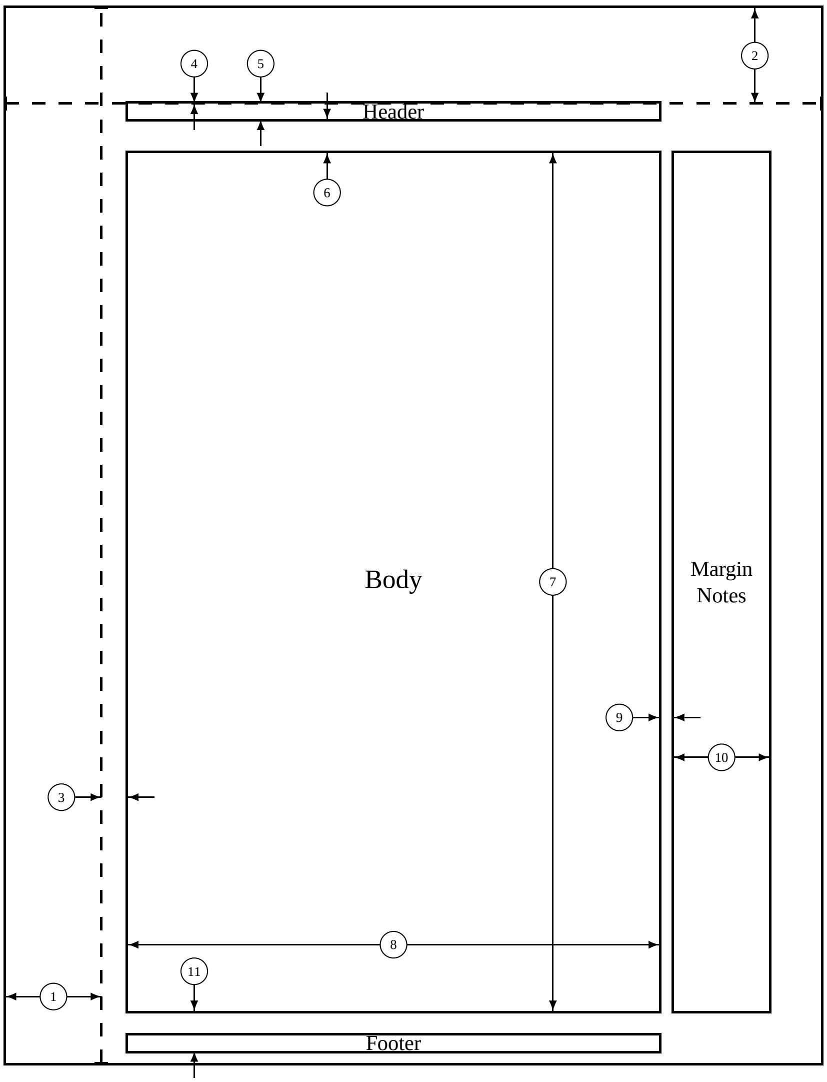
Free Pascal :
Reference guide.

Reference guide for Free Pascal, version 1.0.0
1.8
July 2000

Michaël Van Canneyt

Contents

List of Tables




```
 1   one inch + \hoffset        2   one inch + \voffset
 3   \oddsidemargin = 20pt      4   \topmargin = 0pt
 5   \headheight = 12pt         6   \headsep = 25pt
 7   \textheight = 646pt        8   \textwidth = 400pt
 9   \marginparsep = 11pt      10   \marginparwidth = 72pt
11   \footskip = 30pt               \marginparpush = 5pt (not shown)
     \hoffset = 0pt                 \voffset = 0pt
     \paperwidth = 614pt            \paperheight = 794pt
```

About this guide

This document describes all constants, types, variables, functions and procedures as they are declared in the system unit. Furthermore, 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. The aim is to list which Pascal constructs are supported, and to show where the Free Pascal implementation differs from the Turbo Pascal implementation.

Notations

Throughout this document, we will refer to functions, types and variables with `typewriter` font. Functions and procedures have their own subsections, and for each function or procedure we have the following topics:

Declaration The exact declaration of the function.

Description What does the procedure exactly do ?

Errors What errors can occur.

See Also Cross references to other related functions/commands.

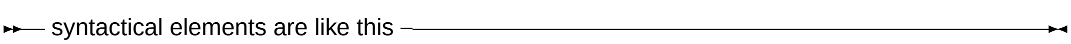
The cross-references come in two flavours:

- References to other functions in this manual. In the printed copy, a number will appear after this reference. It refers to the page where this function is explained. In the on-line help pages, this is a hyperlink, on which you can click to jump to the declaration.
- References to Unix manual pages. (For linux related things only) they are printed in `typewriter` font, and the number after it is the Unix manual section.

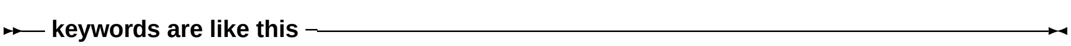
Syntax diagrams

All elements of the pascal language are explained in syntax diagrams. Syntax diagrams are like flow charts. Reading a syntax diagram means that you must get from the left side to the right side, following the arrows. When you are at the right of a syntax diagram, and it ends with a single arrow, this means the syntax diagram is continued on the next line. If the line ends on 2 arrows pointing to each other, then the diagram is ended.

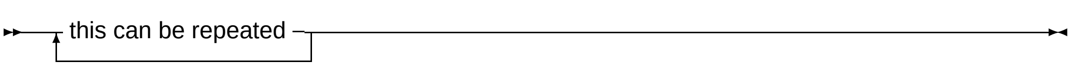
Syntactical elements are written like this



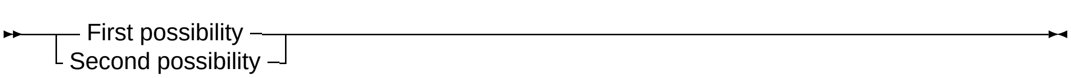
Keywords you must type exactly as in the diagram:



When you can repeat something there is an arrow around it:



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



Note, that one of the possibilities can be empty:

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

Part I

The Pascal language

Chapter 1

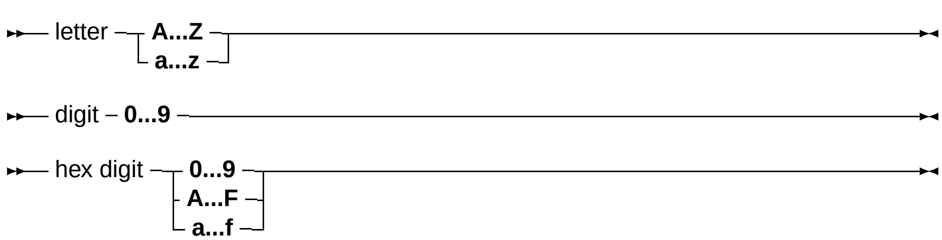
Pascal Tokens

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

1.1 Symbols

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

Recognised symbols



The following characters have a special meaning:

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

and the following character pairs too:

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

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

1.2 Comments

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

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

The following are valid ways of nesting comments:

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

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

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

and

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

The compiler will react with a 'invalid character' error when it encounters such constructs, regardless of the `-So` switch.

1.3 Reserved words

Reserved words are part of the Pascal language, and cannot be redefined. They will be denoted as **this** throughout the syntax diagrams. Reserved words can be typed regardless of case, i.e. Pascal is case insensitive. We make a distinction between Turbo Pascal and Delphi reserved words, since with the `-So` switch, only the Turbo Pascal reserved words are recognised, and the Delphi ones can be redefined. By default, Free Pascal recognises the Delphi reserved words.

Turbo Pascal reserved words

The following keywords exist in Turbo Pascal mode

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

Delphi reserved words

The Delphi (II) reserved words are the same as the pascal ones, plus the following ones:

`as`	`finalization`	`library`	`try`
`class`	`finally`	`on`	
`except`	`initialization`	`property`	
`exports`	`is`	`raise`	

Free Pascal reserved words

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

`dispose`	`false`	`true`
`exit`	`new`	

Modifiers

The following is a list of all modifiers. Contrary to Delphi, Free Pascal doesn't allow you to redefine these modifiers.

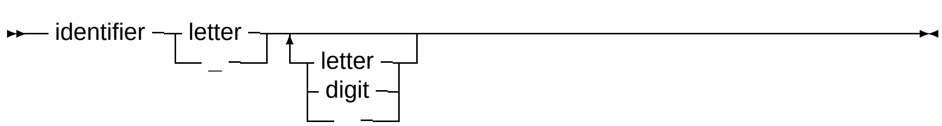
`absolute`	`external`	`pascal`	`register`
`abstract`	`far`	`popstack`	`saveregisters`
`alias`	`forward`	`private`	`stdcall`
`assembler`	`index`	`protected`	`virtual`
`cdecl`	`name`	`public`	`write`
`default`	`near`	`published`	
`export`	`override`	`read`	

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 you can redefine these types. You are, however, not encouraged to do this, as it will cause a lot of confusion.

1.4 Identifiers

Identifiers denote constants, types, variables, procedures and functions, units, and programs. All names of things that you define are identifiers. An identifier consists of 255 significant characters (letters, digits and the underscore character), from which the first must be an alphanumeric character, or an underscore (_) The following diagram gives the basic syntax for identifiers.

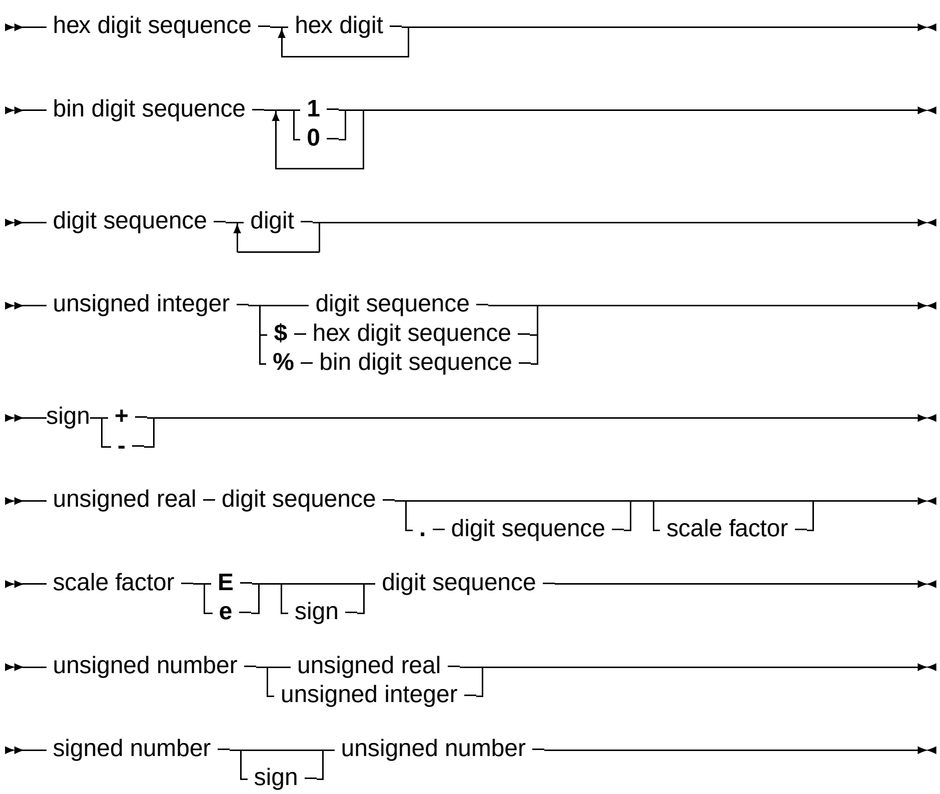
Identifiers



1.5 Numbers

Numbers are denoted in decimal notation. Real (or decimal) numbers are written using engeneering notation (e.g. `0.314E1`). Free Pascal supports hexadecimal format 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. In addition to the support for hexadecimal notation, Free Pascal also supports binary notation. You can specify a binary number 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.

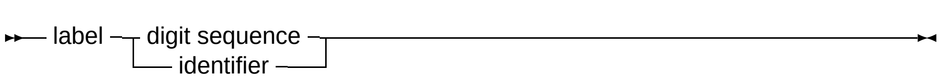
Numbers



1.6 Labels

Labels can be digit sequences or identifiers.

Label

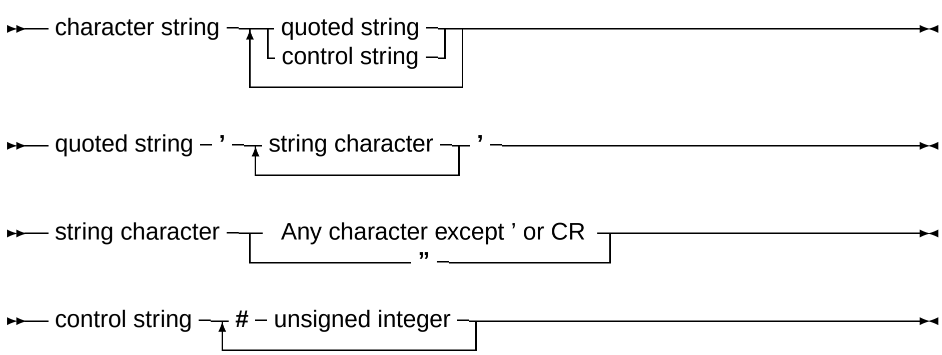


Remark: Note that you must specify the `-Sg` switch before you can use labels. By default, Free Pascal doesn't support `label` and `goto` statements.

1.7 Character strings

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

Character strings



Chapter 2

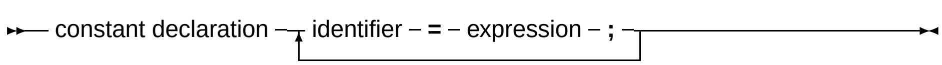
Constants

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

2.1 Ordinary constants

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

Constant declaration



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` can be used, however. For more information on expressions, see chapter 6, page 54. You can only declare constants of the following types: `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);
```

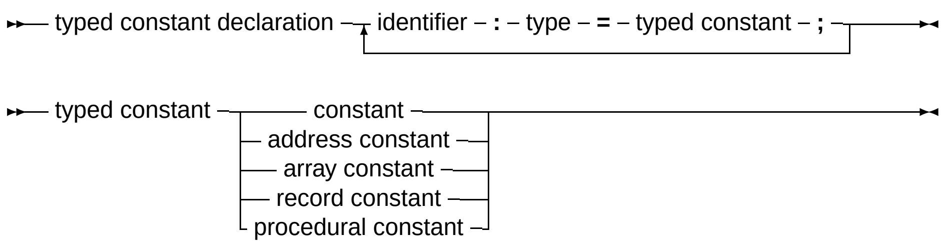
Assigning a value to an ordinary constant is not permitted. Thus, given the previous declaration, the following will result in a compiler error:

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

2.2 Typed constants

Typed constants serve to provide a program with initialised variables. Contrary to ordinary constants, they may be assigned to at run-time. The difference with normal variables is that their value is initialised when the program starts, whereas normal variables must be initialised explicitly.

Typed constant declaration



Given the declaration:

```
Const
  S : String = 'This is a typed constant string';
```

The following is a valid assignment:

```
 S := 'Result : '+Func;
```

Where `Func` is a function that returns a `String`. Typed constants also allow you to initialize arrays and records. For arrays, the initial elements must be specified, surrounded by round brackets, and separated by commas. The number of elements must be exactly the same as the number of elements in the declaration of the type. As an example:

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

For constant records, you should specify each element of the record, in the form `Field : Value`, separated by commas, and surrounded by round brackets. As an example:

```
Type
  Point = record
    X,Y : Real
    end;
Const
  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 you'll get a compile-time error.

2.3 Resource strings

A special kind of constant declaration part is the `Resourestring` part. This part is like a `Const` section, but it only allows to declare constant of type string. This part is only available in the `Delphi` or `objfpc` mode.

The following is an example of a resourcestring definition:

```
Resourcestring

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

All string constants defined in the resourcestring section are stored in special tables, allowing to manipulate the values of the strings at runtime with some special mechanisms.

Semantically, the strings are like constants; you cannot assign values to them, except through the special mechanisms in the objpas unit. However, you can use them in assignments or expressions as normal constants. The main use of the resourcestring section is to provide an easy means of internationalization.

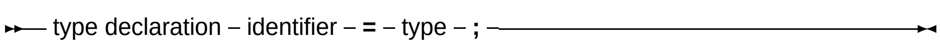
More on the subject of resourcestrings can be found in the Programmers' guide, and in the chapter on the objpas later in this manual.

Chapter 3

Types

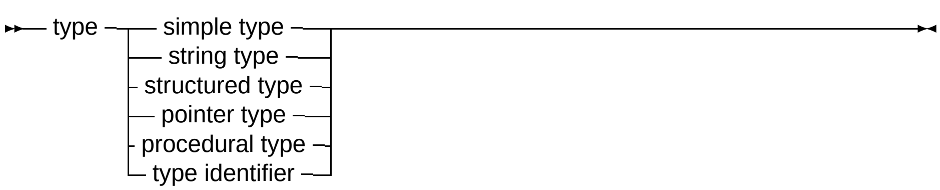
All variables have a type. Free Pascal supports the same basic types as Turbo Pascal, with some extra types from Delphi. You can declare your own types, which is in essence defining an identifier that can be used to denote your custom type when declaring variables further in the source code.

Type declaration



There are 7 major type classes :

Types



The last class, type identifier, is just a means to give another name to a type. This gives you a way to make types platform independent, by only using your own types, and then defining these types for each platform individually. The programmer that uses your units doesn't have to worry about type size and so on. It also allows you to use shortcut names for fully qualified type names. You can e.g. define `system.longint` as `Olongint` and then redefine `longint`.

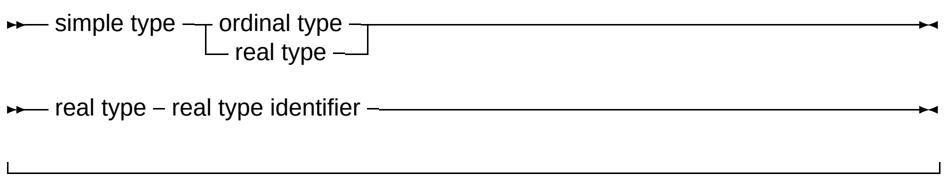
3.1 Base types

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

Simple types

Table 3.1: Predefined ordinal types

Name
Integer
Shortint
SmallInt
Longint
Byte
Word
Cardinal
Boolean
ByteBool
LongBool
Char



Ordinal types

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

1. Ordinal types are countable and ordered, i.e. it is, in principle, possible to start counting them one bye one, in a specified order. This property allows the operation of functions as Inc (136), Ord (146), Dec (124) on ordinal types to be defined.
2. Ordinal values have a smallest possible value. Trying to apply the Pred (148) 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 (162) function on the largest possible value will generate a range check error if range checking is enabled.

Integers

A list of pre-defined ordinal types is presented in table (3.1) The integer types, and their ranges and sizes, that are predefined in Free Pascal are listed in table (3.2). 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`. It also supports the `ByteBool`, `WordBool` and `LongBool` types. 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. Assuming `B` to be of type `Boolean`, the following are valid assignments:

Table 3.2: Predefined integer types

Type	Range	Size in bytes
Byte	0 .. 255	1
Shortint	-127 .. 127	1
Integer	-32768 .. 32767	2[1]
Word	0 .. 65535	2
Longint	-2147483648 .. 2147483648	4
Cardinal	0..4294967296	4

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

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

Boolean expressions are also used in conditions.

Remark: In Free Pascal, boolean expressions are always evaluated in such a way that when the result is known, the rest of the expression will no longer be evaluated (Called short-cut evaluation). In the following example, the function `Func` will never be called, which may have strange side-effects.

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

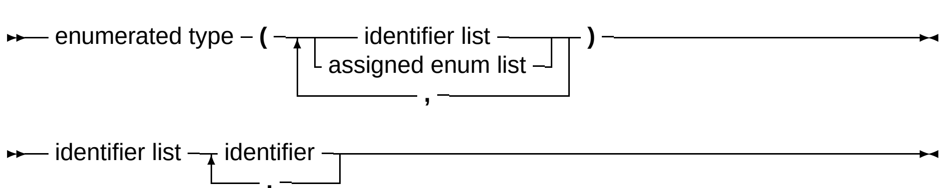
Here `Func` is a function which returns a `Boolean` type.

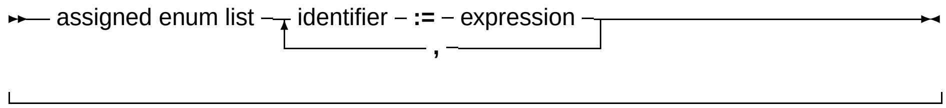
Remark: The `WordBool`, `LongBool` and `ByteBool` types were not supported by Free Pascal until version 0.99.6.

Enumeration types

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

Enumerated types





(see chapter 6, page 54 for how to use expressions) When using assigned enumerated types, the assigned elements must be in ascending numerical order in the list, or the compiler will complain. The expressions used in assigned enumerated elements must be known at compile time. So the following is a correct enumerated type declaration:

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

The C style enumeration type looks as follows:

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

As a result, the ordinal number of `forty` is `40`, and not `3`, as it would be when the `':= 40'` wasn't present. The ordinal value of `fortyone` is then 41, and not `4`, as it would be when the assignment wasn't present. After an assignment in an enumerated definition the compiler adds 1 to the assigned value to assign to the next enumerated value. When specifying such an enumeration type, it is important to keep in mind that you should keep the enumerated elements in ascending order. The following will produce a compiler error:

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

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

1. You cannot use the `Pred` and `Succ` functions on this kind of enumeration types. If you try to do that, you'll get a compiler error.

2. Enumeration types are by default stored in 4 bytes. You can change this behaviour with the `{$PACKENUM n}` compiler directive, which tells the compiler the minimal number of bytes to be used for enumeration types. For instance

   ```
   Type
     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 Programmers' guide, in the compiler directives section.

Table 3.4: Supported Real types

Type	Range	Significant digits	Size[2]
Single	1.5E-45 .. 3.4E38	7-8	4
Real	5.0E-324 .. 1.7E308	15-16	8
Double	5.0E-324 .. 1.7E308	15-16	8
Extended	1.9E-4951 .. 1.1E4932	19-20	10
Comp	-2E64+1 .. 2E63-1	19-20	8

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 it's 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;
```

But you can also define subrange types of enumeration types:

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

Real types

Free Pascal uses the math coprocessor (or an emulation) for all its floating-point calculations. The Real native type is processor dependant, 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). Until version 0.9.1 of the compiler, all the `Real` types were mapped to type `Double`, meaning that they all have size 8. The SizeOf (159) function is your friend here. The `Real` type of turbo pascal is automatically mapped to Double. The `Comp` type is, in effect, a 64-bit integer.

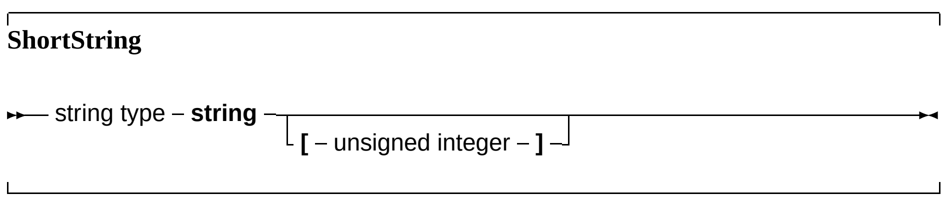
3.2 Character types

Char

Free Pascal supports the type `Char`. A `Char` is exactly 1 byte in size, and contains one character. You can specify a character constant by enclosing the character in single quotes, as follows : 'a' or 'A' are both character constants. You can also specify a character by their ASCII value, by preceding the ASCII 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.) If you want to represent the single quote character, type it two times successively, thus `''` represents the single quote character.

Strings

Free Pascal supports the `String` type as it is defined in Turbo Pascal and it supports ansistrings as in Delphi. To declare a variable as a string, use the following type specification:



The meaning of a string declaration statement is interpreted differently depending on the `{$H}` switch. The above declaration can declare an ansistrng 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 coversions. Note, however, that the result of an expression that contains ansistrings and short strings will always be an ansistring.

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 length specifier, the declaration is a short string declaration.

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

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

For short strings Free Pascal reserves `Size+1` bytes for the string `S`, and in the zeroeth element of the string (`S[0]`) it will store the length of the variable. If you don't specify the size of the string, `255` is taken as a default. For example in

```
{$H-}

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

Table 3.5: AnsiString memory structure

Offset	Contains
-12	Longint with maximum string size.
-8	Longint with actual string size.
-4	Longint with reference count.
0	Actual string, null-terminated.

`NameString` can contain maximum 10 characters. While `StreetString` can contain 255 characters. The sizes of these variables are, respectively, 11 and 256 bytes.

Ansistrings

If the `{$H}` switch is on, then a string definition that doesn't contain a length specifier, will be regarded as an ansistring.

Ansistrings are strings that have no length limit. They are reference counted. Internally, an ansistring is treated as a pointer.

If the string is empty (''), then the pointer is nil. If the string is not empty, then the pointer points to a structure in heap memory that looks as in table (3.5).

Because of this structure, it is 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.

AnsiStrings can be unlimited in length. Since the length is stored, the length of an ansistring is available immediatly, providing for fast access.

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 by one, The referece count of `S1` is increased by one, and finally `S1` (as a pointer) is copied to `S2`. This is a significant speed-up in your code.

If a reference count reaches zero, then the memory occupied by the string is deallocated automatically, 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, global or 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 100,000 times `nil` into `A`. When `A` goes out of scope, then the 100,000 strings will be dereferenced one by one. All this happens invisibly for the programmer, but when considering performance issues, this is important.

Memory will be allocated only when the string is assigned a value. If the string goes out of scope, then it is automatically dereferenced.

If you assign a value 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 impossible to access the length of an ansistring by referring to the zeroeth character. The following statement will generate a compiler error if S is an ansistring:

```
Len:=S[0];
```

Instead, you must use the Length (140) function to get the length of a string.

