equivalent to

```
Object.InheritsFrom(Class)
```

If `Object is Nil`, `False` will be returned.

The following are examples:

```
Var
  A : TObject;
  B : TClass;

begin
  if A is TComponent then ;
  If A is B then;
end;
```

The `as` operator performs a conditional typecast. It results in an expression that has the type of the class:

```
Object as Class
```

This is equivalent to the following statements:

```
If Object=Nil then
  Result:=Nil
else if Object is Class then
  Result:=Class(Object)
else
  Raise Exception.Create(SErrInvalidTypeCast);
```

Note that if the object is `nil`, the `as` operator does not generate an exception.

The following are some examples of the use of the `as` operator:

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

```
begin
  (C as TEdit).Text:='Some text';
  C:=0 as TComponent;
end;
```









































































































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.









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.