To set the length of an ansistring, you can use the SetLength (157) function. Constant ansistrings have a reference count of -1 and are treated specially.

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

You can typecast ansistrings 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. If you typecast an empty ansistring to a pointer, the pointer wil be `Nil`. If you typecast an empty ansistring 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. 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 0. (call uniquestring if you want to ensure a string has reference count 1)

Constant strings

To specify a constant string, you enclose the string in single-quotes, just as a `Char` type, only now you can have more than one character. 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 you can see, the single quote character is represented by 2 single-quote characters next to each other. Strange characters can be specified by their ASCII value. The example shows also that you can add two strings. The resulting string is just the concatenation of the first with the second string, without spaces in between them. Strings can not be substracted, however.

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

Table 3.6: `PChar` pointer arithmetic

Operation	Result
`P + I`	Adds `I` to the address pointed to by `P`.
`I + P`	Adds `I` to the address pointed to by `P`.
`P - I`	Substracts `I` from the address pointed to by `P`.
`P - Q`	Returns, as an integer, the distance between 2 addresses (or the number of characters between `P` and `Q`)

PChar

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 you can do it in C. Since it is equivalent to a pointer to a type `Char` variable, it is also possible to do the following:

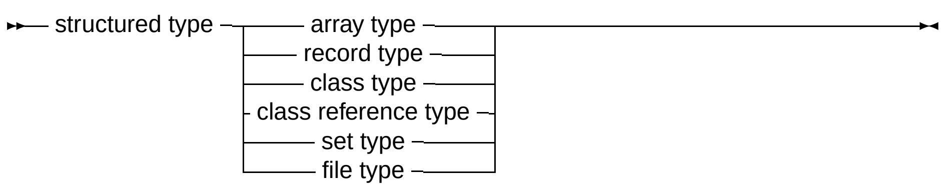
```
Program three;
Var S : String[30];
    P : PChar;
begin
  S := 'This is a null-terminated string.'#0;
  P := @S[1];
  WriteLn (P);
end.
```

This will have the same result as the previous two examples. You cannot add null-terminated strings as you can do with normal Pascal strings. If you want to concatenate two `PChar` strings, you will need to use the unit strings. However, it is possible to do some pointer arithmetic. You can use the operators `+` and `-` to do operations on `PChar` pointers. In table (3.6), `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. Stuctured types can be nested to unlimited levels.

Structured Types

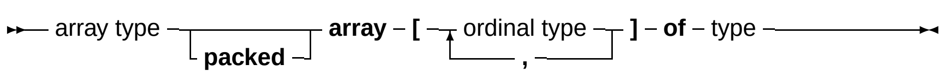


Unlike Delphi, Free Pascal does not support the keyword `Packed` for all structured types, as can be seen in the syntax diagram. It will be mentioned when a type supports the `packed` keyword. In the following, each of the possible structured types is discussed.

Arrays

Free Pascal supports arrays as in Turbo Pascal, multi-dimensional arrays and packed arrays are also supported:

Array types



The following is a valid array declaration:

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

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 following declaration:

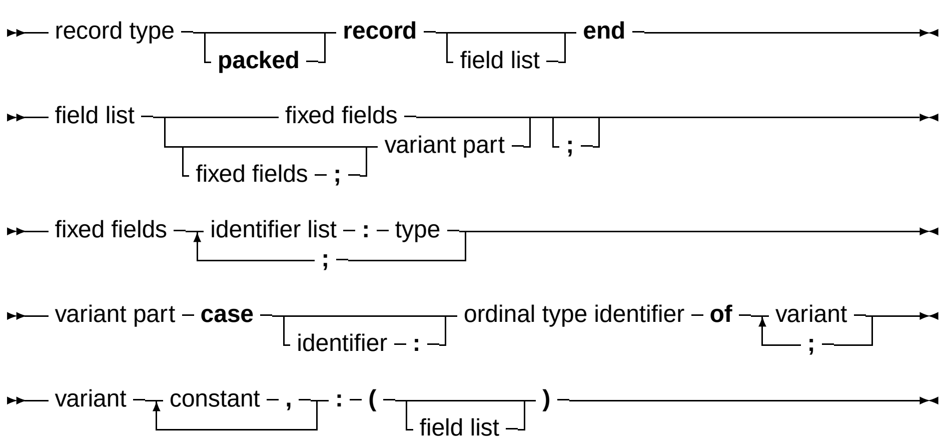
```
Type
   APoints = array[1..100,1..3] of Real;
```

The functions High (135) and Low (141) return the high and low bounds of the leftmost index type of the array. In the above case, this would be 100 and 1.

Record types

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

Record types



So the following are valid record types declarations:

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

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

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 (159) would return 24 for `Point`, 24 for `RPoint` and 26 for `BetterRPoint`. For `MyRec`, the value would be 12. If you want to read a typed file with records, produced by a Turbo Pascal program, then chances are that you will not succeed in reading that file correctly. The reason for this is that by default, elements of a record are aligned at 2-byte boundaries, for performance reasons. This default behaviour can be changed with the `{$PackRecords n}` switch. Possible values for `n` are 1, 2, 4, 16 or `Default`. This switch tells the compiler to align elements of a record or object or class that have size larger than `n` on `n` byte boundaries. Elements that have size smaller or equal than `n` are aligned on natural boundaries, i.e. to the first power of two that is larger than or equal to the size of the record element. The keyword `Default` selects the default value for the platform you're working on (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 it's size. The same holds for `Trec6`. For `Trec7`, `B` is aligned on a 4 byte boundary, since it's 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. As from version 0.9.3, 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 !

Set types

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

Set Types

set type – **set** – **of** – ordinal type

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 maximally `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 this set declarations, the following assignment is legal:

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

The operators and functions for manipulations of sets are listed in table (3.7). You can compare two sets with the <> and = operators, but not (yet) with the < and > operators. As of compiler version 0.9.5, the compiler stores small sets (less than 32 elements) in a Longint, if the type range allows it. This allows for faster processing and decreases program size. Otherwise, sets are stored in 32 bytes.

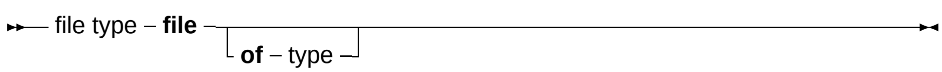
Table 3.7: Set Manipulation operators

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

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. Nothing stops you, however, from writing a file driver that stores it's data in memory. Here is the type declaration for a file type:

File types



If no type identifier is given, then the file is an untyped file; it can be considered as equivalent to a file of bytes. Untyped files require special commands to act on them (see Blockread (118), Blockwrite (119)). The following declaration declares a file of records:

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

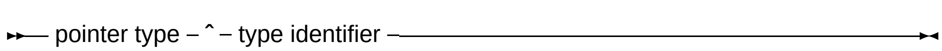
Internally, files are represented by the `FileRec` record, which is declared in the DOS unit.

A special file type is the `Text` file type, represented by the `TextRec` record. A file of type `Text` uses special input-output routines.

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.

Pointer types



As can be seen from this diagram, pointers are typed, which means that they point to a particular kind of data. The type of this data must be known at compile time. Dereferencing the pointer (denoted by adding ˆ 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 bytes of memory (enough to keep an adress in memory).

Remark: Free Pascal treats pointers much the same way as C does. This means that you can treat a pointer to some type as being an array of this type. 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 you use the former, you must allocate memory yourself, using the Getmem (134) function. The reference `Pˆ` 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 descrease 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. You can also use normal arithmetic operators on pointers, that is, the following are valid pointer arithmetic operations:

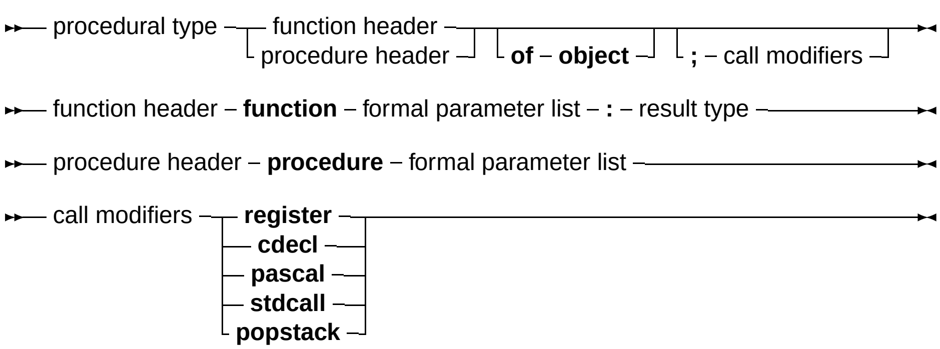
```
var  p1,p2 : ^Longint;
     L : Longint;
begin
  P1 := @P2;
  P2 := @L;
  L := P1-P2;
  P1 := P1-4;
  P2 := P2+4;
end.
```

Here, the value that is added or substracted is *not* multiplied by the size of the type the pointer points to.

3.5 Procedural types

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

Procedural types



For a description of formal parameter lists, see chapter 8, page 74. 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;
...
P := @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 (unless you use the `-So` switch, in which case you can drop the address operator.)

Remark: The modifiers concerning the calling conventions (`cdecl`, `pascal`, `stdcall` and `popstack` stick to 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
P := @printit;
end.
```

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

Chapter 4

Objects

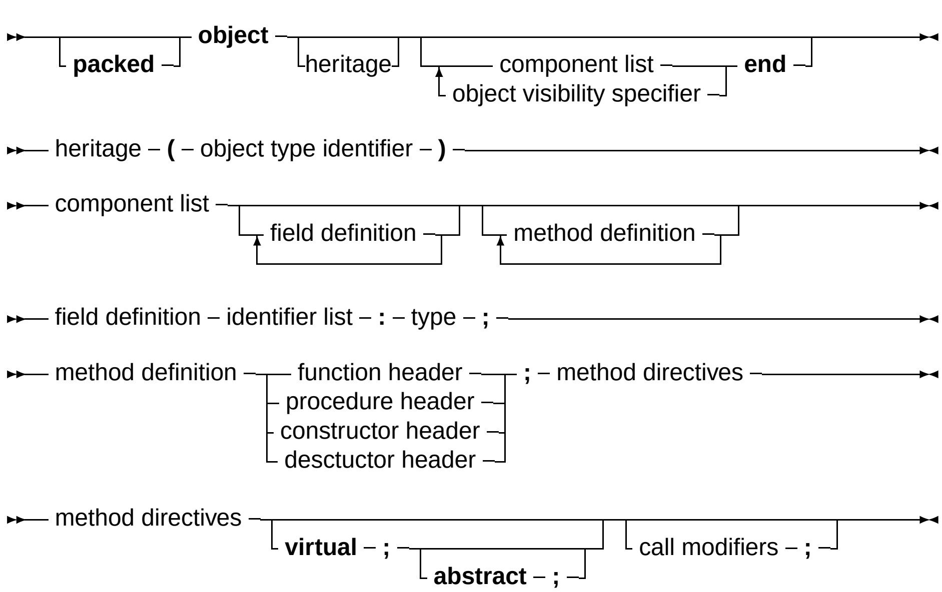
4.1 Declaration

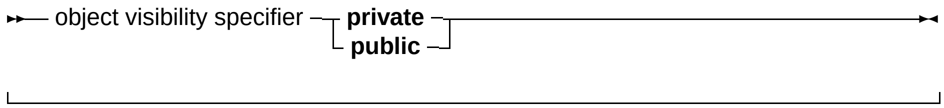
Free Pascal supports object oriented programming. In fact, most of the compiler is written using objects. Here we present some technical questions regarding object oriented programming in Free Pascal. Objects should be treated as a special kind of record. The record contains all the fields that are declared in the objects definition, and pointers to the methods that are associated to the objects' type.

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

Furthermore, you can declare fields, procedures and functions as `public` 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. The prototype declaration of an object is as follows:

object types





As you can see, you can repeat as many `private` and `public` blocks as you want. `Method definitions` are normal function or procedure declarations. You cannot put fields after methods in the same block, i.e. the following will generate an error when compiling:

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

But the following will be accepted:

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

because the field is in a different section.

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.

4.2 Fields

Object Fields are like record fields. They are accessed in the same way as you would access a record field : 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;
```

You cannot access fields that are in a private section of an object from outside the objects' methods. If you do, the compiler will complain about an unknown identifier. It is also possible to use the `with` statement with an object instance:

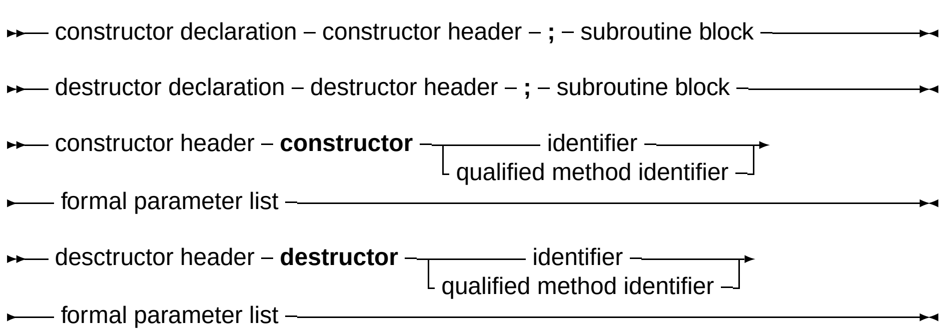
```
With AnObject do
  begin
  Afield := 12;
  AMethod;
  end;
```

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 7.2, page 70

4.3 Constructors and destructors

As can be seen in the syntax diagram for an object declaration, Free Pascal supports constructors and destructors. You are responsible for calling the constructor and the destructor explicitly when using objects. The declaration of a constructor or destructor is as follows:

Constructors and destructors



A constructor/destructor pair is *required* if you use virtual methods. In the declaration of the object type, you should use a simple identifier for the name of the constuctor or destructor. When you implement the constructor or destructor, you should use a qulified method identifier, i.e. an identifier of the form `objectidentifier.methodidentifier`. Free Pascal supports also the extended syntax of the `New` and `Dispose` procedures. In case you want to allocate a dynamic variable of an object type, you can specify the constructor's name in the call to `New`. The `New` is implemented as a function which returns a pointer to the instantiated object. Consider the following declarations:

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

Then the following 3 calls are equivalent:

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

and

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

and also

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

In the last case, the compiler will issue a warning that you should use the extended syntax of `new` and `dispose` to generate instances of an object. You can 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.

4.4 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.

4.5 Method invocation

Methods are called just as normal procedures are called, only they have an object instance identifier prepended to them (see also chapter 7, page 63). To determine which method is called, it is necessary to know the type of the method. We treat the different types in what follows.

Static methods

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

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

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

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

Of the three invocations of `Doit`, only the last one will call `TChild.Doit`, the other two calls will call `TParent.Doit`. This is because for static methods, the compiler determines at compile time which method should be called. Since `ParentB` is of type `TParent`, the compiler decides that it must be called with `TParent.Doit`, even though it will be created as a `TChild`. There may be times when you want the method that is actually called to 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. 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;
   ParentA := New(PParent,Init);
   ParentB := New(PChild,Init);
   Child := New(PChild,Init);
   ParentA^.Doit;
   ParentB^.Doit;
   Child^.Doit;
```

Now, different methods will be called, depending on the actual run-time type of the object. For `ParentA`, nothing changes, since it is created as a `TParent` instance. For `Child`, the situation also doesn't change: it is again created as an instance of `TChild`. For `ParentB` however, the situation does change: Even though it was declared as a `TParent`, it is created as an instance of `TChild`. Now, when the program runs, before calling `Doit`, the program checks what the actual type of `ParentB` is, and only then decides which method must be called. Seeing that `ParentB` is of type `TChild`, `TChild.Doit` will be called. The code for this run-time checking of the actual type of an object is inserted by the compiler at compile time. The `TChild.Doit` is said to *override* the `TParent.Doit`. It is possible to acces the `TParent.Doit` from within the varTChild.Doit, with the `inherited` keyword:

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

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

Abstract methods

An abstract method is a special kind of virtual method. A method can not be abstract if it is not virtual (this is not obvious from the syntax diagram). You cannot create an instance of an object that has an abstract method. The reason is obvious: there is no method where the compiler could jump to ! A method that is declared `abstract` does not have an implementation for this method. It is up to inherited objects to override and implement this method. Continuing our example, take a look at this:

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

```
    ...
    procedure Doit;virtual;
    ...
    end;
  PChild = ^TChild;
```

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

```
Var ParentA,ParentB : PParent;
    Child           : PChild;
   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 you 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 you override an abstract method, you cannot call the parent method 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)

4.6 Visibility

For objects, only 2 visibility specifiers exist : `private` and `public`. If you don't specify a visibility specifier, `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.

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

Chapter 5

Classes

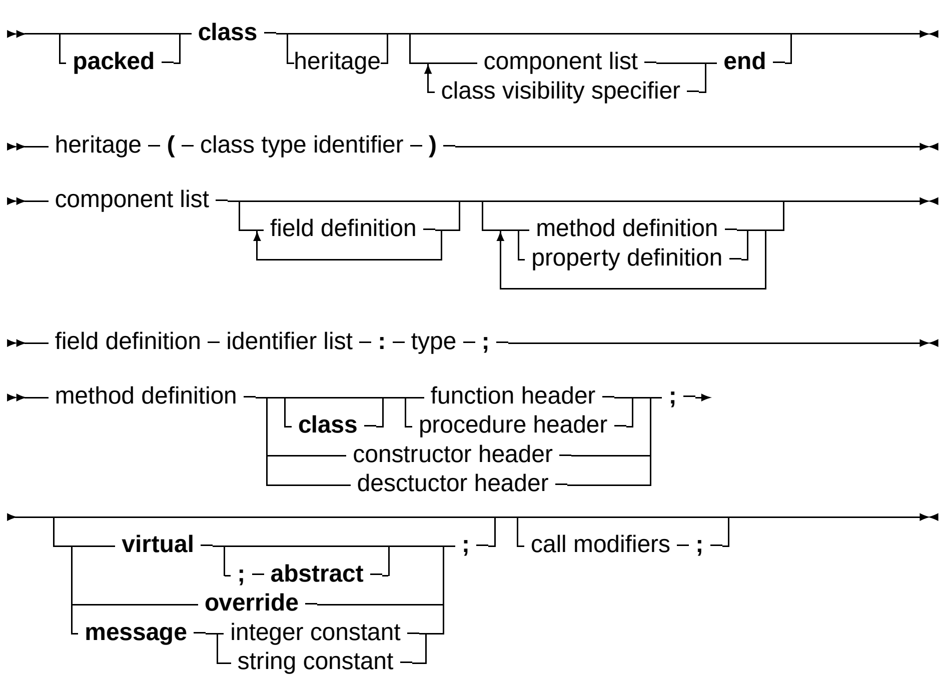
In the Delphi approach to Object Oriented Programming, everything revolves around the concept of 'Classes'. A class can be seen as a pointer to an object, or a pointer to a record.

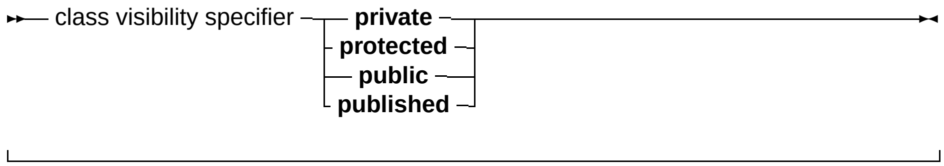
Remark: In earlier versions of Free Pascal it was necessary, in order to use classes, to put the objpas unit in the uses clause of your unit or program. *This is no longer needed* as of version 0.99.12. As of version 0.99.12 the system unit contains the basic definitions of `TObject` and `TClass`, as well as some auxiliary methods for using classes. The objpas unit still exists, and contains some redefinitions of basic types, so they coincide with Delphi types. The unit will be loaded automatically if you specify the `-S2` or `-Sd` options.

5.1 Class definitions

The prototype declaration of a class is as follows :

Class types





Again, You can repeat as many `private`, `protected`, `published` and `public` blocks as you want. Methods are normal function or procedure declarations. As you can see, the declaration of a class is almost identical to the declaration of an object. The real difference between objects and classes is in the way they are created (see further in this chapter). The visibility of the different sections is as follows:

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

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

Public sections are always accessible.

Published Is the same as a `Public` section, but the compiler generates also type information that is needed for automatic streaming of these classes. Fields defined in a `published` section must be of class type. Array peroperties cannot be in a `published` section.

5.2 Class instantiation

Classes must be created using their constructor. Remember that a class is a pointer to an object, so when you declare a variable of some class, the compiler just allocates a pointer, not the entire object. The constructor of a class returns a pointer to an initialized instance of the object. So, to initialize an instance of some class, you would do the following :

```
ClassVar := ClassType.ConstructorName;
```

You cannot use the extended syntax of `new` and `dispose` to instantiate and destroy class instances. That construct is reserved for use with objects only. Calling the constructor will provoke a call to `getmem`, to allocate enough space to hold the class instance data. After that, the constuctor's code is executed. The constructor has a pointer to it's 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, you can declare a packed class. 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 4, since a class is but a pointer to an object. To get the size of the class instance data, use the `TObject.InstanceSize` method.

5.3 Methods

invocation

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

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

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: you *must* override virtual methods 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.

Message methods

New in classes are `message` methods. Pointers to message methods are stored in a special table, together with the integer or string cnstant that they were declared with. They are primarily intended to ease programming of callback functions in several `GUI` toolkits, such as `Win32` or `GTK`. In difference with Delphi, Free Pascal also accepts strings as message identifiers.

Message methods that are declared with an integer constant can take only one var argument (typed or not):

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

The method implementation of a message function is no different from an ordinary method. It is also possible to call a message method directly, but you should not do this. Instead use the `TObject.Dispatch` method.

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 it's ancestors (starting at the object, and searching up the 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 you might need. `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 messae 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 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 it's ancestors (starting at the object, and searching up the 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 you might need. `DefaultHandlerStr` is declared as follows:

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

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

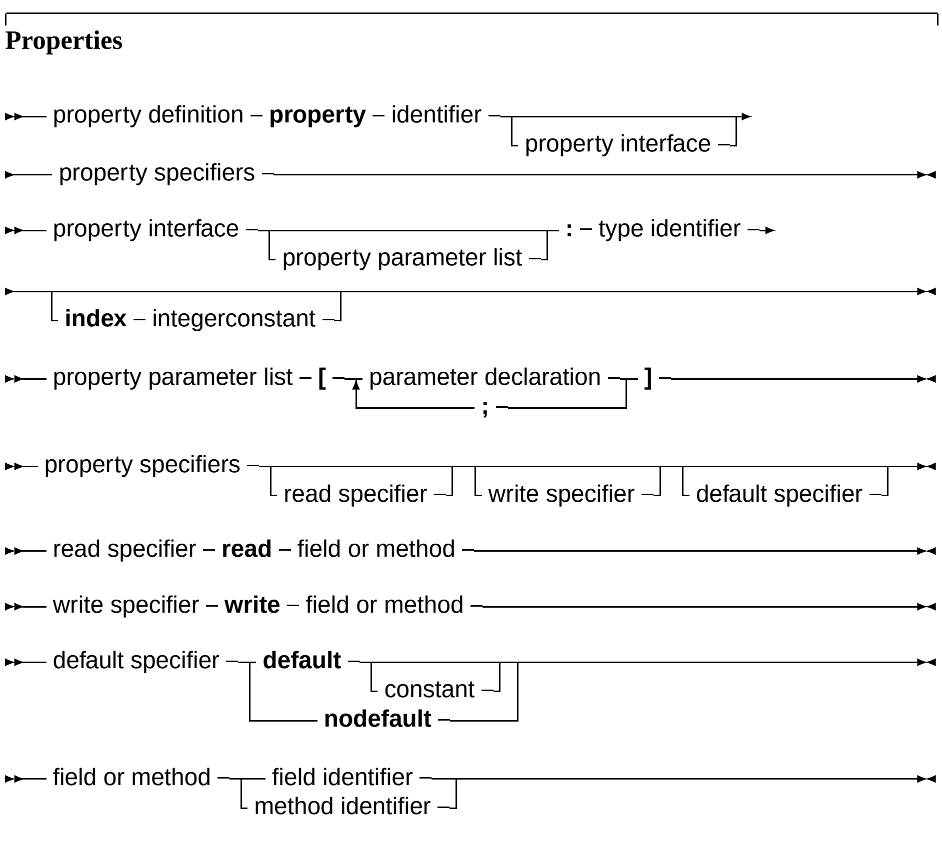
```
TMyObject.StrMsgHandler(Data : Pointer; Self : TMyObject);Message 'OnClick';
```

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 you define the method for.

5.4 Properties

Classes can contain properties as part of their fields list. A property acts like a normal field, i.e. you can get or set it's value, but 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. A `read specifier` is optional, making the property write-only. 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. The section (`private`, `published`) in which the specified function or procedure resides is irrelevant. Usually, however, this will be a protected or private method. Example: Given the following declaration:

```
Type
  MyClass = Class
    Private
    Field1 : Longint;
    Field2 : Longint;
    Field3 : Longint;
    Procedure  Sety (value : Longint);
    Function Gety : Longint;
    Function Getz : Longint;
    Public
    Property X : Longint Read Field1 write Field2;
    Property Y : Longint Read GetY Write Sety;
    Property Z : Longint Read GetZ;
    end;
Var MyClass : TMyClass;
```

The following are valid statements:

```
WriteLn ('X : ',MyClass.X);
WriteLn ('Y : ',MyClass.Y);
WriteLn ('Z : ',MyClass.Z);
MyClass.X := 0;
MyClass.Y := 0;
```

But the following would generate an error:

```
MyClass.Z := 0;
```

because Z is a read-only property. What happens in the above statements is that when a value needs to be read, the compiler inserts a call to the various `getNNN` methods of the object, and the result of this call is used. When an assignment is made, the compiler passes the value that must be assigned as a paramater to the various `setNNN` methods. Because of this mechanism, properties cannot be passed as var arguments to a function or procedure, since there is no known address of the property (at least, not always). If the property definition contains an index, then the read and write specifiers must be a function and a procedure. Moreover, these functions require an additional parameter : An integer parameter. This allows to read or write several properties with the same function. For this, the properties must have the same type. The following is an example of a property with an index:

```
{$mode objfpc}
Type TPoint = Class(TObject)
       Private
       FX,FY : Longint;
       Function GetCoord (Index : Integer): Longint;
       Procedure SetCoord (Index : Integer; Value : longint);
       Public
       Property X : Longint index 1 read GetCoord Write SetCoord;
       Property Y : Longint index 2 read GetCoord Write SetCoord;
       Property Coords[Index : Integer] 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. You can also have array properties. These are properties that accept an index, just as an array does. Only now the index doesn't have to be an ordinal type, but can be any type.

A `read specifier` for an array property is the name method function that has the same return type as the property type. The function must accept as a sole arguent a variable of the same type as the index type. For an array property, you cannot specify fields as `read specifiers`.

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

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

Then the following statements would be valid:

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

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

While the following statements would generate errors:

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

Because the index types are wrong. Array properties can be declared as `default` properties. This means that it is not necessary to specify the property name when assigning or reading it. If, in the previous example, 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;
```

You can have only one default property per class, and descendent classes cannot redeclare the default property.

Chapter 6

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 (6.1) are used. When determining the precedence, the compiler uses the following rules:

1. Operators with equal precedence are executed from left to right.
2. In operations with unequal precedences the operands belong to the operater with the highest precedence. For example, in `5*3+7`, the multiplication is higher in precedence than the addition, so it is executed first. The result would be 22.
3. If parentheses are used in an epression, their contents is evaluated first. Thus, `5*(3+7)` would result in 50.

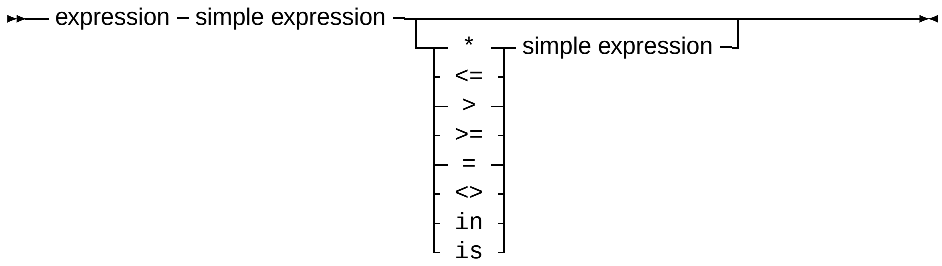
6.1 Expression syntax

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.

Expressions

Table 6.1: 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



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.

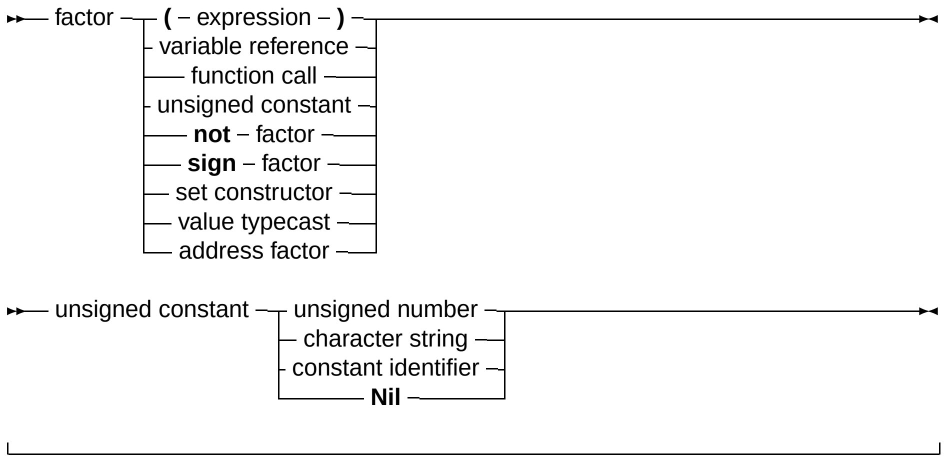
Terms

Here are some valid terms:

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

Factors are all other constructions:

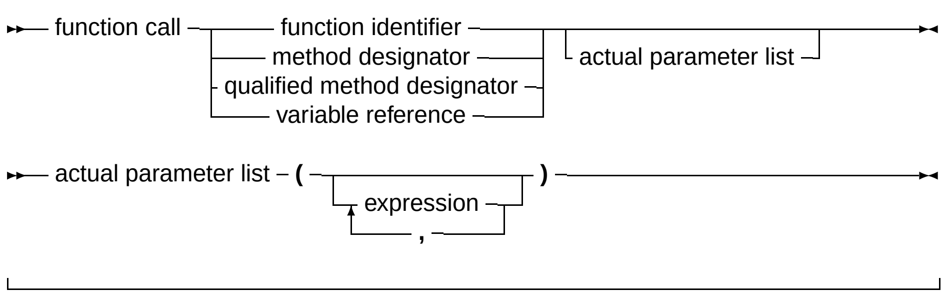
Factors



6.2 Function calls

Function calls are part of expressions (although, using extended syntax, they can be statements too). They are constructed as follows:

Function calls



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.
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. Depending on the fact of the function is overloaded (i.e. multiple functions with the same name, but different parameter lists) the error will be different. There are cases when the compiler will not execute the function call in an expression. This is the case when you are assigning a value to a procedural type variable, as in the following example:

```
Type
  FuncType = Function: Integer;
Var A : Integer;
```

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

In the above listing, the assigment to F will not cause the function AddOne to be called. The assignment to N, however, will call AddOne. A problem with this syntax is the following construction:

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

Should the compiler compare the addresses of `F` and `AddOne`, or should it call both functions, and compare the result ? Free Pascal solves this by deciding that a procedural variable is equivalent to a pointer. Thus the compiler will give a type mismatch error, since AddOne is considered a call to a function with integer result, and F is a pointer, Hence a type mismatch occurs. How then, should one compare whether `F` points to the function `AddOne` ? To do this, one should use the address operator `@`:

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

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

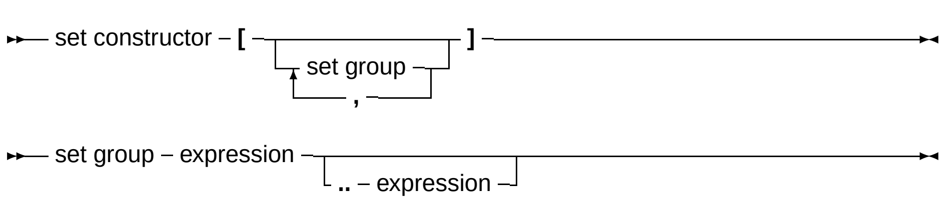
```
  If F()=Addone then
    WriteLn ('Functions return same values ');
```

Remark that this behaviour is not compatible with Delphi syntax.

6.3 Set constructors

When you want to enter a set-type constant in an expression, you must give a set constructor. In essence this is the same thing as when you define a set type, only you have no identifier to identify the set with. A set constructor is a comma separated list of expressions, enclosed in square brackets.

Set constructors



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. If the first range specifier has a bigger ordinal value than the second the set is empty, e.g., `[Z..A]` denotes an empty set. The following are valid set constructors:

```
[today,tomorrow]
[Monday..Friday,Sunday]
[ 2, 3*2, 6*2, 9*2 ]
['A'..'Z','a'..'z','0'..'9']
```

6.4 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:

Typecasts

Value typecasts cannot be used on the left side of assignments, as variable typecasts. Here are some valid typecasts:

```
Byte('A')
Char(48)
boolean(1)
longint(@Buffer)
```

The type size of the expression and the size of the type cast must be the same. That is, the following doesn't work:

```
Integer('A')
Char(4875)
boolean(100)
Word(@Buffer)
```

This is different from Delphi or Turbo Pascal behaviour.

6.5 The @ operator

The address operator @ returns the address of a variable, procedure or function. It is used as follows:

Address factor

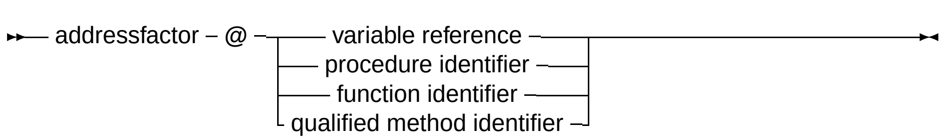


Table 6.2: Binary arithmetic operators

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

The `@` operator returns a typed pointer if the `$T` switch is on. If the `$T` switch is off then the address operator returns an untyped pointer, which is assigment compatible with all pointer types. The type of the pointer is `^T`, where `T` is the type of the variable reference. For example, the following will compile

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

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

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

Changing the `{$T-}` to `{$T+}` will prevent the compiler from compiling this. It will give a type mismatch error. By default, the address operator returns an untyped pointer. Applying the address operator to a function, method, or procedure identifier will give a pointer to the entry point of that function. The result is an untyped pointer. By default, you must use the address operator if you want to assign a value to a procedural type variable. This behaviour can be avoided by using the `-So` or `-S2` switches, which result in a more compatible Delphi or Turbo Pascal syntax.

6.6 Operators

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

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 (6.2), unary operators are listed in table (6.3). With the exception 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. 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

Table 6.3: Unary arithmetic operators

Operator	Operation
`+`	Sign identity
`-`	Sign inversion

Table 6.4: Logical operators

Operator	Operation
`not`	Bitwise negation (unary)
`and`	Bitwise and
`or`	Bitwise or
`xor`	Bitwise xor
`shl`	Bitwise shift to the left
`shr`	Bitwise shift to the right

is zero. The sign of the result of a `Mod` operator is the same as the sign of the left side operand of the `Mod` operator. In fact, the `Mod` operator is equivalent to the following operation :

```
I mod J = I - (I div J) * J
```

but it executes faster than the right hand side expression.

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 (6.4). The following are valid logical expressions:

```
A shr 1  { same as A div 2, but faster}
Not 1    { equals -2 }
Not 0    { equals -1 }
Not -1   { equals 0  }
B shl 2  { same as B * 2 for integers }
1 or 2   { equals 3 }
3 xor 1  { equals 2 }
```

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 (6.5)

Remark: Boolean expressions are ALWAYS 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

Table 6.5: Boolean operators

Operator	Operation
`not`	logical negation (unary)
`and`	logical and
`or`	logical or
`xor`	logical xor

Table 6.6: Set operators

Operator	Action
`+`	Union
`-`	Difference
`*`	Intersection

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)

String operators

There is only one string operator : `+`. It's action is to concatenate the contents of the two strings (or characters) it stands between. You 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.

Set operators

The following operations on sets can be performed with operators: Union, difference and intersection. The operators needed for this are listed in table (6.6). The set type of the operands must be the same, or an error will be generated by the compiler.

Relational operators

The relational operators are listed in table (6.7) Left and right operands must be of the same type. You can only mix integer and real types in relational expressions. Comparing strings is done on the basis of their ASCII code representation. When comparing pointers, the addresses to which they point are compared. This also is true for `PChar` type pointers. If you want to compare the strings the `Pchar` points to, you must use the `StrComp` function from the strings unit. The `in` returns `True` if the left operand (which must have the same ordinal type as the set type) is an element of the set which is the right operand, otherwise it returns `False`

Table 6.7: Relational operators

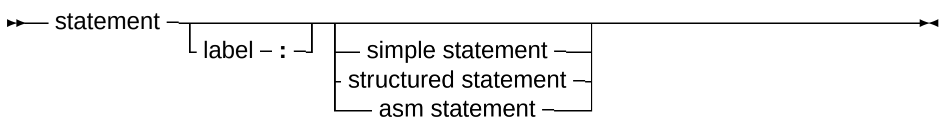
Operator	Action
`=`	Equal
`<>`	Not equal
`<`	Stricty less than
`>`	Strictly greater than
`<=`	Less than or equal
`>=`	Greater than or equal
`in`	Element of

Chapter 7

Statements

The heart of each algorithm are the actions it takes. These actions are contained in the statements of your program or unit. You can label your statements, and jump to them (within certain limits) with `Goto` statements. This can be seen in the following syntax diagram:

Statements

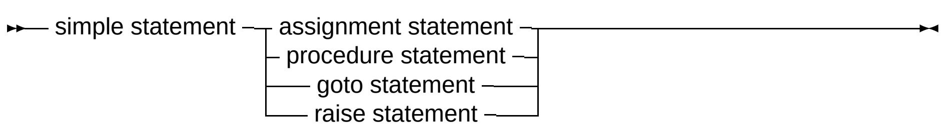


A label can be an identifier or an integer digit.

7.1 Simple statements

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

Simple statements



Of these statements, the *raise statement* will be explained in the chapter on Exceptions (chapter 11, page 98)

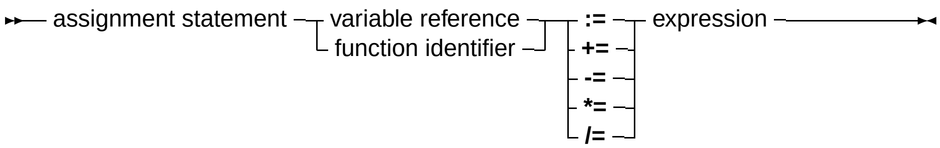
Assignments

Assignments give a value to a variable, replacing any previous value the variable might have had:

Table 7.1: Allowed C constructs in Free Pascal

Assignment	Result
a += b	Adds b to a, and stores the result in a.
a -= b	Substracts b from a, and stores the result in a.
a *= b	Multiplies a with b, and stores the result in a.
a /= b	Divides a through b, and stores the result in a.

Assignments



In addition to the standard Pascal assignment operator (`:=`), which simply replaces the value of the varable with the value resulting from the expression on the right of the := operator, Free Pascal supports some c-style constructions. All available constructs are listed in table (7.1). For these constructs to work, you should specify the `-Sc` command-line switch.

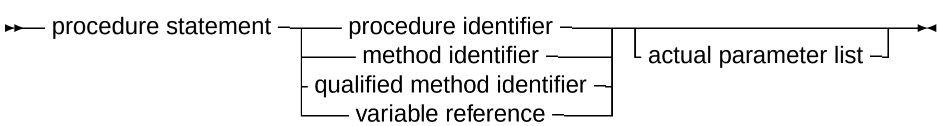
Remark: These constructions are just for typing convenience, they don't generate different code. Here are some examples of valid assignment statements:

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

Procedure statements

Procedure statements are calls to subroutines. There are different possibilities for procedure calls: A normal procedure call, an object method call (fully qualified or not), or even a call to a procedural type variable. All types are present in the following diagram.

Procedure statements



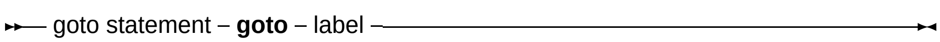
The Free Pascal compiler will look for a procedure with the same name as given in the procedure statement, and with a declared parameter list that matches the actual parameter list. The following are valid procedure statements:

```
Usage;
WriteLn('Pascal is an easy language !');
Doit();
```

Goto statements

Free Pascal supports the `goto` jump statement. Its prototype syntax is

Goto statement



When using `goto` statements, you must keep the following in mind:

1. The jump label must be defined in the same block as the `Goto` statement.
2. Jumping from outside a loop to the inside of a loop or vice versa can have strange effects.
3. To be able to use the `Goto` statement, you need to specify the `-Sg` compiler switch.

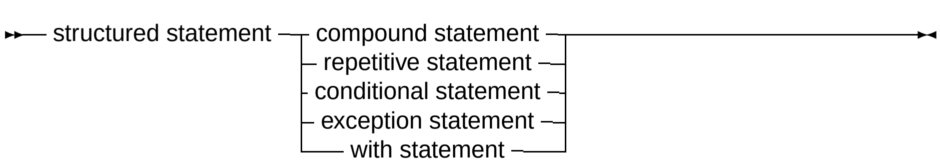
`Goto` statements are considered bad practice and should be avoided as much as possible. It is always possible to replace a `goto` statement by a construction that doesn't need a `goto`, although this construction may not be as clear as a goto statement. For instance, the following is an allowed goto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

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

7.2 Structured statements

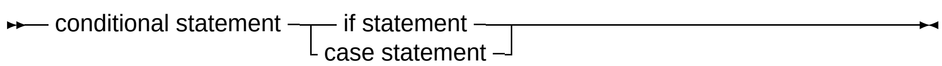
Structured statements can be broken into smaller simple statements, which should be executed repeatedly, conditionally or sequentially:

Structured statements



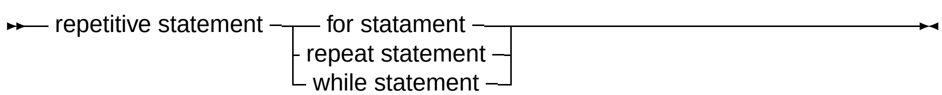
Conditional statements come in 2 flavours :

Conditional statements



Repetitive statements come in 3 flavours:

Repetitive statements

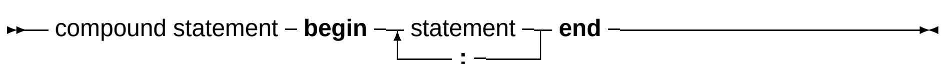


The following sections deal with each of these statements.

Compound statements

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

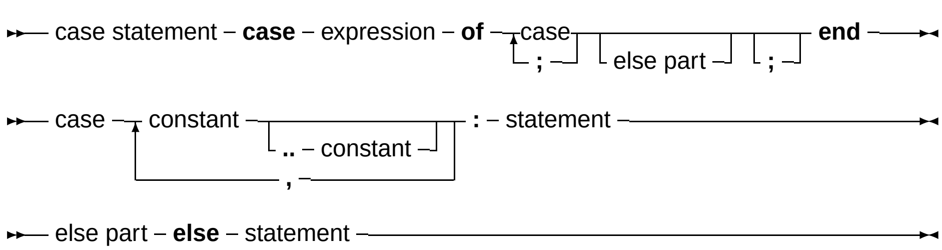
Compound statements



The `Case` statement

Free Pascal supports the `case` statement. Its syntax diagram is

Case statement



The constants appearing in the various case parts must be known at compile-time, and can be of the following types : enumeration types, Ordinal types (except boolean), and chars. The expression must be also of this type, or a compiler error will occur. All case constants must have the same type. The compiler will evaluate the expression. If one of the case constants values matches the value of the expression, the statement that follows this constant is executed. After that, the program continues after the final `end`. If none of the case constants match the expression value, the statement after the `else` keyword is executed. This can be an empty statement. If no else part is present, and no case constant matches the expression value, program flow continues after the final `end`. The case statements can be compound statements (i.e. a `begin..End` block).

Remark: Contrary to Turbo Pascal, duplicate case labels are not allowed in Free Pascal, so the following code will generate an error when compiling:

```
Var i : integer;
...
Case i of
 3 : DoSomething;
 1..5 : DoSomethingElse;
end;
```

The compiler will generate a `Duplicate case label` error when compiling this, because the 3 also appears (implicitly) in the range `1..5`. This is similar to Delhpi syntax.

The following are valid case statements:

```
Case C of
 'a' : WriteLn ('A pressed');
 'b' : WriteLn ('B pressed');
 'c' : WriteLn ('C pressed');
else
  WriteLn ('unknown letter pressed : ',C);
end;
```

Or

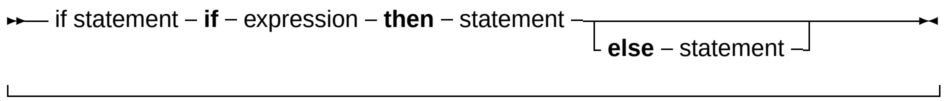
```
Case C of
 'a','e','i','o','u' : WriteLn ('vowel pressed');
 'y' : WriteLn ('This one depends on the language');
else
  WriteLn ('Consonant pressed');
end;

Case Number of
 1..10   : WriteLn ('Small number');
 11..100 : WriteLn ('Normal, medium number');
else
 WriteLn ('HUGE number');
end;
```

The `If..then..else` statement

The `If .. then .. else..` prototype syntax is

If then statements



The expression between the `if` and `then` keywords must have a boolean return type. If the expression evaluates to `True` then the statement following `then` is executed.

If the expression evaluates to `False`, then the statement following `else` is executed, if it is present.

Be aware of the fact that the boolean expression will be short-cut evaluated. (Meaning that the evaluation will be stopped at the point where the outcome is known with certainty) Also, before the `else` keyword, no semicolon (`;`) is allowed, but all statements can be compound statements. In nested `If.. then .. else` constructs, some ambiguity may araise as to which `else` statement pairs with which `if` statement. The rule is that the `else` keyword matches the first `if` keyword not already matched by an `else` keyword. For example:

```
If exp1 Then
  If exp2 then
    Stat1
else
  stat2;
```

Despite it's appearance, the statement is syntactically equivalent to

```
If exp1 Then
   begin
   If exp2 then
      Stat1
   else
      stat2
   end;
```

and not to

```
{ NOT EQUIVALENT }
If exp1 Then
   begin
   If exp2 then
      Stat1
   end
else
   stat2
```

If it is this latter construct you want, you must explicitly put the `begin` and `end` keywords. When in doubt, add them, they don't hurt.

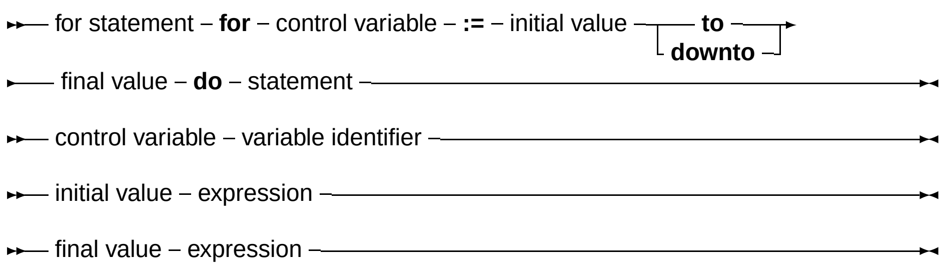
The following is a valid statement:

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

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

Free Pascal supports the `For` loop construction. A for loop is used in case one wants to calculated something a fixed number of times. The prototype syntax is as follows:

For statement



`Statement` can be a compound statement. When this statement is encountered, the control variable is initialized with the initial value, and is compared with the final value. What happens next depends on whether `to` or `downto` is used:

1. In the case `To` is used, if the initial value larger than the final value then `Statement` will never be executed.
2. In the case `DownTo` is used, if the initial value larger than the final value then `Statement` will never be executed.

After this check, the statement after `Do` is executed. After the execution of the statement, the control variable is increased or decreased with 1, depending on whether `To` or `Downto` is used. The control variable must be an ordinal type, no other types can be used as counters in a loop.

Remark: Contrary to ANSI pascal specifications, Free Pascal first initializes the counter variable, and only then calculates the upper bound.

The following are valid loops:

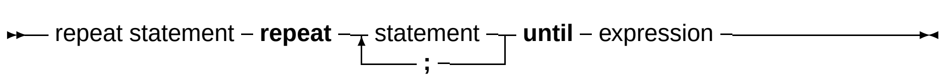
```
For Day := Monday to Friday do Work;
For I := 100 downto 1 do
  WriteLn ('Counting down : ',i);
For I := 1 to 7*dwarfs do KissDwarf(i);
```

If the statement is a compound statement, then the Break (119) and Continue (121) reserved words can be used to jump to the end or just after the end of the `For` statement.

The `Repeat..until` statement

The `repeat` statement is used to execute a statement until a certain condition is reached. The statement will be executed at least once. The prototype syntax of the `Repeat..until` statement is

Repeat statement



This will execute the statements between `repeat` and `until` up to the moment when `Expression` evaluates to `True`. Since the `expression` is evaluated *after* the execution of the statements, they are executed at least once. Be aware of the fact that the boolean expression `Expression` will be short-cut evaluated. (Meaning that the evaluation will be stopped at the point where the outcome is known with certainty) The following are valid `repeat` statements

```
repeat
  WriteLn ('I =',i);
  I := I+2;
until I>100;
repeat
 X := X/2
until x<10e-3
```

The Break (119) and Continue (121) reserved words can be used to jump to the end or just after the end of the `repeat .. until` statement.

The `While..do` statement

A `while` statement is used to execute a statement as long as a certain condition holds. This may imply that the statement is never executed. The prototype syntax of the `While..do` statement is

While statements

→→ while statement – **while** – expression – **do** – statement ——→←

This will execute `Statement` as long as `Expression` evaluates to `True`. Since `Expression` is evaluated *before* the execution of `Statement`, it is possible that `Statement` isn't executed at all. `Statement` can be a compound statement. Be aware of the fact that the boolean expression `Expression` will be short-cut evaluated. (Meaning that the evaluation will be stopped at the point where the outcome is known with certainty) The following are valid `while` statements:

```
I := I+2;
while i<=100 do
  begin
  WriteLn ('I =',i);
  I := I+2;
  end;
X := X/2;
while x>=10e-3 do
  X := X/2;
```

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

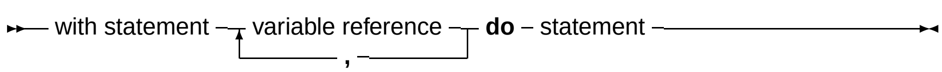
If the statement is a compound statement, then the Break (119) and Continue (121) reserved words can be used to jump to the end or just after the end of the `While` statement.

The `With` statement

The `with` statement serves to access the elements of a record[1] or object or class, without having to specify the name of the each time. The syntax for a `with` statement is

[1] The `with` statement does not work correctly when used with objects or classes until version 0.99.6

With statement



The variable reference must be a variable of a record, object or class type. In the `with` statement, any variable reference, or method reference is checked to see if it is a field or method of the record or object or class. If so, then that field is accessed, or that method is called. Given the declaration:

```
Type Passenger = Record
       Name : String[30];
       Flight : String[10];
       end;
Var TheCustomer : Passenger;
```

The following statements are completely equivalent:

```
TheCustomer.Name := 'Michael';
TheCustomer.Flight := 'PS901';
```

and

```
With TheCustomer do
  begin
  Name := 'Michael';
  Flight := 'PS901';
  end;
```

The statement

```
With A,B,C,D do Statement;
```

is equivalent to

```
With A do
 With B do
  With C do
   With D do Statement;
```

This also is a clear example of the fact that the variables are tried *last to first*, i.e., when the compiler encounters a variable reference, it will first check if it is a field or method of the last variable. If not, then it will check the last-but-one, and so on. The following example shows this;

```
Program testw;
Type AR = record
      X,Y : Longint;
     end;
     PAR = Record;

Var S,T : Ar;
begin
  S.X := 1;S.Y := 1;
```

```
  T.X := 2;T.Y := 2;
  With S,T do
    WriteLn (X,' ',Y);
end.
```

The output of this program is

```
2 2
```

Showing thus that the `X,Y` in the `WriteLn` statement match the `T` record variable.

Remark: If you use a `With` statement with a pointer, or a class, it is not permitted to change the pointer or the class in the `With` block. With the definitions of the previous example, the following illiustrates what it is about:

```
Var p : PAR;

begin
  With P^ do
   begin
   // Do some operations
   P:=OtherP;
   X:=0.0;  // Wrong X will be used !!
   end;
```

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

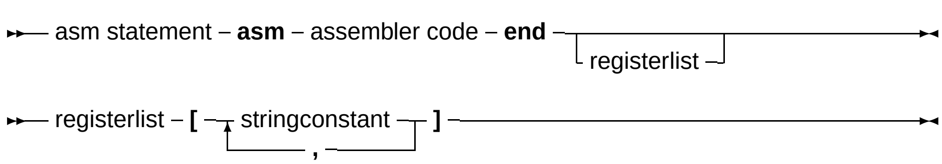
Exception Statements

As of version 0.99.7, Free Pascal supports exceptions. Exceptions provide a convenient way to program error and error-recovery mechanisms, and are closely related to classes. Exception support is explained in chapter 11, page 98

7.3 Assembler statements

An assembler statement allows you to insert assembler code right in your pascal code.

Assembler statements



More information about assembler blocks can be found in the Programmers' guide. The register list is used to indicate the registers that are modified by an assembler statement in your code. The compiler stores certain results in the registers. If you modify the registers in an assembler statement, the compiler should, sometimes, be told about it. The registers are denoted with their Intel names for the I386 processor, i.e., 'EAX', 'ESI' etc... As an example, consider the following assembler code:

```
asm
  Movl $1,%ebx
  Movl $0,%eax
  addl %eax,%ebx
end; ['EAX','EBX'];
```

This will tell the compiler that it should save and restore the contents of the EAX and EBX registers when it encounters this asm statement.

Free Pascal supports various styles of assembler syntax. By default, AT&T syntax is assumed. You can change the default assembler style with the {$asmmode xxx} switch in your code, or the -R command-line option. More about this can be found in the Programmers' guide.

Chapter 8

Using functions and procedures

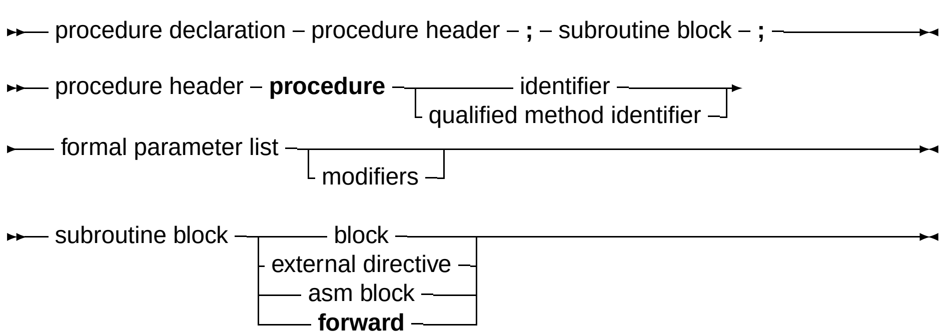
Free Pascal supports the use of functions and procedures, but with some extras: Function overloading is supported, as well as `Const` parameters and open arrays.

Remark: In many of the subsequent paragraphs the words `procedure` and `function` will be used interchangeably. The statements made are valid for both, except when indicated otherwise.

8.1 Procedure declaration

A procedure declaration defines an identifier and associates it with a block of code. The procedure can then be called with a procedure statement.

Procedure declaration



See section 8.3, page 75 for the list of parameters. A procedure declaration that is followed by a block implements the action of the procedure in that block. The following is a valid procedure :

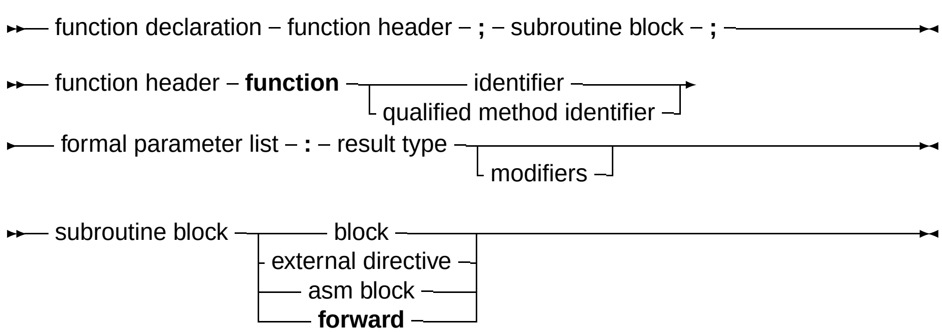
```
Procedure DoSomething (Para : String);
begin
  Writeln ('Got parameter : ',Para);
  Writeln ('Parameter in upper case : ',Upper(Para));
end;
```

Note that it is possible that a procedure calls itself.

8.2 Function declaration

A function declaration defines an identifier and associates it with a block of code. The block of code will return a result. The function can then be called inside an expression, or with a procedure statement, if extended syntax is on.

Function declaration

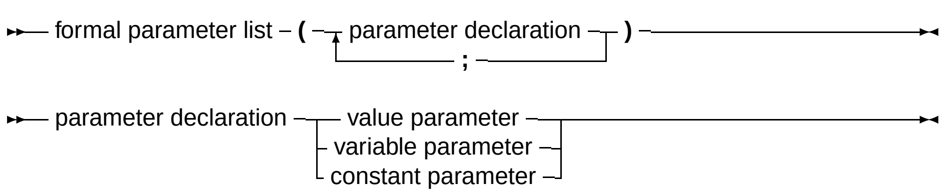


The result type of a function can be any previously declared type. contrary to Turbo pascal, where only simple types could be returned.

8.3 Parameter lists

When you need to pass arguments to a function or procedure, these parameters must be declared in the formal parameter list of that function or procedure. The parameter list is a declaration of identifiers that can be referred to only in that procedure or function's block.

Parameters

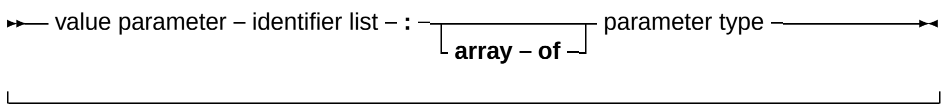


Constant parameters and variable parameters can also be `untyped` parameters if they have no type identifier.

Value parameters

Value parameters are declared as follows:

Value parameters



When you declare parameters as value parameters, the procedure gets *a copy* of the parameters that the calling block passes. Any modifications to these parameters are purely local to the procedure's block, and do not propagate back to the calling block. A block that wishes to call a procedure with value parameters must pass assignment compatible parameters to the procedure. This means that the types should not match exactly, but can be converted (conversion code is inserted by the compiler itself)

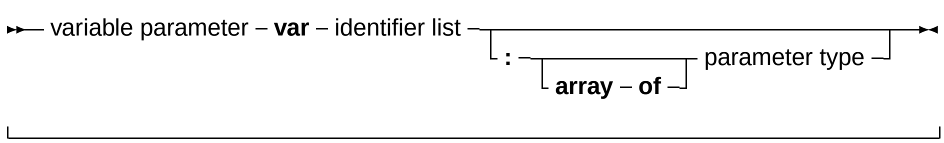
Take care that using value parameters makes heavy use of the stack, especially if you pass large parameters. The total size of all parameters in the formal parameter list should be below 32K for portability's sake (the Intel version limits this to 64K).

You can pass open arrays as value parameters. See section 8.3, page 77 for more information on using open arrays.

Variable parameters

Variable parameters are declared as follows:

Variable parameters



When you declare parameters as variable parameters, the procedure or function accesses immediatly the variable that the calling block passed in its parameter list. The procedure gets a pointer to the variable that was passed, and uses this pointer to access the variable's value. From this, it follows that any changes that you make to the parameter, will proagate back to the calling block. This mechanism can be used to pass values back in procedures. Because of this, the calling block must pass a parameter of *exactly* the same type as the declared parameter's type. If it does not, the compiler will generate an error.

Variable parameters can be untyped. In that case the variable has no type, and hence is incompatible with all other types. However, you can use the address operator on it, or you can pass it to a function that has also an untyped parameter. If you want to use an untyped parameter in an assigment, or you want to assign to it, you must use a typecast.

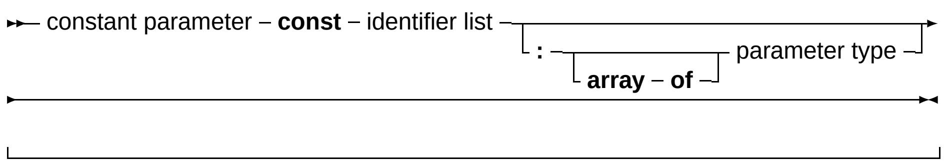
File type variables must always be passed as variable parameters.

You can pass open arrays as variable parameters. See section 8.3, page 77 for more information on using open arrays.

Constant parameters

In addition to variable parameters and value parameters Free Pascal also supports Constant parameters. You can specify a constant parameter as follows:

Constant parameters



A constant argument is passed by reference if it's size is larger than a longint. It is passed by value if the size equals 4 or less. This means that the function or procedure receives a pointer to the passed argument, but you are not allowed to assign to it, this will result in a compiler error. Likewise, you cannot pass a const parameter on to another function that requires a variable parameter. The main use for this is reducing the stack size, hence improving performance, and still retaining the semantics of passing by value...

Constant parameters can also be untyped. See section 8.3, page 76 for more information about untyped parameters.

You can pass open arrays as constant parameters. See section 8.3, page 77 for more information on using open arrays.

Open array parameters

Free Pascal supports the passing of open arrays, i.e. you can declare a procedure with an array of unspecified length as a parameter, as in Delphi. Open array parameters can be accessed in the procedure or function as an array that is declared with starting index 0, and last element index `High(paremeter)`. For example, the parameter

```
Row : Array of Integer;
```

would be equivalent to

```
Row : Array[0..N-1] of Integer;
```

Where `N` would be the actual size of the array that is passed to the function. `N-1` can be calculated as `High(Row)`. Open parameters can be passed by value, by reference or as a constant parameter. In the latter cases the procedure receives a pointer to the actual array. In the former case, it receives a copy of the array. In a function or procedure, you can pass open arrays only to functions which are also declared with open arrays as parameters, *not* to functions or procedures which accept arrays of fixed length. The following is an example of a function using an open array:

```
Function Average (Row : Array of integer) : Real;
Var I : longint;
    Temp : Real;
begin
  Temp := Row[0];
  For I := 1 to High(Row) do
    Temp := Temp + Row[i];
  Average := Temp / (High(Row)+1);
end;
```

Array of const

In Object Pascal or Delphi mode, Free Pascal supports the `Array of Const` construction to pass parameters to a subroutine.

This is a special case of the `Open array` construction, where you are allowed to pass any expression in an array to a function or procedure.

In the procedure, passed the arguments can be examined using a special record:

```
Type
   PVarRec = ^TVarRec;
   TVarRec = record
     case VType : Longint of
       vtInteger    : (VInteger: Longint);
       vtBoolean    : (VBoolean: Boolean);
       vtChar       : (VChar: Char);
       vtExtended   : (VExtended: PExtended);
       vtString     : (VString: PShortString);
       vtPointer    : (VPointer: Pointer);
       vtPChar      : (VPChar: PChar);
       vtObject     : (VObject: TObject);
       vtClass      : (VClass: TClass);
       vtAnsiString : (VAnsiString: Pointer);
       vtWideString : (VWideString: Pointer);
       vtInt64      : (VInt64: PInt64);
   end;
```

Inside the procedure body, the array of const is equivalent to an open array of TVarRec:

```
Procedure Testit (Args: Array of const);

Var I : longint;

begin
  If High(Args)<0 then
    begin
    Writeln ('No aguments');
    exit;
    end;
  Writeln ('Got ',High(Args)+1,' arguments :');
  For i:=0 to High(Args) do
    begin
    write ('Argument ',i,' has type ');
    case Args[i].vtype of
      vtinteger    :
        Writeln ('Integer, Value :',args[i].vinteger);
      vtboolean    :
        Writeln ('Boolean, Value :',args[i].vboolean);
      vtchar       :
        Writeln ('Char, value : ',args[i].vchar);
      vtextended   :
        Writeln ('Extended, value : ',args[i].VExtended^);
      vtString     :
        Writeln ('ShortString, value :',args[i].VString^);
      vtPointer    :
        Writeln ('Pointer, value : ',Longint(Args[i].VPointer));
      vtPChar      :
        Writeln ('PCHar, value : ',Args[i].VPChar);
      vtObject     :
```

```
        Writeln ('Object, name : ',Args[i].VObject.Classname);
      vtClass      :
        Writeln ('Class reference, name :',Args[i].VClass.Classname);
      vtAnsiString :
        Writeln ('AnsiString, value :',AnsiString(Args[I].VAnsiStr
    else
        Writeln ('(Unknown) : ',args[i].vtype);
    end;
    end;
end;
```

In your code, it is possible to pass an arbitrary array of elements to this procedure:

```
  S:='Ansistring 1';
  T:='AnsiString 2';
  Testit ([]);
  Testit ([1,2]);
  Testit (['A','B']);
  Testit ([TRUE,FALSE,TRUE]);
  Testit (['String','Another string']);
  Testit ([S,T])  ;
  Testit ([P1,P2]);
  Testit ([@testit,Nil]);
  Testit ([ObjA,ObjB]);
  Testit ([1.234,1.234]);
  TestIt ([AClass]);
```

If the procedure is declared with the `cdecl` modifier, then the compiler will pass the array as a C compiler would pass it. This, in effect, emulates the C construct of a varable number of arguments, as the following example will show:

```
program testaocc;
{$mode objfpc}

Const
  P : Pchar = 'example';
  Fmt : PChar =
        'This %s uses printf to print numbers (%d) and strings.'#10;

// Declaration of standard C function printf:
procedure printf (fm : pchar; args : array of const);cdecl; external 'c';

begin
 printf(Fmt,[P,123]);
end.
```

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

8.4 Function overloading

Function overloading simply means that you can define the same function more than once, but each time with a different formal parameter list. The parameter lists must differ at least in one of it's elements type. When the compiler encounters a function call, it will look at the function parameters to decide which one of the defined functions it should call. This can be useful if you want to define the same function for different types. For example, in the RTL, the `Dec` procedure is is defined as:

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

When the compiler encounters a call to the dec function, it will first search which function it should use. It therefore checks the parameters in your function call, and looks if there is a function definition which matches the specified parameter list. If the compiler finds such a function, a call is inserted to that function. If no such function is found, a compiler error is generated. You cannot have overloaded functions that have a `cdecl` or `export` modifier (Technically, because these two modifiers prevent the mangling of the function name by the compiler).

8.5 Forward defined functions

You can define a function without having it followed by it's implementation, by having it followed by the `forward` procedure. The effective implementation of that function must follow later in the module. The function can be used after a `forward` declaration as if it had been implemented already. The following is an example of a forward declaration.

```
Program testforward;
Procedure First (n : longint); forward;
Procedure Second;
begin
  WriteLn ('In second. Calling first...');
  First (1);
end;
Procedure First (n : longint);
begin
  WriteLn ('First received : ',n);
end;
begin
  Second;
end.
```

You cannot define a function twice as forward (nor is there any reason why you would want to do that). Likewise, in units, you cannot have a forward declared function of a function that has been declared in the interface part. The interface declaration counts as a `forward` declaration. The following unit will give an error when compiled:

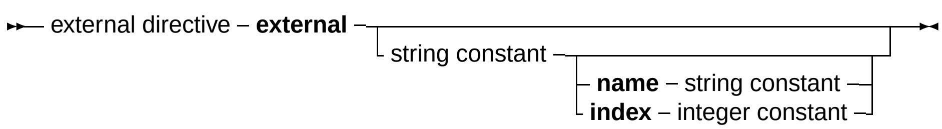
```
Unit testforward;
interface
Procedure First (n : longint);
Procedure Second;
implementation
Procedure First (n : longint); forward;
Procedure Second;
begin
  WriteLn ('In second. Calling first...');
```

```
  First (1);
end;
Procedure First (n : longint);
begin
  WriteLn ('First received : ',n);
end;
end.
```

8.6 External functions

The `external` modifier can be used to declare a function that resides in an external object file. It allows you to use the function in your code, and at linking time, you must link the object file containing the implementation of the function or procedure.

External directive



It replaces, in effect, the function or procedure code block. As such, it can be present only in an implementation block of a unit, or in a program. As an example:

```
program CmodDemo;
{$Linklib c}
Const P : PChar = 'This is fun !';
Function strlen (P : PChar) : Longint; cdecl; external;
begin
  WriteLn ('Length of (',p,') : ',strlen(p))
end.
```

Remark: The parameters in our declaration of the `external` function should match exactly the ones in the declaration in the object file.

If the `external` modifier is followed by a string constant:

```
external 'lname';
```

Then this tells the compiler that the function resides in library 'lname'. The compiler will then automatically link this library to your program.

You can also specify the name that the function has in the library:

```
external 'lname' name Fname;
```

This tells the compiler that the function resides in library 'lname', but with name 'Fname'. The compiler will then automatically link this library to your program, and use the correct name for the function. Under WINDOWS 32-BIT and OS/2, you can also use the following form:

```
external 'lname' Index Ind;
```

This tells the compiler that the function resides in library 'lname', but with index `Ind`. The compiler will then automatically link this library to your program, and use the correct index for the function.

8.7 Assembler functions

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

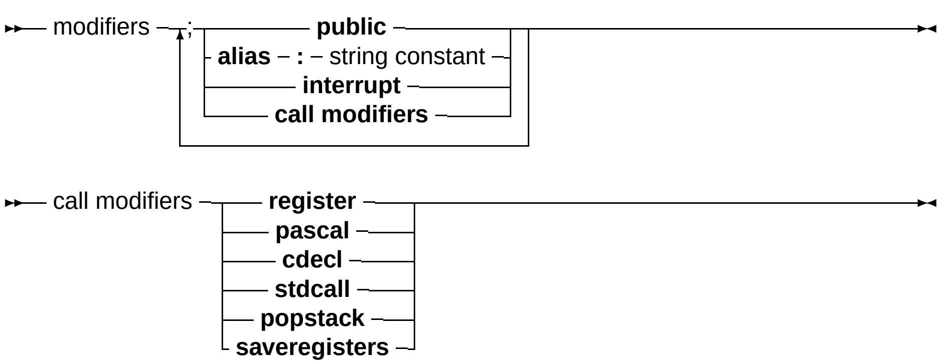
Assembler functions

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

8.8 Modifiers

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

Modifiers



Free Pascal doesn't support all Turbo Pascal modifiers, but does support a number of additional modifiers. They are used mainly for assembler and reference to C object files. More on the use of modifiers can be found in the Programmers' guide.

Public

The `Public` keyword is used to declare a function globally in a unit. This is useful if you don't want a function to be accessible from the unit file, but you do want the function to be accessible from the object file. as an example:

```
Unit someunit;
interface
Function First : Real;
Implementation
Function First : Real;
begin
```

```
  First := 0;
end;
Function Second : Real; [Public];
begin
  Second := 1;
end;
end.
```

If another program or unit uses this unit, it will not be able to use the function `Second`, since it isn't declared in the interface part. However, it will be possible to access the function `Second` at the assembly-language level, by using it's mangled name (see the Programmers' guide).

cdecl

The `cdecl` modifier can be used to declare a function that uses a C type calling convention. This must be used if you wish to acces functions in an object file generated by a C compiler. It allows you to use the function in your code, and at linking time, you must link the object file containing the `C` implementation of the function or procedure. As an example:

```
program CmodDemo;
{$LINKLIB c}
Const P : PChar = 'This is fun !';
Function strlen (P : PChar) : Longint; cdecl; external;
begin
  WriteLn ('Length of (',p,') : ',strlen(p))
end.
```

When compiling this, and linking to the C-library, you will be able to call the `strlen` function throughout your program. The `external` directive tells the compiler that the function resides in an external object filebrary (see 8.6).

Remark: The parameters in our declaration of the `C` function should match exactly the ones in the declaration in `C`. Since `C` is case sensitive, this means also that the name of the function must be exactly the same. the Free Pascal compiler will use the name *exactly* as it is typed in the declaration.

popstack

Popstack does the same as `cdecl`, namely it tells the Free Pascal compiler that a function uses the C calling convention. In difference with the `cdecl` modifier, it still mangles the name of the function as it would for a normal pascal function. With `popstack` you could access functions by their pascal names in a library.

Export

Sometimes you must provide a callback function for a C library, or you want your routines to be callable from a C program. Since Free Pascal and C use different calling schemes for functions and procedures[1], the compiler must be told to generate code that can be called from a C routine. This is where the `Export` modifier comes in. Contrary to the other modifiers, it must be specified separately, as follows:

```
function DoSquare (X : Longint) : Longint; export;
```

[1]More techically: In C the calling procedure must clear the stack. In Free Pascal, the subroutine clears the stack.

```
begin
...
end;
```

The square brackets around the modifier are not allowed in this case.

Remark: as of version 0.9.8, Free Pascal supports the Delphi `cdecl` modifier. This modifier works in the same way as the `export` modifier. More information about these modifiers can be found in the Programmers' guide, in the section on the calling mechanism and the chapter on linking.

StdCall

As of version 0.9.8, Free Pascal supports the Delphi `stdcall` modifier. This modifier does actually nothing, since the Free Pascal compiler by default pushes parameters from right to left on the stack, which is what the modifier does under Delphi (which pushes parameters on the stack from left to right). More information about this modifier can be found in the Programmers' guide, in the section on the calling mechanism and the chapter on linking.

saveregisters

As of version 0.99.15, Free Pascal has the `saveregisters` modifier. If this modifier is specified after a procedure or function, then the Free Pascal compiler will save all registers on procedure entry, and restore them when the procedure exits (except for registers where return values are stored).

You should not need this modifier, except maybe when calling assembler code.

Alias

The `Alias` modifier allows you to specify a different name for a procedure or function. This is mostly useful for referring to this procedure from assembly language constructs. As an example, consider the following program:

```
Program Aliases;
Procedure Printit; [Alias : ’DOIT’];
begin
  WriteLn (’In Printit (alias : "DOIT")’);
end;
begin
  asm
  call DOIT
  end;
end.
```

Remark: the specified alias is inserted straight into the assembly code, thus it is case sensitive.

The `Alias` modifier, combined with the `Public` modifier, make a powerful tool for making externally accessible object files.

8.9 Unsupported Turbo Pascal modifiers

The modifiers that exist in Turbo pascal, but aren't supported by Free Pascal, are listed in table (8.1).

Table 8.1: Unsupported modifiers

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

Chapter 9

Operator overloading

9.1 Introduction

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

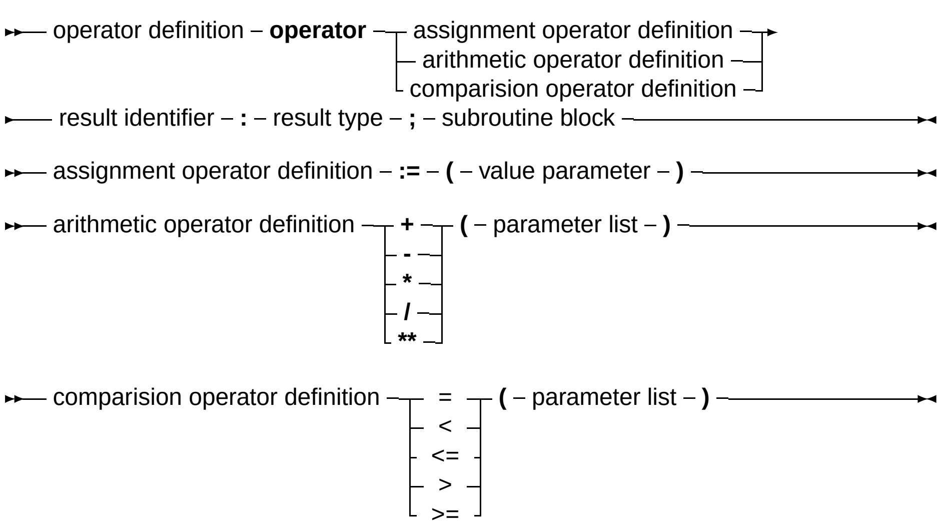
Defining the action of an operator is much like the definition of a function or procedure, only there are some restrictions on the possible definitions, as will be shown in the subsequent.

Operator overloading is, in essence, a powerful notational tool; but it is also not more than that, since the same results can be obtained with regular function calls. When using operator overloading, It is important to keep in mind that some implicit rules may produce some unexpected results. This will be indicated.

9.2 Operator declarations

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

Operator definitions



The parameter list for a comparision operator or an arithmetic operator must always contain 2 parameters. The result type of the comparision operator must be `Boolean`.

The statement block contains the necessary statements to determine the result of the operation. It can contain arbitrary large pieces of code; it is executed whenever the operation is encountered in some expression. The result of the statement block must always be defined; error conditions are not checked bythe compiler, and the code must take care of all possible cases, throwing a run-time error if some error condition is encountered.

In the following, the three types of operator definitions will be examined. As an example, throughout this chapter the following type will be used to define overloaded operators on :

```
type
  complex = record
    re : real;
    im : real;
  end;
```

this type will be used in all examples.

The sources of the Run-Time Library contain a unit ucomplex, which contains a complete calculus for complex numbers, based on operator overloading.

9.3 Assignment operators

The assignment operator defines the action of a assignent of one type of variable to another. The result type must match the type of the variable at the left of the assignment statement, the single parameter to the assignment operator must have the same type as the expression at the right of the assignment operator.

This system can be used to declare a new type, and define an assignment for that type. For instance, to be able to assign a newly defined type 'Complex'

```
Var
  C,Z : Complex; // New type complex

begin
  Z:=C;  // assignments between complex types.
end;
```

You would have to define the following assignment operator:

```
Operator := (C : Complex) z : complex;
```

To be able to assign a real type to a complex type as follows:

```
var
  R : real;
  C : complex;

begin
  C:=R;
end;
```

the following assignment operator must be defined:

```
Operator := (r : real) z : complex;
```

As can be seen from this statement, it defines the action of the operator `:=` with at the right a real expression, and at the left a complex expression.

an example implementation of this could be as follows:

```
operator := (r : real) z : complex;

begin
  z.re:=r;
  z.im:=0.0;
end;
```

As can be seen in the example, the result identifier (`z` in this case) is used to store the result of the assignment. When compiling in Delphi mode or objfpc mode, the use of the special identifier `Result` is also allowed, and can be substituted for the `z`, so the above would be equivalent to

```
operator := (r : real) z : complex;

begin
  Result.re:=r;
  Result.im:=0.0;
end;
```

The assignment operator is also used to convert types from one type to another. The compiler will consider all overloaded assignment operators till it finds one that matches the types of the left hand and right hand expressions. If no such operator is found, a 'type mismatch' error is given.

Remark: The assignment operator is not commutative; the compiler will never reverse the role of the two arguments. in other words, given the above definition of the assignment operator, the following is *not* possible:

```
var
  R : real;
  C : complex;

begin
  R:=C;
end;
```

if the reverse assignment should be possible (this is not so for reals and complex numbers) then the assigment operator must be defined for that as well.

Remark: The assignment operator is also used in implicit type conversions. This can have unwanted effects. Consider the following definitions:

```
operator := (r : real) z : complex;
function exp(c : complex) : complex;
```

then the following assignment will give a type mismatch:

```
Var
  r1,r2 : real;

begin
  r1:=exp(r2);
end;
```

because the compiler will encounter the definition of the `exp` function with the complex argument. It implicitly converts r2 to a complex, so it can use the above `exp` function. The result of this function is a complex, which cannot be assigned to r1, so the compiler will give a 'type mismatch' error. The compiler will not look further for another `exp` which has the correct arguments.

It is possible to avoid this particular problem by specifying

```
  r1:=system.exp(r2);
```

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

9.4 Arithmetic operators

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

multiplication to multiply two types, the `*` multiplication operator must be overloaded.

division to divide two types, the `/` division operator must be overloaded.

addition to add two types, the `+` addition operator must be overloaded.

substraction to substract two types, the `-` substraction operator must be overloaded.

exponentiation to exponentiate two types, the `**` exponentiation operator must be overloaded.

The definition of an arithmetic operator takes two parameters. The first parameter must be of the type that occurs at the left of the operator, the second parameter must be of the type that is at the right of the arithmetic operator. The result type must match the type that results after the arithmetic operation.

To compile an expression as

```
var
  R : real;
  C,Z : complex;

begin
  C:=R*Z;
end;
```

one needs a definition of the multiplication operator as:

```
Operator * (r : real; z1 : complex) z : complex;

begin
  z.re := z1.re * r;
  z.im := z1.im * r;
end;
```

As can be seen, the first operator is a real, and the second is a complex. The result type is complex.

Multiplication and addition of reals and complexes are commutative operations. The compiler, however, has no notion of this fact so even if a multiplication between a real and a complex is defined, the compiler will not use that definition when it encounters a complex and a real (in that order). It is necessary to define both operations.

So, given the above definition of the multiplication, the compiler will not accept the following statement:

```
var
  R : real;
  C,Z : complex;

begin
  C:=Z*R;
end;
```

since the types of `Z` and `R` don't match the types in the operator definition.

The reason for this behaviour is that it is possible that a multiplication is not always commutative. e.g. the multiplication of a `(n,m)` with a `(m,n)` matrix will result in a `(n,n)` matrix, while the mutiplication of a `(m,n)` with a `(n,m)` matrix is a `(m,m)` matrix, which needn't be the same in all cases.

9.5 Comparision operator

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

The comparision operators that can be overloaded are:

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

less than (<) to determine if one variable is less than another.

greater than (>) to determine if one variable is greater than another.

greater than or equal to (>=) to determine if one variable is greater than or equal to another.

less than or equal to (<=) to determine if one variable is greater than or equal to another.

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

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

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

the above definition allows comparisions of the following form:

```
Var
  C1,C2 : Complex;

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

The comparision operator definition needs 2 parameters, with the types that the operator is meant to compare. Here also, the compiler doesn't apply commutativity; if the two types are different, then it necessary to define 2 comparision operators.

In the case of complex numbers, it is, for instance necessary to define 2 comparsions: one with the complex type first, and one with the real type first.

Given the definitions

```
operator = (z1 : complex;r : real) b : boolean;
operator = (r : real; z1 : complex) b : boolean;
```

the following two comparisions are possible:

```
Var
  R,S : Real;
  C : Complex;

begin
  If (C=R) or (S=C) then
   Writeln ('Ok');
end;
```

Note that the order of the real and complex type in the two comparisions is reversed.

Chapter 10

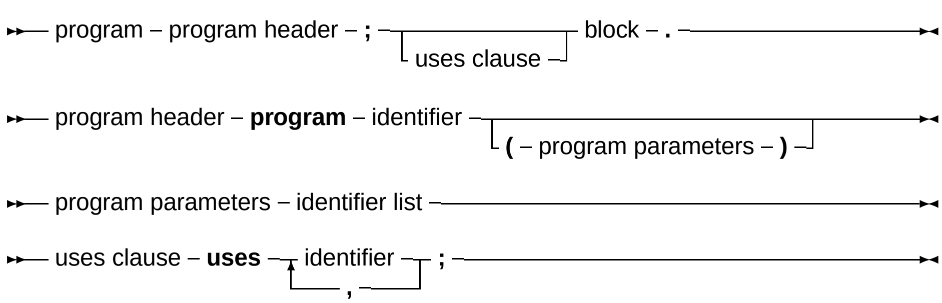
Programs, units, blocks

A Pascal program consists of modules called `units`. A unit can be used to group pieces of code together, or to give someone code without giving the sources. Both programs and units consist of code blocks, which are mixtures of statements, procedures, and variable or type declarations.

10.1 Programs

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

Programs



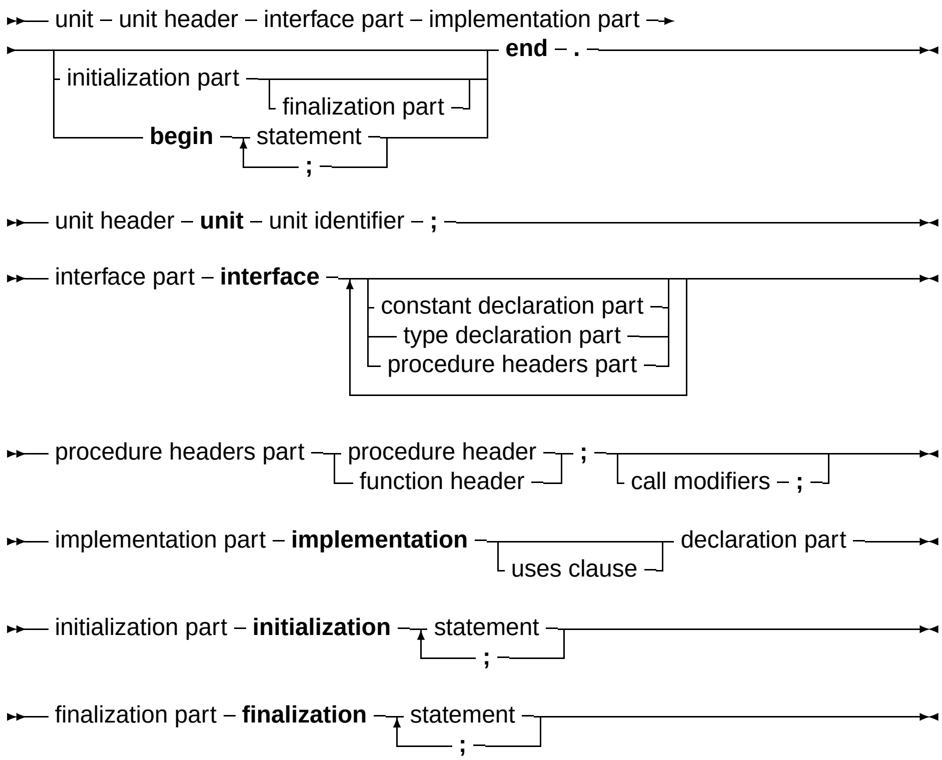
The program header is provided for backwards compatibility, and is ignored by the compiler. The uses clause serves to identify all units that are needed by the program. The system unit doesn't have to be in this list, since it is always loaded by the compiler. The order in which the units appear is significant, it determines in which order they are initialized. Units are initialized in the same order as they appear in the uses clause. Identifiers are searched in the opposite order, i.e. when the compiler searches for an identifier, then it looks first in the last unit in the uses clause, then the last but one, and so on. This is important in case two units declare different types with the same identifier. When the compiler looks for unit files, it adds the extension .ppu (.ppw for Win32 platforms) to the name of the unit. On LINUX, unit names are converted to all lowercase when looking for a unit.

If a unit name is longer than 8 characters, the compiler will first look for a unit name with this length, and then it will truncate the name to 8 characters and look for it again. For compatibility reasons, this is also true on platforms that suport long file names.

10.2 Units

A unit contains a set of declarations, procedures and functions that can be used by a program or another unit. The syntax for a unit is as follows:

Units



The interface part declares all identifiers that must be exported from the unit. This can be constant, type or variable identifiers, and also procedure or function identifier declarations. Declarations inside the implementation part are *not* accessible outside the unit. The implementation must contain a function declaration for each function or procedure that is declared in the interface part. If a function is declared in the interface part, but no declaration of that function is present in the implementation part, then the compiler will give an error.

When a program uses a unit (say unitA) and this units uses a second unit, say unitB, then the program depends indirectly also on `unitB`. This means that the compiler must have access to unitB when trying to compile the program. If the unit is not present at compile time, an error occurs.

Note that the identifiers from a unit on which a program depends indirectly, are not accessible to the program. To have access to the identifiers of a unit, you must put that unit in the uses clause of the program or unit where you want to yuse the identifier.

Units can be mutually dependent, that is, they can reference each other in their uses clauses. This is allowed, on the condition that at least one of the references is in the implementation section of the unit. This also holds for indirect mutually dependent units.

If it is possible to start from one interface uses clause of a unit, and to return there via uses clauses of interfaces only, then there is circular unit dependence, and the compiler will generate an error. As and example : the following is not allowed:

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

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

But this is allowed :

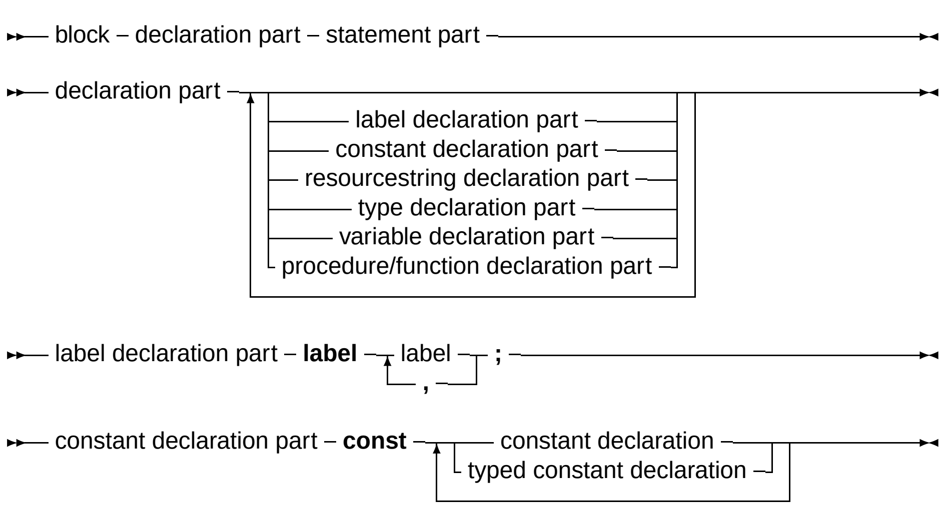
```
Unit UnitA;
interface
Uses UnitB;
implementation
end.
Unit UnitB
implementation
Uses UnitA;
end.
```

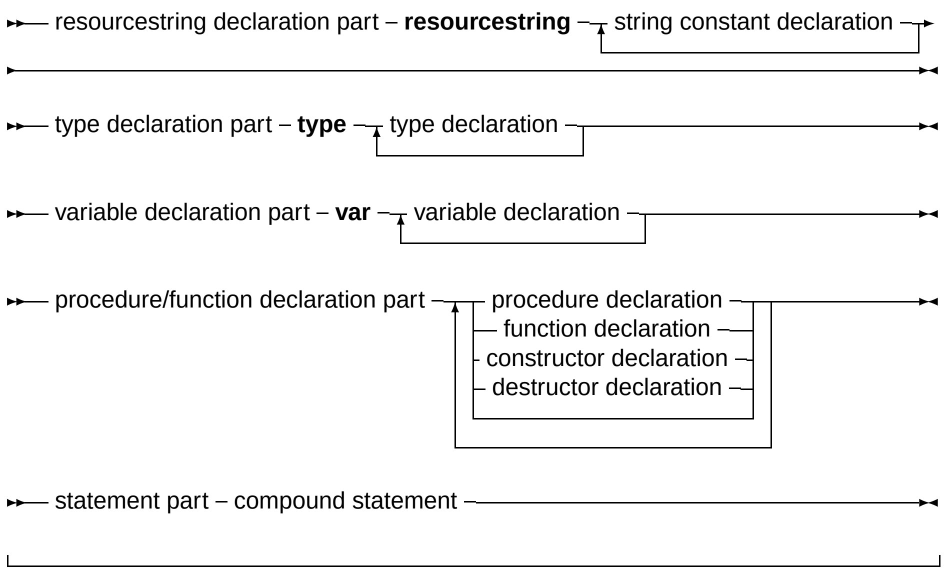
Because UnitB uses UnitA only in it's implentation section. In general, it is a bad idea to have circular unit dependencies, even if it is only in implementation sections.

10.3 Blocks

Units and programs are made of blocks. A block is made of declarations of labels, constants, types variables and functions or procedures. Blocks can be nested in certain ways, i.e., a procedure or function declaration can have blocks in themselves. A block looks like the following:

Blocks





Labels that can be used to identify statements in a block are declared in the label declaration part of that block. Each label can only identify one statement. Constants that are to be used only in one block should be declared in that block's constant declaration part. Variables that are to be used only in one block should be declared in that block's constant declaration part. Types that are to be used only in one block should be declared in that block's constant declaration part. Lastly, functions and procedures that will be used in that block can be declared in the procedure/function declaration part. After the different declaration parts comes the statement part. This contains any actions that the block should execute. All identifiers declared before the statement part can be used in that statement part.

10.4 Scope

Identifiers are valid from the point of their declaration until the end of the block in which the declaration occurred. The range where the identifier is known is the *scope* of the identifier. The exact scope of an identifier depends on the way it was defined.

Block scope

The *scope* of a variable declared in the declaration part of a block, is valid from the point of declaration until the end of the block. If a block contains a second block, in which the identfier is redeclared, then inside this block, the second declaration will be valid. Upon leaving the inner block, the first declaration is valid again. Consider the following example:

```
Program Demo;
Var X : Real;
{ X is real variable }
Procedure NewDeclaration
Var X : Integer;  { Redeclare X as integer}
begin
 // X := 1.234; {would give an error when trying to compile}
 X := 10; { Correct assigment}
end;
{ From here on, X is Real again}
begin
```

```
 X := 2.468;
end.
```

In this example, inside the procedure, X denotes an integer variable. It has it's own storage space, independent of the variable X outside the procedure.

Record scope

The field identifiers inside a record definition are valid in the following places:

1. to the end of the record definition.
2. field designators of a variable of the given record type.
3. identifiers inside a With statement that operates on a variable of the given record type.

Class scope

A component identifier is valid in the following places:

1. From the point of declaration to the end of the class definition.
2. In all descendent types of this class, unless it is in the private part of the class declaration.
3. In all method declaration blocks of this class and descendent classes.
4. In a with statement that operators on a variable of the given class's definition.

Note that method designators are also considered identifiers.

Unit scope

All identifiers in the interface part of a unit are valid from the point of declaration, until the end of the unit. Furthermore, the identifiers are known in programs or units that have the unit in their uses clause. Identifiers from indirectly dependent units are *not* available. Identifiers declared in the implementation part of a unit are valid from the point of declaration to the end of the unit. The system unit is automatically used in all units and programs. It's identifiers are therefore always known, in each program or unit you make. The rules of unit scope implie that you can redefine an identifier of a unit. To have access to an identifier of another unit that was redeclared in the current unit, precede it with that other units name, as in the following example:

```
unit unitA;
interface
Type
  MyType = Real;
implementation
end.
Program prog;
Uses UnitA;

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

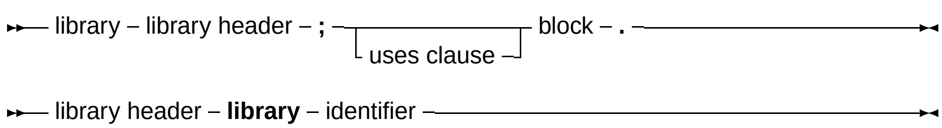
This is especially useful if you redeclare the system unit's identifiers.

10.5 Libraries

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

A Library is just like a unit or a program:

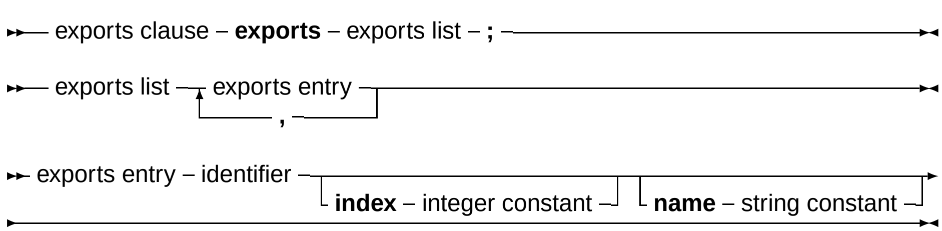
Libraries



By default, functions and procedures that are declared and implemented in library are not available to a programmer that wishes to use your library.

In order to make functions or procedures available from the library, you must export them in an export clause:

Exports clause



Under Win32, an index clause can be added to an exports entry. an index entry must be a positive number larger or equal than 1. It is best to use low index values, although nothing forces you to do this.

Optionally, an exports entry can have a name specifier. If present, the name specifier gives the exact name (case sensitive) of the function in the library.

If neither of these constructs is present, the functions or procedures are exported with the exact names as specified in the exports clause.

Chapter 11

Exceptions

As of version 0.99.7, Free Pascal supports exceptions. Exceptions provide a convenient way to program error and error-recovery mechanisms, and are closely related to classes. Exception support is based on 3 constructs:

Raise statements. To raise an exeption. This is usually done to signal an error condition.

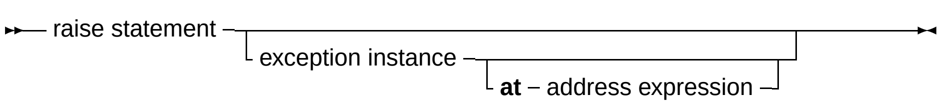
Try ... Except blocks. These block serve to catch exceptions raised within the scope of the block, and to provide exception-recovery code.

Try ... Finally blocks. These block serve to force code to be executed irrespective of an exception occurrence or not. They generally serve to clean up memory or close files in case an exception occurs. The compiler generates many implicit `Try ... Finally` blocks around procedure, to force memory consistence.

11.1 The raise statement

The `raise` statement is as follows:

Raise statement



This statement will raise an exception. If it is specified, the exception instance must be an initialized instance of a class, which is the raise type. The address exception is optional. If itis not specified, the compiler will provide the address by itself. If the exception instance is omitted, then the current exception is re-raised. This construct can only be used in an exception handling block (see further).

Remark: Control *never* returns after an exception block. The control is transferred to the first `try...finally` or `try...except` statement that is encountered when unwinding the stack. If no such statement is found, the Free Pascal Run-Time Library will generate a run-time error 217 (see also section 11.5, page 101).

As an example: The following division checks whether the denominator is zero, and if so, raises an exception of type `EDivException`

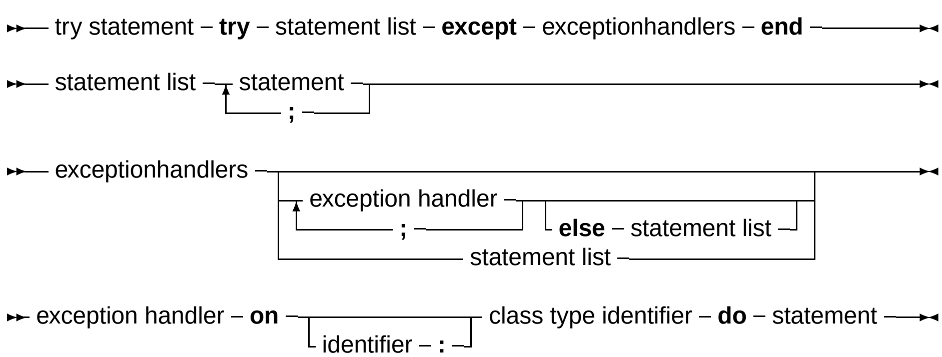
```
Type EDivException = Class(Exception);
Function DoDiv (X,Y : Longint) : Integer;
begin
  If Y=0 then
    Raise EDivException.Create ('Division by Zero would occur');
  Result := X Div Y;
end;
```

The class `Exception` is defined in the Sysutils unit of the rtl. (section 11.5, page 101)

11.2 The try...except statement

A `try...except` exception handling block is of the following form :

Try..except statement



If no exception is raised during the execution of the `statement list`, then all statements in the list will be executed sequentially, and the except block will be skipped, transferring program flow to the statement after the final `end`.

If an exception occurs during the execution of the `statement list`, the program flow will be transferred to the except block. Statements in the statement list between the place where the exception was raised and the exception block are ignored.

In the exception handling block, the type of the exception is checked, and if there is an exception handler where the class type matches the exception object type, or is a parent type of the exception object type, then the statement following the corresponding `Do` will be executed. The first matching type is used. After the `Do` block was executed, the program continues after the `End` statement.

The identifier in an exception handling statement is optional, and declares an exception object. It can be used to manipulate the exception object in the exception handling code. The scope of this declaration is the statement block foillowing the `Do` keyword.

If none of the `On` handlers matches the exception object type, then the statement list after `else` is executed. If no such list is found, then the exception is automatically re-raised. This process allows to nest `try...except` blocks.

If, on the other hand, the exception was caught, then the exception object is destroyed at the end of the exception handling block, before program flow continues. The exception is destroyed through a call to the object's `Destroy` destructor.

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

```
Try
  Z := DoDiv (X,Y);
Except
  On EDivException do Z := 0;
end;
```

If `Y` happens to be zero, then the DoDiv function code will raise an exception. When this happens, program flow is transferred to the except statement, where the Exception handler will set the value of `Z` to zero. If no exception is raised, then program flow continues past the last `end` statement. To allow error recovery, the `Try ... Finally` block is supported. A `Try...Finally` block ensures that the statements following the `Finally` keyword are guaranteed to be executed, even if an exception occurs.

11.3 The try...finally statement

A `Try..Finally` statement has the following form:

Try...finally statement

trystatement – **try** – statement list – **finally** – finally statements – **end**

finally statements – statementlist

If no exception occurs inside the `statement List`, then the program runs as if the `Try`, `Finally` and `End` keywords were not present.

If, however, an exception occurs, the program flow is immediatly transferred from the point where the excepion was raised to the first statement of the `Finally statements`.

All statements after the finally keyword will be executed, and then the exception will be automatically re-raised. Any statements between the place where the exception was raised and the first statement of the `Finally Statements` are skipped.

As an example consider the following routine:

```
Procedure Doit (Name : string);
Var F : Text;
begin
  Try
    Assign (F,Name);
    Rewrite (name);
    ... File handling ...
  Finally
    Close(F);
  end;
```

If during the execution of the file handling an execption occurs, then program flow will continue at the `close(F)` statement, skipping any file operations that might follow between the place where the exception was raised, and the `Close` statement. If no exception occurred, all file operations will be executed, and the file will be closed at the end.

11.4 Exception handling nesting

It is possible to nest `Try...Except` blocks with `Try...Finally` blocks. Program flow will be done according to a `lifo` (last in, first out) principle: The code of the last encountered `Try...Except` or `Try...Finally` block will be executed first. If the exception is not caught, or it was a finally statement, program flow will be transferred to the last-but-one block, *ad infinitum*.

If an exception occurs, and there is no exception handler present, then a runerror 217 will be generated. If you use the sysutils unit, a default handler is installed which will show the exception object message, and the address where the exception occurred, after which the program will exit with a `Halt` instruction.

11.5 Exception classes

The sysutils unit contains a great deal of exception handling. It defines the following exception types:

```
Exception = class(TObject)
 private
   fmessage : string;
   fhelpcontext : longint;
 public
   constructor create(const msg : string);
   constructor createres(indent : longint);
   property helpcontext : longint read fhelpcontext write fhelpcontext;
   property message : string read fmessage write fmessage;
end;
ExceptClass = Class of Exception;
{ mathematical exceptions }
EIntError = class(Exception);
EDivByZero = class(EIntError);
ERangeError = class(EIntError);
EIntOverflow = class(EIntError);
EMathError = class(Exception);
```

The sysutils unit also installs an exception handler. If an exception is unhandled by any exception handling block, this handler is called by the Run-Time library. Basically, it prints the exception address, and it prints the message of the Exception object, and exits with a exit code of 217. If the exception object is not a descendent object of the `Exception` object, then the class name is printed instead of the exception message.

It is recommended to use the `Exception` object or a descendant class for all `raise` statements, since then you can use the message field of the exception object.

Chapter 12

Using assembler

Free Pascal supports the use of assembler in your code, but not inline assembler macros. To have more information on the processor specific assembler syntax and its limitations, see the Programmers' guide.

12.1 Assembler statements

The following is an example of assembler inclusion in your code.

```
...
Statements;
...
Asm
  your asm code here
  ...
end;
...
Statements;
```

The assembler instructions between the `Asm` and `end` keywords will be inserted in the assembler generated by the compiler. You can still use conditionals in your assembler, the compiler will recognise it, and treat it as any other conditionals.

Remark: Before version 0.99.1, Free Pascal did not support reference to variables by their names in the assembler parts of your code.

12.2 Assembler procedures and functions

Assembler procedures and functions are declared using the `Assembler` directive. The `Assembler` keyword is supported as of version 0.9.7. 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.

Remark: From version 0.99.1 to 0.99.5 (*excluding* FPC 0.99.5a), the `Assembler` directive did not have the same effect as in Turbo Pascal, so beware! The stack frame would be omitted if there were no local variables, in this case if the assembly routine had any parameters, they would be referenced directly via the stack pointer. This was *NOT* like Turbo Pascal where the stack frame is only omitted if there are no parameters *and* no local variables. As stated earlier, starting from version 0.99.5a, Free Pascal now has the same behaviour as Turbo Pascal.

Part II

Reference : The System unit

Chapter 13

The system unit

The system unit contains the standard supported functions of Free Pascal. It is the same for all platforms. Basically it is the same as the system unit provided with Borland or Turbo Pascal.

Functions are listed in alphabetical order. Arguments of functions or procedures that are optional are put between square brackets.

The pre-defined constants and variables are listed in the first section. The second section contains an overview of all functions, grouped by functionality, and the last section contains the supported functions and procedures.

13.1 Types, Constants and Variables

Types

The following integer types are defined in the System unit:

```
Shortint = -128..127;
SmallInt = -32768..32767;
Longint  = $80000000..$7fffffff;
byte     = 0..255;
word     = 0..65535;
dword    = cardinal;
longword = cardinal;
Integer  = smallint;
```

The following types are used for the functions that need compiler magic such as Val (165) or Str (161):

```
StrLenInt = LongInt;
ValSInt = Longint;
ValUInt = Cardinal;
ValReal = Extended;
```

The following character types are defined for Delphi compatibility:

```
TAnsiChar   = Char;
AnsiChar    = TAnsiChar;
```

And the following pointer types:

```
  PChar = ^char;
  pPChar = ^PChar;
  PAnsiChar   = PChar;
  PQWord      = ^QWord;
  PInt64      = ^Int64;
  pshortstring = ^shortstring;
  plongstring  = ^longstring;
  pansistring  = ^ansistring;
  pwidestring  = ^widestring;
  pextended    = ^extended;
  ppointer     = ^pointer;
```

For the SetJmp (157) and LongJmp (141) calls, the following jump bufer type is defined (for the I386 processor):

```
  jmp_buf = record
    ebx,esi,edi : Longint;
    bp,sp,pc : Pointer;
    end;
  PJmp_buf = ^jmp_buf;
```

The following records and pointers can be used if you want to scan the entries in the string message handler tables:

```
  tmsgstrtable = record
     name : pshortstring;
     method : pointer;
  end;
  pmsgstrtable = ^tmsgstrtable;

  tstringmessagetable = record
     count : dword;
     msgstrtable : array[0..0] of tmsgstrtable;
  end;
  pstringmessagetable = ^tstringmessagetable;
```

The base class for all classes is defined as:

```
Type
  TObject = Class
  Public
    constructor create;
    destructor destroy;virtual;
    class function newinstance : tobject;virtual;
    procedure freeinstance;virtual;
    function safecallexception(exceptobject : tobject;
      exceptaddr : pointer) : longint;virtual;
    procedure defaulthandler(var message);virtual;
    procedure free;
    class function initinstance(instance : pointer) : tobject;
    procedure cleanupinstance;
    function classtype : tclass;
    class function classinfo : pointer;
    class function classname : shortstring;
```

```
    class function classnameis(const name : string) : boolean;
    class function classparent : tclass;
    class function instancesize : longint;
    class function inheritsfrom(aclass : tclass) : boolean;
    class function inheritsfrom(aclass : tclass) : boolean;
    class function stringmessagetable : pstringmessagetable;
    procedure dispatch(var message);
    procedure dispatchstr(var message);
    class function methodaddress(const name : shortstring) : pointer;
    class function methodname(address : pointer) : shortstring;
    function fieldaddress(const name : shortstring) : pointer;
    procedure AfterConstruction;virtual;
    procedure BeforeDestruction;virtual;
    procedure DefaultHandlerStr(var message);virtual;
  end;
  TClass = Class Of TObject;
  PClass = ^TClass;
```

Unhandled exceptions can be treated using a constant of the `TExceptProc` type:

```
TExceptProc = Procedure (Obj : TObject; Addr,Frame: Pointer);
```

`Obj` is the exception object that was used to raise the exception, `Addr` and `Frame` contain the exact address and stack frame where the exception was raised.

The `TVarRec` type is used to access the elements passed in a `Array of Const` argument to a function or procedure:

```
Type
  PVarRec = ^TVarRec;
  TVarRec = record
    case VType : Longint of
    vtInteger    : (VInteger: Longint);
    vtBoolean    : (VBoolean: Boolean);
    vtChar       : (VChar: Char);
    vtExtended   : (VExtended: PExtended);
    vtString     : (VString: PShortString);
    vtPointer    : (VPointer: Pointer);
    vtPChar      : (VPChar: PChar);
    vtObject     : (VObject: TObject);
    vtClass      : (VClass: TClass);
    vtAnsiString : (VAnsiString: Pointer);
    vtWideString : (VWideString: Pointer);
    vtInt64      : (VInt64: PInt64);
  end;
```

The heap manager uses the `TMemoryManager` type:

```
  PMemoryManager = ^TMemoryManager;
  TMemoryManager = record
    Getmem      : Function(Size:Longint):Pointer;
    Freemem     : Function(var p:pointer):Longint;
    FreememSize : Function(var p:pointer;Size:Longint):Longint;
    AllocMem    : Function(Size:longint):Pointer;
    ReAllocMem  : Function(var p:pointer;Size:longint):Pointer;
```

```
    MemSize     : function(p:pointer):Longint;
    MemAvail    : Function:Longint;
    MaxAvail    : Function:Longint;
    HeapSize    : Function:Longint;
  end;
```

More information on using this record can be found in Programmers' guide.

Constants

The following constants define the maximum values that can be used with various types:

```
  MaxSIntValue = High(ValSInt);
  MaxUIntValue = High(ValUInt);
  maxint   = maxsmallint;
  maxLongint  = $7fffffff;
  maxSmallint = 32767;
```

The following constants for file-handling are defined in the system unit:

```
Const
  fmclosed = $D7B0;
  fminput  = $D7B1;
  fmoutput = $D7B2;
  fminout  = $D7B3;
  fmappend = $D7B4;
  filemode : byte = 2;
```

Further, the following non processor specific general-purpose constants are also defined:

```
const
  erroraddr : pointer = nil;
  errorcode : word = 0;
 { max level in dumping on error }
  max_frame_dump : word = 20;
```

Remark: Processor specific global constants are named Testxxxx where xxxx represents the processor number (such as Test8086, Test68000), and are used to determine on what generation of processor the program is running on.

The following constants are defined to access VMT entries:

```
   vmtInstanceSize         = 0;
   vmtParent               = 8;
   vmtClassName            = 12;
   vmtDynamicTable         = 16;
   vmtMethodTable          = 20;
   vmtFieldTable           = 24;
   vmtTypeInfo             = 28;
   vmtInitTable            = 32;
   vmtAutoTable            = 36;
   vmtIntfTable            = 40;
   vmtMsgStrPtr            = 44;
   vmtMethodStart          = 48;
```

```
   vmtDestroy              = vmtMethodStart;
   vmtNewInstance          = vmtMethodStart+4;
   vmtFreeInstance         = vmtMethodStart+8;
   vmtSafeCallException    = vmtMethodStart+12;
   vmtDefaultHandler       = vmtMethodStart+16;
   vmtAfterConstruction    = vmtMethodStart+20;
   vmtBeforeDestruction    = vmtMethodStart+24;
   vmtDefaultHandlerStr    = vmtMethodStart+28;
```

You should always use the constant names, and never their values, because the VMT table can change, breaking your code.

The following constants will be used for the planned `variant` support:

```
  varEmpty     = $0000;
  varNull      = $0001;
  varSmallint  = $0002;
  varInteger   = $0003;
  varSingle    = $0004;
  varDouble    = $0005;
  varCurrency  = $0006;
  varDate      = $0007;
  varOleStr    = $0008;
  varDispatch  = $0009;
  varError     = $000A;
  varBoolean   = $000B;
  varVariant   = $000C;
  varUnknown   = $000D;
  varByte      = $0011;
  varString    = $0100;
  varAny       = $0101;
  varTypeMask  = $0FFF;
  varArray     = $2000;
  varByRef     = $4000;
```

The following constants are used in the `TVarRec` record:

```
vtInteger    = 0;
vtBoolean    = 1;
vtChar       = 2;
vtExtended   = 3;
vtString     = 4;
vtPointer    = 5;
vtPChar      = 6;
vtObject     = 7;
vtClass      = 8;
vtWideChar   = 9;
vtPWideChar  = 10;
vtAnsiString = 11;
vtCurrency   = 12;
vtVariant    = 13;
vtInterface  = 14;
vtWideString = 15;
vtInt64      = 16;
vtQWord      = 17;
```

The ExceptProc is called when an unhandled exception occurs:

```
Const
  ExceptProc : TExceptProc = Nil;
```

It is set in the objpas unit, but you can set it yourself to change the default exception handling.

Variables

The following variables are defined and initialized in the system unit:

```
var
  output,input,stderr : text;
  exitproc : pointer;
  exitcode : word;
  stackbottom : Longint;
  loweststack : Longint;
```

The variables ExitProc, exitcode are used in the Free Pascal exit scheme. It works similarly to the one in Turbo Pascal:

When a program halts (be it through the call of the Halt function or Exit or through a run-time error), the exit mechanism checks the value of ExitProc. If this one is non-Nil, it is set to Nil, and the procedure is called. If the exit procedure exits, the value of ExitProc is checked again. If it is non-Nil then the above steps are repeated. So if you want to install your exit procedure, you should save the old value of ExitProc (may be non-Nil, since other units could have set it before you did). In your exit procedure you then restore the value of ExitProc, such that if it was non-Nil the exit-procedure can be called.

Listing: refex/ex98.pp

```
Program Example98;

{ Program to demonstrate the exitproc function. }

Var
  OldExitProc : Pointer;

Procedure MyExit;

begin
  Writeln('My Exitproc was called. Exitcode = ',ExitCode);
  { restore old exit procedure }
  ExitProc:=OldExitProc;
end;

begin
  OldExitProc:=ExitProc;
  ExitProc:=@MyExit;
  If ParamCount>0 Then
    Halt(66);
end.
```

The ErrorAddr and ExitCode can be used to check for error-conditions. If ErrorAddr is non-Nil, a run-time error has occurred. If so, ExitCode contains the error code. If ErrorAddr is Nil, then ExitCode contains the argument to Halt or 0 if the program terminated normally.

ExitCode is always passed to the operating system as the exit-code of your process.

Remark: The maximum error code under LINUX is 127.

Under GO32, the following constants are also defined :

```
const
   seg0040 = $0040;
   segA000 = $A000;
   segB000 = $B000;
   segB800 = $B800;
```

These constants allow easy access to the bios/screen segment via mem/absolute.

The randomize function uses a seed stored in the `RandSeed` variable:

```
  RandSeed    : Cardinal;
```

This variable is initialized in the initialization code of the system unit.

13.2 Function list by category

What follows is a listing of the available functions, grouped by category. For each function there is a reference to the page where you can find the function.

File handling

Functions concerning input and output from and to file.

Name	Description	Page
Append	Open a file in append mode	115
Assign	Assign a name to a file	116
Blockread	Read data from a file into memory	118
Blockwrite	Write data from memory to a file	119
Close	Close a file	121
Eof	Check for end of file	126
Eoln	Check for end of line	127
Erase	Delete file from disk	127
Filepos	Position in file	129
Filesize	Size of file	130
Flush	Write file buffers to disk	131
IOresult	Return result of last file IO operation	138
Read	Read from file into variable	150
Readln	Read from file into variable and goto next line	151
Rename	Rename file on disk	151
Reset	Open file for reading	152
Rewrite	Open file for writing	152
Seek	Set file position	154
SeekEof	Set file position to end of file	155

SeekEoln	Set file position to end of line	156
SetTextBuf	Set size of file buffer	158
Truncate	Truncate the file at position	164
Write	Write variable to file	165
WriteLn	Write variable to file and append newline	166

Memory management

Functions concerning memory issues.

Name	Description	Page
Addr	Return address of variable	115
Assigned	Check if a pointer is valid	117
CSeg	Return code segment	123
Dispose	Free dynamically allocated memory	125
DSeg	Return data segment	126
Fillchar	Fill memory region with certain character	130
Fillword	Fill memory region with 16 bit pattern	131
Freemem	Release allocated memory	132
Getmem	Allocate new memory	134
GetMemoryManager	Return current memory manager	134
High	Return highest index of open array or enumerated	135
IsMemoryManagerSet	Is the memory manager set	138
Low	Return lowest index of open array or enumerated	141
Mark	Mark current memory position	142
Maxavail	Return size of largest free memory block	143
Memavail	Return total available memory	143
Move	Move data from one location in memory to another	144
New	Dynamically allocate memory for variable	144
Ofs	Return offset of variable	145
Ptr	Combine segmant and offset to pointer	148
Release	Release memory above mark point	151
Seg	Return segment	156
SetMemoryManager	Set a memory manager	157
Sptr	Return current stack pointer	160
SSeg	Return ESS register value	161

Mathematical routines

Functions connected to calculating and coverting numbers.

Name	Description	Page

Abs	Calculate absolute value	114
Arctan	Calculate inverse tangent	116
Cos	Calculate cosine of angle	123
Dec	Decrease value of variable	124
Exp	Exponentiate	129
Frac	Return fractional part of floating point value	132
Hi	Return high byte/word of value	135
Inc	Increase value of variable	136
Int	Calculate integer part of floating point value	138
Ln	Calculate logarithm	140
Lo	Return low byte/word of value	140
Odd	Is a value odd or even ?	145
Pi	Return the value of pi	147
Power	Raise float to integer power	148
Random	Generate random number	149
Randomize	Initialize random number generator	150
Round	Round floating point value to nearest integer number	153
Sin	Calculate sine of angle	159
Sqr	Calculate the square of a value	160
Sqrt	Calculate the square root of a value	161
Swap	Swap high and low bytes/words of a variable	163
Trunc	Truncate a floating point value	163

String handling

All things connected to string handling.

Name	Description	Page
BinStr	Construct binary representation of integer	117
Chr	Convert ASCII code to character	120
Concat	Concatenate two strings	121
Copy	Copy part of a string	122
Delete	Delete part of a string	124
HexStr	Construct hexadecimal representation of integer	134
Insert	Insert one string in another	137
Length	Return length of string	140
Lowercase	Convert string to all-lowercase	142
Pos	Calculate position of one string in another	147
SetLength	Set length of a string	157
Str	Convert number to string representation	161
StringOfChar	Create string consisting of a number of characters	162
Upcase	Convert string to all-uppercase	164
Val	Convert string to number	165

Operating System functions

Functions that are connected to the operating system.

Name	Description	Page
Chdir	Change working directory	120
Getdir	Return current working directory	133
Halt	Halt program execution	134
Paramcount	Number of parameters with which program was called	146
Paramstr	Retrieve parameters with which program was called	147
Mkdir	Make a directory	144
Rmdir	Remove a directory	153
Runerror	Abort program execution with error condition	154

Miscellaneous functions

Functions that do not belong in one of the other categories.

Name	Description	Page
Break	Abort current loop	119
Continue	Next cycle in current loop	121
Exit	Exit current function or procedure	128
LongJmp	Jump to execution point	141
Ord	Return ordinal value of enumerated type	146
Pred	Return previous value of ordinal type	148
SetJmp	Mark execution point for jump	157
SizeOf	Return size of variable or type	159
Succ	Return next value of ordinal type	162

13.3 Functions and Procedures

Abs

Declaration: `Function Abs (X : Every numerical type) : Every numerical type;`

Description: `Abs` returns the absolute value of a variable. The result of the function has the same type as its argument, which can be any numerical type.

Errors: None.

See also: Round (153)

Listing: refex/ex1.pp

```
Program Example1;

{ Program to demonstrate the Abs function. }
```

```
Var
  r : real;
  i : integer;

begin
  r:=abs(-1.0);    { r:=1.0 }
  i:=abs(-21);     { i:=21 }
end.
```

Addr

Declaration: `Function Addr (X : Any type) : Pointer;`

Description: `Addr` returns a pointer to its argument, which can be any type, or a function or procedure name. The returned pointer isn't typed. The same result can be obtained by the `@` operator, which can return a typed pointer (Programmers' guide).

Errors: None

See also: SizeOf (159)

Listing: refex/ex2.pp

```
Program Example2;

{ Program to demonstrate the Addr function. }

Const Zero : integer = 0;

Var p : pointer;
    i : Integer;

begin
  p:=Addr(p);     { P points to itself }
  p:=Addr(I);     { P points to I }
  p:=Addr(Zero);  { P points to 'Zero' }
end.
```

Append

Declaration: `Procedure Append (Var F : Text);`

Description: `Append` opens an existing file in append mode. Any data written to `F` will be appended to the file. If the file didn't exist, it will be created, contrary to the Turbo Pascal implementation of `Append`, where a file needed to exist in order to be opened by `Append`. Only text files can be opened in append mode.

Errors: If the file can't be created, a run-time error will be generated.

See also: Rewrite (152),Close (121), Reset (152)

Listing: refex/ex3.pp

```
Program Example3;

{ Program to demonstrate the Append function. }

Var f : text;

begin
  Assign (f,'test.txt');
  Rewrite (f);            { file is opened for write, and emptied }
  Writeln (F,'This is the first line of text.txt');
  close (f);
  Append(f);              { file is opened for write, but NOT emptied.
                            any text written to it is appended.}
  Writeln (f,'This is the second line of text.txt');
  close (f);
end.
```

Arctan

Declaration: `Function Arctan (X : Real) : Real;`

Description: `Arctan` returns the Arctangent of X, which can be any Real type. The resulting angle is in radial units.

Errors: None

See also: Sin (159), Cos (123)

Listing: refex/ex4.pp

```
Program Example4;

{ Program to demonstrate the ArcTan function. }

Var R : Real;

begin
  R:=ArcTan(0);        { R:=0 }
  R:=ArcTan(1)/pi;     { R:=0.25 }
end.
```

Assign

Declaration: `Procedure Assign (Var F; Name : String);`

Description: `Assign` assigns a name to F, which can be any file type. This call doesn't open the file, it just assigns a name to a file variable, and marks the file as closed.

Errors: None.

See also: Reset (152), Rewrite (152), Append (115)

Listing: refex/ex5.pp

```
Program Example5;

{ Program to demonstrate the Assign function. }

Var F : text;

begin
  Assign (F,'');
  Rewrite (f);
  { The following can be put in any file by redirecting it
    from the command line.}
  Writeln (f,'This goes to standard output !');
  Close (f);
  Assign (F,'Test.txt');
  rewrite (f);
  writeln (f,'This doesn''t go to standard output !');
  close (f);
end.
```

Assigned

Declaration: `Function Assigned (P : Pointer) : Boolean;`

Description: `Assigned` returns `True` if `P` is non-nil and retuns `False` of `P` is nil. The main use of Assigned is that Procedural variables, method variables and class-type variables also can be passed to `Assigned`.

Errors: None

See also: New (144)

Listing: refex/ex96.pp

```
Program Example96;

{ Program to demonstrate the Assigned function. }

Var P : Pointer;

begin
  If Not Assigned(P) then
    Writeln ('Pointer is initially NIL');
  P:=@P;
  If Not Assigned(P) then
    Writeln('Internal inconsistency')
  else
    Writeln('All is well in FPC')
end.
```

BinStr

Declaration: `Function BinStr (Value : longint; cnt : byte) : String;`

Description: `BinStr` returns a string with the binary representation of `Value`. The string has at most `cnt` characters. (i.e. only the `cnt` rightmost bits are taken into account) To have a complete representation of any longint-type value, you need 32 bits, i.e. `cnt=32`

Errors: None.

See also: Str (161),Val (165),HexStr (134)

Listing: refex/ex82.pp

```
Program example82;

{ Program to demonstrate the BinStr function }

Const Value = 45678;

Var I : longint;

begin
  For I:=8 to 20 do
    Writeln (BinStr(Value,I):20);
end.
```

Blockread

Declaration: `Procedure Blockread (Var F : File; Var Buffer; Var Count : Longint [; var Result : Longint]);`

Description: `Blockread` reads `count` or less records from file `F`. A record is a block of bytes with size specified by the Rewrite (152) or Reset (152) statement.

The result is placed in `Buffer`, which must contain enough room for `Count` records. The function cannot read partial records. If `Result` is specified, it contains the number of records actually read. If `Result` isn't specified, and less than `Count` records were read, a run-time error is generated. This behavior can be controlled by the `{$i}` switch.

Errors: If `Result` isn't specified, then a run-time error is generated if less than `count` records were read.

See also: Blockwrite (119), Close (121), Reset (152), Assign (116)

Listing: refex/ex6.pp

```
Program Example6;

{ Program to demonstrate the BlockRead and BlockWrite functions. }

Var Fin, fout : File;
    NumRead,NumWritten : Word;
    Buf : Array[1..2048] of byte;
    Total : Longint;

begin
  Assign (Fin, Paramstr(1));
  Assign (Fout,Paramstr(2));
  Reset (Fin,1);
  Rewrite (Fout,1);
  Total:=0;
  Repeat
    BlockRead (Fin,buf,Sizeof(buf),NumRead);
    BlockWrite (Fout,Buf,NumRead,NumWritten);
    inc(Total,NumWritten);
```

```
  Until (NumRead=0) or (NumWritten<>NumRead);
  Write ('Copied ',Total,' bytes from file ',paramstr(1));
  Writeln (' to file ',paramstr(2));
  close(fin);
  close(fout);
end.
```

Blockwrite

Declaration: `Procedure Blockwrite (Var F : File; Var Buffer; Var Count : Longint);`

Description: `BlockWrite` writes `count` records from `buffer` to the file `F`.A record is a block of bytes with size specified by the Rewrite (152) or Reset (152) statement.

If the records couldn't be written to disk, a run-time error is generated. This behavior can be controlled by the `{$i}` switch.

Errors: A run-time error is generated if, for some reason, the records couldn't be written to disk.

See also: Blockread (118),Close (121), Rewrite (152), Assign (116)

For the example, see Blockread (118).

Break

Declaration: `Procedure Break;`

Description: `Break` jumps to the statement following the end of the current repetitive statement. The code between the `Break` call and the end of the repetitive statement is skipped. The condition of the repetitive statement is NOT evaluated.

This can be used with `For`, varrepeat and `While` statements.

Note that while this is a procedure, `Break` is a reserved word and hence cannot be redefined.

Errors: None.

See also: Continue (121), Exit (128)

Listing: refex/ex87.pp

```
Program Example87;

{ Program to demonstrate the Break function. }

Var I : longint;

begin
  I:=0;
  While I<10 Do
    begin
    Inc(I);
    If I>5 Then
      Break;
    Writeln (i);
    end;
  I:=0;
  Repeat
```

```
    Inc(I);
    If I>5 Then
      Break;
    Writeln (i);
  Until I>=10;
  For I:=1 to 10 do
    begin
    If I>5 Then
      Break;
    Writeln (i);
    end;
end.
```

Chdir

Declaration: `Procedure Chdir (const S : string);`

Description: `Chdir` changes the working directory of the process to S.

Errors: If the directory S doesn't exist, a run-time error is generated.

See also: Mkdir (144), Rmdir (153)

Listing: refex/ex7.pp

```
Program Example7;

{ Program to demonstrate the ChDir function. }

begin
  {$I-}
  ChDir (ParamStr(1));
  if IOresult<>0 then
    Writeln ('Cannot change to directory : ',paramstr (1));
end.
```

Chr

Declaration: `Function Chr (X : byte) :  Char;`

Description: `Chr` returns the character which has ASCII value X.

Errors: None.

See also: Ord (146), Str (161)

Listing: refex/ex8.pp

```
Program Example8;

{ Program to demonstrate the Chr function. }

begin
  Write (chr(10),chr(13)); { The same effect as Writeln; }
end.
```

Close

Declaration: `Procedure Close (Var F : Anyfiletype);`

Description: `Close` flushes the buffer of the file `F` and closes `F`. After a call to `Close`, data can no longer be read from or written to `F`. To reopen a file closed with `Close`, it isn't necessary to assign the file again. A call to Reset (152) or Rewrite (152) is sufficient.

Errors: None.

See also: Assign (116), Reset (152), Rewrite (152), Flush (131)

Listing: refex/ex9.pp

```
Program Example9;

{ Program to demonstrate the Close function. }

Var F : text;

begin
 Assign (f,'Test.txt');
 ReWrite (F);
 Writeln (F,'Some text written to Test.txt');
 close (f); { Flushes contents of buffer to disk,
              closes the file. Omitting this may
              cause data NOT to be written to disk.}
end.
```

Concat

Declaration: `Function Concat (S1,S2 [,S3, ... ,Sn]) : String;`

Description: `Concat` concatenates the strings `S1`,`S2` etc. to one long string. The resulting string is truncated at a length of 255 bytes. The same operation can be performed with the `+` operation.

Errors: None.

See also: Copy (122), Delete (124), Insert (137), Pos (147), Length (140)

Listing: refex/ex10.pp

```
Program Example10;

{ Program to demonstrate the Concat function. }
Var
  S : String;

begin
  S:=Concat('This can be done',' Easier ','with the + operator !');
end.
```

Continue

Declaration: `Procedure Continue;`

Description: `Continue` jumps to the end of the current repetitive statement. The code between the `Continue` call and the end of the repetitive statement is skipped. The condition of the repetitive statement is then checked again.

This can be used with `For`, varrepeat and `While` statements.

Note that while this is a procedure, `Continue` is a reserved word and hence cannot be redefined.

Errors: None.

See also: Break (119), Exit (128)

Listing: refex/ex86.pp

```
Program Example86;

{ Program to demonstrate the Continue function. }

Var I : longint;

begin
  I:=0;
  While I<10 Do
    begin
    Inc(I);
    If I<5 Then
      Continue;
    Writeln (i);
    end;
  I:=0;
  Repeat
    Inc(I);
    If I<5 Then
      Continue;
    Writeln (i);
  Until I>=10;
  For I:=1 to 10 do
    begin
    If I<5 Then
      Continue;
    Writeln (i);
    end;
end.
```

Copy

Declaration: `Function Copy (Const S : String;Index : Integer;Count : Byte) : String;`

Description: `Copy` returns a string which is a copy if the `Count` characters in `S`, starting at position `Index`. If `Count` is larger than the length of the string `S`, the result is truncated. If `Index` is larger than the length of the string `S`, then an empty string is returned.

Errors: None.

See also: Delete (124), Insert (137), Pos (147)

Listing: refex/ex11.pp

```
Program Example11;

{ Program to demonstrate the Copy function. }

Var S,T : String;

begin
  T:='1234567';
  S:=Copy (T,1,2);    { S:='12'   }
  S:=Copy (T,4,2);    { S:='45'   }
  S:=Copy (T,4,8);    { S:='4567' }
end.
```

Cos

Declaration: `Function Cos (X : Real) : Real;`

Description: `Cos` returns the cosine of X, where X is an angle, in radians.

If the absolute value of the argument is larger than `2^63`, then the result is undefined.

Errors: None.

See also: Arctan (116), Sin (159)

Listing: refex/ex12.pp

```
Program Example12;

{ Program to demonstrate the Cos function. }

Var R : Real;

begin
  R:=Cos(Pi);      { R:=-1 }
  R:=Cos(Pi/2);    { R:=0  }
  R:=Cos(0);       { R:=1  }
end.
```

CSeg

Declaration: `Function CSeg : Word;`

Description: `CSeg` returns the Code segment register. In Free Pascal, it returns always a zero, since Free Pascal is a 32 bit compiler.

Errors: None.

See also: DSeg (126), Seg (156), Ofs (145), Ptr (148)

Listing: refex/ex13.pp

```
Program Example13;

{ Program to demonstrate the CSeg function. }
```

```
var W : word;

begin
  W:=CSeg; {W:=0, provided for compatibility,
                  FPC is 32 bit.}
end.
```

Dec

Declaration: `Procedure Dec (Var X : Any ordinal type[; Decrement : Longint]);`

Description: `Dec` decreases the value of `X` with `Decrement`. If `Decrement` isn't specified, then 1 is taken as a default.

Errors: A range check can occur, or an underflow error, if you try to decrease `X` below its minimum value.

See also: Inc (136)

Listing: refex/ex14.pp

```
Program Example14;

{ Program to demonstrate the Dec function. }

Var
  I  : Integer;
  L  : Longint;
  W  : Word;
  B  : Byte;
  Si : ShortInt;

begin
 I:=1;
 L:=2;
 W:=3;
 B:=4;
 Si:=5;
 Dec (i);      { i:=0  }
 Dec (L,2);    { L:=0  }
 Dec (W,2);    { W:=1  }
 Dec (B,-2);   { B:=6  }
 Dec (Si,0);   { Si:=5 }
end.
```

Delete

Declaration: `Procedure Delete (var S : string;Index : Integer;Count : Integer);`

Description: `Delete` removes `Count` characters from string `S`, starting at position `Index`. All characters after the delected characters are shifted `Count` positions to the left, and the length of the string is adjusted.

Errors: None.

See also: Copy (122),Pos (147),Insert (137)

Listing: refex/ex15.pp

```
Program Example15;

{ Program to demonstrate the Delete function. }

Var
  S : String;

begin
  S:='This is not easy !';
  Delete (S,9,4); { S:='This is easy !' }
end.
```

Dispose

Declaration: `Procedure Dispose (P : pointer);`
`Procedure Dispiose (P : Typed Pointer; Des : Procedure);`

Description: The first form `Dispose` releases the memory allocated with a call to New (144). The pointer P must be typed. The released memory is returned to the heap.

The second form of `Dispose` accepts as a first parameter a pointer to an object type, and as a second parameter the name of a destructor of this object. The destructor will be called, and the memory allocated for the object will be freed.

Errors: An error will occur if the pointer doesn't point to a location in the heap.

See also: New (144), Getmem (134), Freemem (132)

Listing: refex/ex16.pp

```
Program Example16;

{ Program to demonstrate the Dispose and New functions. }

Type SS = String[20];

     AnObj = Object
       I : integer;
       Constructor Init;
       Destructor Done;
       end;

Var
  P : ^SS;
  T : ^AnObj;

Constructor Anobj.Init;

begin
 Writeln ('Initializing an instance of AnObj !');
end;

Destructor AnObj.Done;

begin
  Writeln ('Destroying an instance of AnObj !');
end;
```

```
begin
  New (P);
  P^:='Hello, World!';
  Dispose (P);
  { P is undefined from here on !}
  New(T,Init);
  T^.i:=0;
  Dispose (T,Done);
end.
```

DSeg

Declaration: `Function DSeg : Word;`

Description: `DSeg` returns the data segment register. In Free Pascal, it returns always a zero, since Free Pascal is a 32 bit compiler.

Errors: None.

See also: CSeg (123), Seg (156), Ofs (145), Ptr (148)

Listing: refex/ex17.pp

```
Program Example17;

{ Program to demonstrate the DSeg function. }

Var
  W : Word;

begin
  W:=DSeg; {W:=0, This function is provided for compatibility,
                   FPC is a 32 bit comiler.}
end.
```

Eof

Declaration: `Function Eof [(F : Any file type)] : Boolean;`

Description: `Eof` returns `True` if the file-pointer has reached the end of the file, or if the file is empty. In all other cases `Eof` returns `False`. If no file `F` is specified, standard input is assumed.

Errors: None.

See also: Eoln (127), Assign (116), Reset (152), Rewrite (152)

Listing: refex/ex18.pp

```
Program Example18;

{ Program to demonstrate the Eof function. }

Var T1,T2 : text;
    C : Char;
```

```
begin
  { Set file to read from. Empty means from standard input.}
  assign (t1,paramstr(1));
  reset (t1);
  { Set file to write to. Empty means to standard output. }
  assign (t2,paramstr(2));
  rewrite (t2);
  While not eof(t1) do
    begin
    read (t1,C);
    write (t2,C);
    end;
  Close (t1);
  Close (t2);
end.
```

Eoln

Declaration: `Function Eoln [(F : Text)] : Boolean;`

Description: `Eof` returns `True` if the file pointer has reached the end of a line, which is demarcated by a line-feed character (ASCII value 10), or if the end of the file is reached. In all other cases `Eof` returns `False`. If no file `F` is specified, standard input is assumed. It can only be used on files of type `Text`.

Errors: None.

See also: Eof (126), Assign (116), Reset (152), Rewrite (152)

Listing: refex/ex19.pp

```
Program Example19;

{ Program to demonstrate the Eoln function. }

begin
  { This program waits for keyboard input. }
  { It will print True when an empty line is put in,
    and false when you type a non-empty line.
    It will only stop when you press enter.}
  While not Eoln do
    Writeln (eoln);
end.
```

Erase

Declaration: `Procedure Erase (Var F : Any file type);`

Description: `Erase` removes an unopened file from disk. The file should be assigned with `Assign`, but not opened with `Reset` or `Rewrite`

Errors: A run-time error will be generated if the specified file doesn't exist, or is opened by the program.

See also: Assign (116)

Listing: refex/ex20.pp

```
Program Example20;

{ Program to demonstrate the Erase function. }

Var F : Text;

begin
  { Create a file with a line of text in it}
  Assign (F,'test.txt');
  Rewrite (F);
  Writeln (F,'Try and find this when I''m finished !');
  close (f);
  { Now remove the file }
  Erase (f);
end.
```

Exit

Declaration: `Procedure Exit ([Var X : return type )];`

Description: `Exit` exits the current subroutine, and returns control to the calling routine. If invoked in the main program routine, exit stops the program. The optional argument X allows to specify a return value, in the case `Exit` is invoked in a function. The function result will then be equal to X.

Errors: None.

See also: Halt (134)

Listing: refex/ex21.pp

```
Program Example21;

{ Program to demonstrate the Exit function. }

Procedure DoAnExit (Yes : Boolean);

{ This procedure demonstrates the normal Exit }

begin
  Writeln ('Hello from DoAnExit !');
  If Yes then
    begin
    Writeln ('Bailing out early.');
    exit;
    end;
  Writeln ('Continuing to the end.');
end;

Function Positive (Which : Integer) : Boolean;

{ This function demonstrates the extra FPC feature of Exit :
  You can specify a return value for the function }

begin
  if Which>0 then
    exit (True)
```

```
  else
    exit (False);
end;

begin
  { This call will go to the end }
  DoAnExit (False);
  { This call will bail out early }
  DoAnExit (True);
  if Positive (-1) then
    Writeln ('The compiler is nuts, -1 is not positive.')
  else
    Writeln ('The compiler is not so bad, -1 seems to be negative.');
end.
```

Exp

Declaration: `Function Exp (Var X : Real) : Real;`

Description: `Exp` returns the exponent of X, i.e. the number e to the power X.

Errors: None.

See also: Ln (140), Power (148)

Listing: refex/ex22.pp

```
Program Example22;

{ Program to demonstrate the Exp function. }

begin
  Writeln (Exp(1):8:2); { Should print 2.72 }
end.
```

Filepos

Declaration: `Function Filepos (Var F : Any file type) : Longint;`

Description: `Filepos` returns the current record position of the file-pointer in file F. It cannot be invoked with a file of type `Text`. If you try to do this, a compiler error will be generated.

Errors: None.

See also: Filesize (130)

Listing: refex/ex23.pp

```
Program Example23;

{ Program to demonstrate the FilePos function. }

Var F : File of Longint;
    L,FP : longint;

begin
```

```
  { Fill a file with data :
    Each position contains the position ! }
  Assign (F,'test.dat');
  Rewrite (F);
  For L:=0 to 100 do
    begin
    FP:=FilePos(F);
    Write (F,FP);
    end;
  Close (F);
  Reset (F);
  { If all goes well, nothing is displayed here. }
  While not (Eof(F)) do
    begin
    FP:=FilePos (F);
    Read (F,L);
    if L<>FP then
      Writeln ('Something wrong: Got ',l,' on pos ',FP);
    end;
  Close (F);
  Erase (f);
end.
```

Filesize

Declaration: `Function Filesize (Var F : Any file type) : Longint;`

Description: `Filesize` returns the total number of records in file F. It cannot be invoked with a file of type `Text`. (under LINUX, this also means that it cannot be invoked on pipes.) If F is empty, 0 is returned.

Errors: None.

See also: Filepos (129)

Listing: refex/ex24.pp

```
Program Example24;

{ Program to demonstrate the FileSize function. }

Var F : File Of byte;
    L : File Of Longint;

begin
  Assign (F,paramstr(1));
  Reset (F);
  Writeln ('File size in bytes : ',FileSize(F));
  Close (F);
  Assign (L,paramstr (1));
  Reset (L);
  Writeln ('File size in Longints : ',FileSize(L));
  Close (f);
end.
```

Fillchar

Declaration: `Procedure Fillchar (Var X;Count : Longint;Value : char or byte);;`

Description: `Fillchar` fills the memory starting at `X` with `Count` bytes or characters with value equal to `Value`.

Errors: No checking on the size of `X` is done.

See also: Fillword (131), Move (144)

Listing: refex/ex25.pp

```
Program Example25;

{ Program to demonstrate the FillChar function. }

Var S : String[10];
    I : Byte;
begin
  For i:=10 downto 0 do
    begin
    { Fill S with i spaces }
    FillChar (S,SizeOf(S),' ');
    { Set Length }
    SetLength(S,I);
    Writeln (s,'*');
    end;
end.
```

Fillword

Declaration: `Procedure Fillword (Var X;Count : Longint;Value : Word);;`

Description: `Fillword` fills the memory starting at `X` with `Count` words with value equal to `Value`.

Errors: No checking on the size of `X` is done.

See also: Fillchar (130), Move (144)

Listing: refex/ex76.pp

```
Program Example76;

{ Program to demonstrate the FillWord function. }

Var W : Array[1..100] of Word;

begin
  { Quick initialization of array W}
  FillWord(W,100,0);
end.
```

Flush

Declaration: `Procedure Flush (Var F : Text);`

Description: `Flush` empties the internal buffer of an opened file `F` and writes the contents to disk. The file is *not* closed as a result of this call.

Errors: If the disk is full, a run-time error will be generated.

See also: Close (121)

Listing: refex/ex26.pp

```
Program Example26;

{ Program to demonstrate the Flush function. }

Var F : Text;

begin
  { Assign F to standard output }
  Assign (F,'');
  Rewrite (F);
  Writeln (F,'This line is written first, but appears later !');
  { At this point the text is in the internal pascal buffer,
    and not yet written to standard output }
  Writeln ('This line appears first, but is written later !');
  { A writeln to 'output' always causes a flush - so this text is
    written to screen }
  Flush (f);
  { At this point, the text written to F is written to screen. }
  Write (F,'Finishing ');
  Close (f);  { Closing a file always causes a flush first }
  Writeln ('off.');
end.
```

Frac

Declaration: `Function Frac (X : Real) : Real;`

Description: `Frac` returns the non-integer part of X.

Errors: None.

See also: Round (153), Int (138)

Listing: refex/ex27.pp

```
Program Example27;

{ Program to demonstrate the Frac function. }

Var R : Real;

begin
  Writeln (Frac (123.456):0:3);  { Prints  O.456 }
  Writeln (Frac (-123.456):0:3); { Prints -O.456 }
end.
```

Freemem

Declaration: `Procedure Freemem (Var P : pointer; Count : Longint);`

Description: `Freemem` releases the memory occupied by the pointer `P`, of size `Count` (in bytes), and returns it to the heap. `P` should point to the memory allocated to a dynamical variable.

Errors: An error will occur when `P` doesn't point to the heap.

See also: Getmem (134), New (144), Dispose (125)

Listing: refex/ex28.pp

```
Program Example28;

{ Program to demonstrate the FreeMem and GetMem functions. }

Var P : Pointer;
    MM : Longint;

begin
  { Get memory for P }
  MM:=MemAvail;
  Writeln ('Memory available before GetMem : ',MemAvail);
  GetMem (P,80);
  MM:=MM-Memavail;
  Write   ('Memory available after GetMem  : ',MemAvail);
  Writeln (' or ',MM,' bytes less than before the call.');
  { fill it with spaces }
  FillChar (P^,80,' ');
  { Free the memory again }
  FreeMem (P,80);
  Writeln ('Memory available after FreeMem : ',MemAvail);
end.
```

Getdir

Declaration: `Procedure Getdir (drivenr : byte;var dir : string);`

Description: `Getdir` returns in `dir` the current directory on the drive `drivenr`, where drivenr is 1 for the first floppy drive, 3 for the first hard disk etc. A value of 0 returns the directory on the current disk. On LINUX, `drivenr` is ignored, as there is only one directory tree.

Errors: An error is returned under DOS, if the drive requested isn't ready.

See also: Chdir (120)

Listing: refex/ex29.pp

```
Program Example29;

{ Program to demonstrate the GetDir function. }

Var S : String;

begin
  GetDir (0,S);
  Writeln ('Current directory is : ',S);
end.
```

Getmem

Declaration: `Procedure Getmem (var p : pointer;size : Longint);`

Description: `Getmem` reserves `Size` bytes memory on the heap, and returns a pointer to this memory in `p`. If no more memory is available, nil is returned.

Errors: None.

See also: Freemem (132), Dispose (125), New (144)

For an example, see Freemem (132).

GetMemoryManager

Declaration: `procedure GetMemoryManager(var MemMgr: TMemoryManager);`

Description: `GetMemoryManager` stores the current Memory Manager record in `MemMgr`.

Errors: None.

See also: SetMemoryManager (157), IsMemoryManagerSet (138).

For an example, see Programmers' guide.

Halt

Declaration: `Procedure Halt [(Errnum : byte)];`

Description: `Halt` stops program execution and returns control to the calling program. The optional argument `Errnum` specifies an exit value. If omitted, zero is returned.

Errors: None.

See also: Exit (128)

Listing: refex/ex30.pp

```
Program Example30;

{ Program to demonstrate the Halt function. }

begin
 Writeln ('Before Halt.');
 Halt (1); { Stop with exit code 1 }
 Writeln ('After Halt doesn''t get executed.');
end.
```

HexStr

Declaration: `Function HexStr (Value : longint; cnt : byte) : String;`

Description: `HexStr` returns a string with the hexadecimal representation of `Value`. The string has at most `cnt` charaters. (i.e. only the `cnt` rightmost nibbles are taken into account) To have a complete representation of a Longint-type value, you need 8 nibbles, i.e. `cnt=8`.

Errors: None.

See also: Str (161), Val (165), BinStr (117)

Listing: refex/ex81.pp

```
Program example81;

{ Program to demonstrate the HexStr function }

Const Value = 45678;

Var I : longint;

begin
  For I:=1 to 10 do
    Writeln (HexStr(Value,I));
end.
```

Hi

Declaration: `Function Hi (X : Ordinal type) : Word or byte;`

Description: `Hi` returns the high byte or word from `X`, depending on the size of X. If the size of X is 4, then the high word is returned. If the size is 2 then the high byte is returned. `Hi` cannot be invoked on types of size 1, such as byte or char.

Errors: None

See also: Lo (140)

Listing: refex/ex31.pp

```
Program Example31;

{ Program to demonstrate the Hi function. }

var
  L : Longint;
  W : Word;

begin
  L:=1 Shl 16;     { = $10000 }
  W:=1 Shl 8;      { = $100 }
  Writeln (Hi(L)); { Prints 1 }
  Writeln (Hi(W)); { Prints 1 }
end.
```

High

Declaration: `Function High (Type identifier or variable reference) : Ordinal;`

Description: The return value of `High` depends on it's argument:

1. If the argument is an ordinal type, `High` returns the lowest value in the range of the given ordinal type.
2. If the argument is an array type or an array type variable then `High` returns the highest possible value of it's index.

3.If the argument is an open array identifier in a function or procedure, then `High` returns the highest index of the array, as if the array has a zero-based index.

The return type is always the same type as the type of the argument (This can lead to some nasty surprises !).

Errors: None.

See also: Low (141), Ord (146), Pred (148), Succ (162)

Listing: refex/ex80.pp

```
Program example80;

{ Example to demonstrate the High and Low functions. }

Type TEnum = ( North, East, South, West );
     TRange = 14..55;
     TArray = Array [2..10] of Longint;

Function Average (Row : Array of Longint) : Real;

Var I : longint;
    Temp : Real;


begin
  Temp := Row[0];
  For I := 1 to High(Row) do
     Temp := Temp + Row[i];
  Average := Temp / (High(Row)+1);
end;

Var A : TEnum;
    B : TRange;
    C : TArray;
    I : longint;

begin
  Writeln ('TEnum  goes from : ',Ord(Low(TEnum)),' to ', Ord(high(TEnum)),'.');
  Writeln ('A      goes from : ',Ord(Low(A)),' to ', Ord(high(A)),'.');
  Writeln ('TRange goes from : ',Ord(Low(TRange)),' to ', Ord(high(TRange)),'.');
  Writeln ('B      goes from : ',Ord(Low(B)),' to ', Ord(high(B)),'.');
  Writeln ('TArray index goes from : ',Ord(Low(TArray)),' to ', Ord(high(TArray)),'.');
  Writeln ('C index      goes from : ',Low(C),' to ', high(C),'.');
  For I:=Low(C) to High(C) do
    C[i]:=I;
  Writeln ('Average :',Average(c));
  Write ('Type of return value is always same as type of argument:');
  Writeln(high(high(word)));
end.
```

Inc

Declaration: `Procedure Inc (Var X : Any ordinal type[; Increment : Longint]);`

Description: `Inc` increases the value of `X` with `Increment`. If `Increment` isn't specified, then 1 is taken as a default.

Errors: If range checking is on, then A range check can occur, or an overflow error, if you try to increase X over its maximum value.

See also: Dec (124)

Listing: refex/ex32.pp

```
Program Example32;

{ Program to demonstrate the Inc function. }

Const
  C : Cardinal  = 1;
  L : Longint   = 1;
  I : Integer   = 1;
  W : Word      = 1;
  B : Byte      = 1;
  SI : ShortInt = 1;
  CH : Char     = 'A';

begin
  Inc (C);        { C:=2     }
  Inc (L,5);      { L:=6     }
  Inc (I,-3);     { I:=-2    }
  Inc (W,3);      { W:=4     }
  Inc (B,100);    { B:=101   }
  Inc (SI,-3);    { Si:=-2   }
  Inc (CH,1);     { ch:='B'  }
end.
```

Insert

Declaration: `Procedure Insert (Const Source : String;var S : String;Index : Longint);`

Description: `Insert` inserts string `Source` in string `S`, at position `Index`, shifting all characters after `Index` to the right. The resulting string is truncated at 255 characters, if needed. (i.e. for shortstrings)

Errors: None.

See also: Delete (124), Copy (122), Pos (147)

Listing: refex/ex33.pp

```
Program Example33;

{ Program to demonstrate the Insert function. }

Var S : String;

begin
  S:='Free Pascal is difficult to use !';
  Insert ('NOT ',S,pos('difficult',S));
  writeln (s);
end.
```

IsMemoryManagerSet

Declaration: `function IsMemoryManagerSet: Boolean;`

Description: `IsMemoryManagerSet` will return `True` if the memory manager has been set to another value than the system heap manager, it will return `False` otherwise.

Errors: None.

See also: SetMemoryManager (157), GetMemoryManager (134)

Int

Declaration: `Function Int (X : Real) : Real;`

Description: `Int` returns the integer part of any Real `X`, as a Real.

Errors: None.

See also: Frac (132), Round (153)

Listing: refex/ex34.pp

```
Program Example34;

{ Program to demonstrate the Int function. }

begin
  Writeln (Int(123.456):0:1);  { Prints  123.0 }
  Writeln (Int(-123.456):0:1); { Prints -123.0 }
end.
```

IOresult

Declaration: `Function IOresult : Word;`

Description: IOresult contains the result of any input/output call, when the `{$i-}` compiler directive is active, disabling IO checking. When the flag is read, it is reset to zero. If `IOresult` is zero, the operation completed successfully. If non-zero, an error occurred. The following errors can occur:

DOS errors :

2 File not found.

3 Path not found.

4 Too many open files.

5 Access denied.

6 Invalid file handle.

12 Invalid file-access mode.

15 Invalid disk number.

16 Cannot remove current directory.

17 Cannot rename across volumes.

I/O errors :

100 Error when reading from disk.

101 Error when writing to disk.

102 File not assigned.

103 File not open.

104 File not opened for input.

105 File not opened for output.

106 Invalid number.

Fatal errors :

150 Disk is write protected.

151 Unknown device.

152 Drive not ready.

153 Unknown command.

154 CRC check failed.

155 Invalid drive specified..

156 Seek error on disk.

157 Invalid media type.

158 Sector not found.

159 Printer out of paper.

160 Error when writing to device.

161 Error when reading from device.

162 Hardware failure.

Errors: None.

See also: All I/O functions.

Listing: refex/ex35.pp

```
Program Example35;

{ Program to demonstrate the IOResult function. }

Var F : text;

begin
  Assign (f,paramstr(1));
  {$i-}
  Reset (f);
  {$i+}
  If IOresult<>0 then
    writeln ('File ',paramstr(1),' doesn''t exist')
  else
    writeln ('File ',paramstr(1),' exists');
end.
```

Length

Declaration: `Function Length (S : String) : Byte;`

Description: `Length` returns the length of the string `S`, which is limited to 255 for shortstrings. If the strings `S` is empty, 0 is returned. *Note:* The length of the string `S` is stored in `S[0]` for shortstrings only. Ansistrings have their length stored elsewhere, the `Length` fuction should always be used on ansistrings.

Errors: None.

See also: Pos (147)

Listing: refex/ex36.pp

```
Program Example36;

{ Program to demonstrate the Length function. }

Var S : String;
    I : Integer;

begin
  S:='';
  for i:=1 to 10 do
    begin
    S:=S+'*';
    Writeln (Length(S):2,' : ',s);
    end;
end.
```

Ln

Declaration: `Function Ln (X : Real) : Real;`

Description: `Ln` returns the natural logarithm of the Real parameter `X`. `X` must be positive.

Errors: An run-time error will occur when `X` is negative.

See also: Exp (129), Power (148)

Listing: refex/ex37.pp

```
Program Example37;

{ Program to demonstrate the Ln function. }

begin
  Writeln (Ln(1));       { Prints 0 }
  Writeln (Ln(Exp(1))); { Prints 1 }
end.
```

Lo

Declaration: `Function Lo (O : Word or Longint) : Byte or Word;`

Description: Lo returns the low byte of its argument if this is of type Integer or Word. It returns the low word of its argument if this is of type Longint or Cardinal.

Errors: None.

See also: Ord (146), Chr (120), Hi (135)

Listing: refex/ex38.pp

```
Program Example38;

{ Program to demonstrate the Lo function. }

Var L : Longint;
    W : Word;

begin
  L:=(1 Shl 16) + (1 Shl 4);   { $10010 }
  Writeln (Lo(L));             { Prints 16 }
  W:=(1 Shl 8) + (1 Shl 4);    { $110    }
  Writeln (Lo(W));             { Prints 16 }
end.
```

LongJmp

Declaration: `Procedure LongJmp (Var env : Jmp_Buf; Value : Longint);`

Description: LongJmp jumps to the adress in the env jmp_buf, and resores the registers that were stored in it at the corresponding SetJmp (157) call. In effect, program flow will continue at the SetJmp call, which will return value instead of 0. If you pas a value equal to zero, it will be converted to 1 before passing it on. The call will not return, so it must be used with extreme care. This can be used for error recovery, for instance when a segmentation fault occurred.

Errors: None.

See also: SetJmp (157)

For an example, see SetJmp (157)

Low

Declaration: `Function Low (Type identifier or variable reference) : Longint;`

Description: The return value of Low depends on it's argument:

1. If the argument is an ordinal type, Low returns the lowest value in the range of the given ordinal type.
2. If the argument is an array type or an array type variable then Low returns the lowest possible value of it's index.

The return type is always the same type as the type of the argument

Errors: None.

See also: High (135), Ord (146), Pred (148), Succ (162)

for an example, see High (135).

Lowercase

Declaration: `Function Lowercase (C : Char or String) : Char or String;`

Description: `Lowercase` returns the lowercase version of its argument `C`. If its argument is a string, then the complete string is converted to lowercase. The type of the returned value is the same as the type of the argument.

Errors: None.

See also: Upcase (164)

Listing: refex/ex73.pp

```
Program Example73;

{ Program to demonstrate the Lowercase function. }

Var I : Longint;

begin
  For i:=ord('A') to ord('Z') do
    write (lowercase(chr(i)));
  Writeln;
  Writeln (Lowercase('ABCDEFGHIJKLMNOPQRSTUVWXYZ'));
end.
```

Mark

Declaration: `Procedure Mark (Var P : Pointer);`

Description: `Mark` copies the current heap-pointer to `P`.

Errors: None.

See also: Getmem (134), Freemem (132), New (144), Dispose (125), Maxavail (143)

Listing: refex/ex39.pp

```
Program Example39;

{ Program to demonstrate the Mark and Release functions. }

Var P,PP,PPP,MM : Pointer;

begin
  Getmem (P,100);
  Mark (MM);
  Writeln ('Getmem 100   : Memory available : ',MemAvail,' (marked)');
  GetMem (PP,1000);
  Writeln ('Getmem 1000  : Memory available : ',MemAvail);
  GetMem (PPP,100000);
  Writeln ('Getmem 10000 : Memory available : ',MemAvail);
  Release (MM);
  Writeln ('Released     : Memory available : ',MemAvail);
  { At this point, PP and PPP are invalid ! }
end.
```

Maxavail

Declaration: `Function Maxavail : Longint;`

Description: `Maxavail` returns the size, in bytes, of the biggest free memory block in the heap.

Remark: The heap grows dynamically if more memory is needed than is available.

Errors: None.

See also: Release (151), Memavail (143),Freemem (132), Getmem (134)

Listing: refex/ex40.pp

```
Program Example40;

{ Program to demonstrate the MaxAvail function. }

Var
  P : Pointer;
  I : longint;

begin
  { This will allocate memory until there is no more memory}
  I:=0;
  While MaxAvail>=1000 do
    begin
    Inc (I);
    GetMem (P,1000);
    end;
  { Default 4MB heap is allocated, so 4000 blocks
    should be allocated.
    When compiled with the -Ch10000 switch, the program
    will be able to allocate 10 block }
  Writeln ('Allocated ',i,' blocks of 1000 bytes');
end.
```

Memavail

Declaration: `Function Memavail : Longint;`

Description: `Memavail` returns the size, in bytes, of the free heap memory.

Remark: The heap grows dynamically if more memory is needed than is available.

Errors: None.

See also: Maxavail (143),Freemem (132), Getmem (134)

Listing: refex/ex41.pp

```
Program Example41;

{ Program to demonstrate the MemAvail function. }

Var
  P, PP : Pointer;

begin
```

```
  GetMem (P,100);
  GetMem (PP,10000);
  FreeMem (P,100);
  { Due to the heap fragmentation introduced
    By the previous calls, the maximum amount of memory
    isn't equal to the maximum block size available. }
  Writeln ('Total heap available    (Bytes) : ',MemAvail);
  Writeln ('Largest block available (Bytes) : ',MaxAvail);
end.
```

Mkdir

Declaration: `Procedure Mkdir (const S : string);`

Description: `Mkdir` creates a new directory `S`.

Errors: If a parent-directory of directory `S` doesn't exist, a run-time error is generated.

See also: Chdir (120), Rmdir (153)

For an example, see Rmdir (153).

Move

Declaration: `Procedure Move (var Source,Dest;Count : Longint);`

Description: `Move` moves `Count` bytes from `Source` to `Dest`.

Errors: If either `Dest` or `Source` is outside the accessible memory for the process, then a run-time error will be generated. With older versions of the compiler, a segmentation-fault will occur.

See also: Fillword (131), Fillchar (130)

Listing: refex/ex42.pp

```
Program Example42;

{ Program to demonstrate the Move function. }

Var S1,S2 : String [30];

begin
  S1:='Hello World !';
  S2:='Bye, bye    !';
  Move (S1,S2,Sizeof(S1));
  Writeln (S2);
end.
```

New

Declaration: `Procedure New (Var P : Pointer[, Constructor]);`

Description: `New` allocates a new instance of the type pointed to by `P`, and puts the address in `P`. If P is an object, then it is possible to specify the name of the constructor with which the instance will be created.

Errors: If not enough memory is available, `Nil` will be returned.

See also: Dispose (125), Freemem (132), Getmem (134), Memavail (143), Maxavail (143)

For an example, see Dispose (125).

Odd

Declaration: `Function Odd (X : Longint) : Boolean;`

Description: `Odd` returns `True` if `X` is odd, or `False` otherwise.

Errors: None.

See also: Abs (114), Ord (146)

Listing: refex/ex43.pp

```
Program Example43;

{ Program to demonstrate the Odd function. }

begin
  If Odd(1) Then
    Writeln ('Everything OK with 1 !');
  If Not Odd(2) Then
    Writeln ('Everything OK with 2 !');
end.
```

Ofs

Declaration: `Function Ofs Var X : Longint;`

Description: `Ofs` returns the offset of the address of a variable. This function is only supported for compatibility. In Free Pascal, it returns always the complete address of the variable, since Free Pascal is a 32 bit compiler.

Errors: None.

See also: DSeg (126), CSeg (123), Seg (156), Ptr (148)

Listing: refex/ex44.pp

```
Program Example44;

{ Program to demonstrate the Ofs function. }

Var W : Pointer;


begin
  W:=Pointer(Ofs(W)); { W contains its own offset. }
end.
```

Ord

Declaration: `Function Ord (X : Any ordinal type) : Longint;`

Description: `Ord` returns the Ordinal value of a ordinal-type variable `X`.

Errors: None.

See also: Chr (120), Succ (162), Pred (148), High (135), Low (141)

Listing: refex/ex45.pp

```
Program Example45;

{ Program to demonstrate the Ord,Pred,Succ functions. }

Type
  TEnum = (Zero, One, Two, Three, Four);

Var
  X : Longint;
  Y : TEnum;

begin
  X:=125;
  Writeln (Ord(X));  { Prints 125 }
  X:=Pred(X);
  Writeln (Ord(X));  { prints 124 }
  Y:= One;
  Writeln (Ord(y));  { Prints 1 }
  Y:=Succ(Y);
  Writeln (Ord(Y));  { Prints 2}
end.
```

Paramcount

Declaration: `Function Paramcount : Longint;`

Description: `Paramcount` returns the number of command-line arguments. If no arguments were given to the running program, `0` is returned.

Errors: None.

See also: Paramstr (147)

Listing: refex/ex46.pp

```
Program Example46;

{ Program to demonstrate the ParamCount and ParamStr functions. }
Var
  I : Longint;

begin
  Writeln (paramstr(0),' : Got ',ParamCount,' command-line parameters: ');
  For i:=1 to ParamCount do
    Writeln (ParamStr (i));
end.
```

Paramstr

Declaration: `Function Paramstr (L : Longint) :  String;`

Description: `Paramstr` returns the `L`-th command-line argument. `L` must be between `0` and `Paramcount`, these values included. The zeroth argument is the name with which the program was started.

In all cases, the command-line will be truncated to a length of 255, even though the operating system may support bigger command-lines. If you want to access the complete command-line, you must use the `argv` pointer to access the Real values of the command-line parameters.

Errors: None.

See also: Paramcount (146)

For an example, see Paramcount (146).

Pi

Declaration: `Function Pi :  Real;`

Description: `Pi` returns the value of Pi (3.1415926535897932385).

Errors: None.

See also: Cos (123), Sin (159)

Listing: refex/ex47.pp

```
Program Example47;

{ Program to demonstrate the Pi function. }

begin
  Writeln (Pi);         {3.1415926}
  Writeln (Sin(Pi));
end.
```

Pos

Declaration: `Function Pos (Const Substr :  String;Const S : String) :  Byte;`

Description: `Pos` returns the index of `Substr` in `S`, if `S` contains `Substr`. In case `Substr` isn't found, `0` is returned. The search is case-sensitive.

Errors: None

See also: Length (140), Copy (122), Delete (124), Insert (137)

Listing: refex/ex48.pp

```
Program Example48;

{ Program to demonstrate the Pos function. }

Var
  S : String;
```

```
begin
  S:='The first space in this sentence is at position : ';
  Writeln (S,pos(' ',S));
  S:='The last letter of the alphabet doesn''t appear in this sentence ';
  If (Pos ('Z',S)=0) and (Pos('z',S)=0) then
    Writeln (S);
end.
```

Power

Declaration: `Function Power (base,expon : Real) : Real;`

Description: `Power` returns the value of `base` to the power `expon`. `Base` and `expon` can be of type Longint, in which case the result will also be a Longint.

The function actually returns `Exp(expon*Ln(base))`

Errors: None.

See also: Exp (129), Ln (140)

Listing: refex/ex78.pp

```
Program Example78;

{ Program to demonstrate the Power function. }

begin
  Writeln (Power(exp(1.0),1.0):8:2); { Should print 2.72 }
end.
```

Pred

Declaration: `Function Pred (X : Any ordinal type) : Same type;`

Description: `Pred` returns the element that precedes the element that was passed to it. If it is applied to the first value of the ordinal type, and the program was compiled with range checking on (`{$R+}`, then a run-time error will be generated.

Errors: Run-time error 201 is generated when the result is out of range.

See also: Ord (146), Pred (148), High (135), Low (141)

for an example, see Ord (146)

Ptr

Declaration: `Function Ptr (Sel,Off : Longint) : Pointer;`

Description: `Ptr` returns a pointer, pointing to the address specified by segment `Sel` and offset `Off`.

Remark:

1. In the 32-bit flat-memory model supported by Free Pascal, this function is obsolete.

2.The returned address is simply the offset. If you recompile the RTL with `-dDoMapping` defined, then the compiler returns the following : `ptr := pointer($e0000000+sel shl 4+off)` under DOS, or `ptr := pointer(sel shl 4+off)` on other OSes.

Errors: None.

See also: Addr (115)

Listing: refex/ex59.pp

```
Program Example59;

{ Program to demonstrate the Ptr function. }

Var P : ^ String;
    S : String;

begin
  S:='Hello, World !';
  P:=Ptr(Seg(S),Longint(Ofs(S)));
  {P now points to S !}
  Writeln (P^);
end.
```

Random

Declaration: `Function Random [(L : Longint)] : Longint or Real;`

Description: `Random` returns a random number larger or equal to `0` and strictly less than `L`. If the argument `L` is omitted, a Real number between 0 and 1 is returned. (0 included, 1 excluded)

Errors: None.

See also: Randomize (150)

Listing: refex/ex49.pp

```
Program Example49;

{ Program to demonstrate the Random and Randomize functions. }

Var I,Count,guess : Longint;
    R : Real;

begin
  Randomize; { This way we generate a new sequence every time
               the program is run}
  Count:=0;
  For i:=1 to 1000 do
    If Random>0.5 then inc(Count);
  Writeln ('Generated ',Count,' numbers > 0.5');
  Writeln ('out of 1000 generated numbers.');
  count:=0;
  For i:=1 to 5 do
    begin
    write ('Guess a number between 1 and 5 : ');
    readln(Guess);
```

```
    If Guess=Random(5)+1 then inc(count);
    end;
  Writeln ('You guessed ',Count,' out of 5 correct.');
end.
```

Randomize

Declaration: `Procedure Randomize ;`

Description: `Randomize` initializes the random number generator of Free Pascal, by giving a value to `Randseed`, calculated with the system clock.

Errors: None.

See also: Random (149)

For an example, see Random (149).

Read

Declaration: `Procedure Read ([Var F : Any file type], V1 [, V2, ... , Vn]);`

Description: `Read` reads one or more values from a file `F`, and stores the result in `V1`, `V2`, etc.; If no file `F` is specified, then standard input is read. If `F` is of type `Text`, then the variables `V1, V2` etc. must be of type `Char`, `Integer`, `Real`, `String` or `PChar`. If `F` is a typed file, then each of the variables must be of the type specified in the declaration of `F`. Untyped files are not allowed as an argument.

Errors: If no data is available, a run-time error is generated. This behavior can be controlled with the `{$i}` compiler switch.

See also: Readln (151), Blockread (118), Write (165), Blockwrite (119)

Listing: refex/ex50.pp

```
Program Example50;

{ Program to demonstrate the Read(Ln) function. }

Var S : String;
    C : Char;
    F : File of char;

begin
  Assign (F,'ex50.pp');
  Reset (F);
  C:='A';
  Writeln ('The characters before the first space in ex50.pp are : ');
  While not Eof(f) and (C<>' ') do
    Begin
    Read (F,C);
    Write (C);
    end;
  Writeln;
  Close (F);
  Writeln ('Type some words. An empty line ends the program.');
  repeat
```

```
    Readln (S);
  until S='';
end.
```

Readln

Declaration: `Procedure Readln [Var F : Text], V1 [, V2, ... , Vn]);`

Description: `Read` reads one or more values from a file F, and stores the result in V1, V2, etc. After that it goes to the next line in the file (defined by the `LineFeed (#10)` character). If no file F is specified, then standard input is read. The variables `V1, V2` etc. must be of type `Char`, `Integer`, `Real`, `String` or `PChar`.

Errors: If no data is available, a run-time error is generated. This behavior can be controlled with the `{$i}` compiler switch.

See also: Read (150), Blockread (118), Write (165), Blockwrite (119)

For an example, see Read (150).

Release

Declaration: `Procedure Release (Var P : pointer);`

Description: `Release` sets the top of the Heap to the location pointed to by P. All memory at a location higher than P is marked empty.

Errors: A run-time error will be generated if P points to memory outside the heap.

See also: Mark (142), Memavail (143), Maxavail (143), Getmem (134), Freemem (132) New (144), Dispose (125)

For an example, see Mark (142).

Rename

Declaration: `Procedure Rename (Var F : Any Filetype; Const S : String);`

Description: `Rename` changes the name of the assigned file F to S. F must be assigned, but not opened.

Errors: A run-time error will be generated if F isn't assigned, or doesn't exist.

See also: Erase (127)

Listing: refex/ex77.pp

```
Program Example77;

{ Program to demonstrate the Rename function. }
Var F : Text;

begin
  Assign (F,paramstr(1));
  Rename (F,paramstr(2));
end.
```

Reset

Declaration: `Procedure Reset (Var F : Any File Type[; L : Longint]);`

Description: `Reset` opens a file `F` for reading. `F` can be any file type. If `F` is an untyped or typed file, then it is opened for reading and writing. If `F` is an untyped file, the record size can be specified in the optional parameter `L`. Default a value of 128 is used.

Errors: If the file cannot be opened for reading, then a run-time error is generated. This behavior can be changed by the `{$i}` compiler switch.

See also: Rewrite (152), Assign (116), Close (121), Append (115)

Listing: refex/ex51.pp

```
Program Example51;

{ Program to demonstrate the Reset function. }

Function FileExists (Name : String) : boolean;

Var F : File;

begin
  {$i-}
  Assign (F,Name);
  Reset (F);
  {$I+}
  FileExists:=(IoResult=0) and (Name<>'');
  Close (f);
end;

begin
  If FileExists (Paramstr(1)) then
    Writeln ('File found')
  else
    Writeln ('File NOT found');
end.
```

Rewrite

Declaration: `Procedure Rewrite (Var F : Any File Type[; L : Longint]);`

Description: `Rewrite` opens a file `F` for writing. `F` can be any file type. If `F` is an untyped or typed file, then it is opened for reading and writing. If `F` is an untyped file, the record size can be specified in the optional parameter `L`. Default a value of 128 is used. if `Rewrite` finds a file with the same name as `F`, this file is truncated to length `0`. If it doesn't find such a file, a new file is created.

Contrary to Turbo Pascal, Free Pascal opens the file with mode `fmoutput`. If you want to get it in `fminout` mode, you'll need to do an extra call to Reset (152).

Errors: If the file cannot be opened for writing, then a run-time error is generated. This behavior can be changed by the `{$i}` compiler switch.

See also: Reset (152), Assign (116), Close (121), Flush (131), Append (115)

Listing: refex/ex52.pp

```
Program Example52;

{ Program to demonstrate the Rewrite function. }

Var F : File;
    I : longint;

begin
  Assign (F,'Test.dat');
  { Create the file. Recordsize is 4 }
  Rewrite (F,Sizeof(I));
  For I:=1 to 10 do
    BlockWrite (F,I,1);
  close (f);
  { F contains now a binary representation of
    10 longints going from 1 to 10 }
end.
```

Rmdir

Declaration: `Procedure Rmdir (const S : string);`

Description: `Rmdir` removes the directory S.

Errors: If S doesn't exist, or isn't empty, a run-time error is generated.

See also: Chdir (120), Mkdir (144)

Listing: refex/ex53.pp

```
Program Example53;

{ Program to demonstrate the MkDir and RmDir functions. }

Const D : String[8] = 'TEST.DIR';

Var S : String;

begin
  Writeln ('Making directory ',D);
  Mkdir (D);
  Writeln ('Changing directory to ',D);
  ChDir (D);
  GetDir (0,S);
  Writeln ('Current Directory is : ',S);
  WRiteln ('Going back');
  ChDir ('..');
  Writeln ('Removing directory ',D);
  RmDir (D);
end.
```

Round

Declaration: `Function Round (X : Real) : Longint;`

Description: `Round` rounds X to the closest integer, which may be bigger or smaller than X.

Errors: None.

See also: Frac (132), Int (138), Trunc (163)

Listing: refex/ex54.pp

```
Program Example54;

{ Program to demonstrate the Round function. }

begin
  Writeln (Round(1234.56));   { Prints 1235  }
  Writeln (Round(-1234.56));  { Prints -1235 }
  Writeln (Round(12.3456));   { Prints 12    }
  Writeln (Round(-12.3456));  { Prints -12   }
end.
```

Runerror

Declaration: `Procedure Runerror (ErrorCode : Word);`

Description: `Runerror` stops the execution of the program, and generates a run-time error `ErrorCode`.

Errors: None.

See also: Exit (128), Halt (134)

Listing: refex/ex55.pp

```
Program Example55;

{ Program to demonstrate the RunError function. }

begin
  { The program will stop end emit a run-error 106 }
  RunError (106);
end.
```

Seek

Declaration: `Procedure Seek (Var F; Count : Longint);`

Description: `Seek` sets the file-pointer for file `F` to record Nr. `Count`. The first record in a file has `Count=0`. F can be any file type, except `Text`. If `F` is an untyped file, with no record size specified in Reset (152) or Rewrite (152), 128 is assumed.

Errors: A run-time error is generated if `Count` points to a position outside the file, or the file isn't opened.

See also: Eof (126), SeekEof (155), SeekEoln (156)

Listing: refex/ex56.pp

```
Program Example56;

{ Program to demonstrate the Seek function. }
```

```
Var
  F : File;
  I,j : longint;

begin
  { Create a file and fill it with data }
  Assign (F,'test.dat');
  Rewrite(F); { Create file }
  Close(f);
  FileMode:=2;
  ReSet (F,Sizeof(i)); { Opened read/write }
  For I:=0 to 10 do
    BlockWrite (F,I,1);
  { Go Back to the begining of the file }
  Seek(F,0);
  For I:=0 to 10 do
    begin
    BlockRead (F,J,1);
    If J<>I then
      Writeln ('Error: expected ' ,i,', got ',j);
    end;
  Close (f);
end.
```

SeekEof

Declaration: `Function SeekEof [(Var F : text)] : Boolean;`

Description: `SeekEof` returns `True` is the file-pointer is at the end of the file. It ignores all whitespace. Calling this function has the effect that the file-position is advanced until the first non-whitespace character or the end-of-file marker is reached. If the end-of-file marker is reached, `True` is returned. Otherwise, False is returned. If the parameter `F` is omitted, standard `Input` is assumed.

Errors: A run-time error is generated if the file `F` isn't opened.

See also: Eof (126), SeekEoln (156), Seek (154)

Listing: refex/ex57.pp

```
Program Example57;

{ Program to demonstrate the SeekEof function. }
Var C : Char;

begin
  { this will print all characters from standard input except
    Whitespace characters. }
  While Not SeekEof do
    begin
    Read (C);
    Write (C);
    end;
end.
```

SeekEoln

Declaration: `Function SeekEoln [(Var F : text)] :  Boolean;`

Description: `SeekEoln` returns `True` is the file-pointer is at the end of the current line. It ignores all whitespace. Calling this function has the effect that the file-position is advanced until the first non-whitespace character or the end-of-line marker is reached. If the end-of-line marker is reached, `True` is returned. Otherwise, False is returned. The end-of-line marker is defined as `#10`, the LineFeed character. If the parameter `F` is omitted, standard `Input` is assumed.

Errors: A run-time error is generated if the file `F` isn't opened.

See also: Eof (126), SeekEof (155), Seek (154)

Listing: refex/ex58.pp

```
Program Example58;

{ Program to demonstrate the SeekEoln function. }
Var
  C : Char;

begin
  { This will read the first line of standard output and print
    all characters except whitespace. }
  While not SeekEoln do
    Begin
    Read (c);
    Write (c);
    end;
end.
```

Seg

Declaration: `Function Seg Var X : Longint;`

Description: `Seg` returns the segment of the address of a variable. This function is only supported for compatibility. In Free Pascal, it returns always 0, since Free Pascal is a 32 bit compiler, segments have no meaning.

Errors: None.

See also: DSeg (126), CSeg (123), Ofs (145), Ptr (148)

Listing: refex/ex60.pp

```
Program Example60;

{ Program to demonstrate the Seg function. }
Var
  W : Word;

begin
  W:=Seg(W);  { W contains its own Segment}
end.
```

SetMemoryManager

Declaration: `procedure SetMemoryManager(const MemMgr: TMemoryManager);`

Description: `SetMemoryManager` sets the current memory manager record to `MemMgr`.

Errors: None.

See also: GetMemoryManager (134), IsMemoryManagerSet (138)

For an example, see Programmers' guide.

SetJmp

Declaration: `Function SetJmp (Var Env : Jmp_Buf) : Longint;`

Description: `SetJmp` fills `env` with the necessary data for a jump back to the point where it was called. It returns zero if called in this way. If the function returns nonzero, then it means that a call to LongJmp (141) with `env` as an argument was made somewhere in the program.

Errors: None.

See also: LongJmp (141)

Listing: refex/ex79.pp

```
program example79;

{ Program to demonstrate the setjmp, longjmp functions }

procedure dojmp(var env : jmp_buf; value : longint);

begin
  value:=2;
  Writeln ('Going to jump !');
  { This will return to the setjmp call,
    and return value instead of 0 }
  longjmp(env,value);
end;

var env : jmp_buf;

begin
  if setjmp(env)=0 then
    begin
    writeln ('Passed first time.');
    dojmp(env,2);
    end
  else
    writeln ('Passed second time.');
end.
```

SetLength

Declaration: `Procedure SetLength(var S : String; Len : Longint);`

Description: SetLength sets the length of the string S to Len. S can be an ansistring or a short string. For ShortStrings, Len can maximally be 255. For AnsiStrings it can have any value. For AnsiString strings, SetLength *must* be used to set the length of the string.

Errors: None.

See also: Length (140)

Listing: refex/ex85.pp

```
Program Example85;

{ Program to demonstrate the SetLength function. }

Var S : String;

begin
  FillChar(S[1],100,#32);
  Setlength(S,100);
  Writeln ('"',S,'"');
end.
```

SetTextBuf

Declaration: Procedure SetTextBuf (Var f : Text; Var Buf[; Size : Word]);

Description: SetTextBuf assigns an I/O buffer to a text file. The new buffer is located at Buf and is Size bytes long. If Size is omitted, then SizeOf(Buf) is assumed. The standard buffer of any text file is 128 bytes long. For heavy I/0 operations this may prove too slow. The SetTextBuf procedure allows you to set a bigger buffer for your application, thus reducing the number of system calls, and thus reducing the load on the system resources. The maximum size of the newly assigned buffer is 65355 bytes.

Remark:

- Never assign a new buffer to an opened file. You can assign a new buffer immediately after a call to Rewrite (152), Reset (152) or Append, but not after you read from/wrote to the file. This may cause loss of data. If you still want to assign a new buffer after read/write operations have been performed, flush the file first. This will ensure that the current buffer is emptied.
- Take care that the buffer you assign is always valid. If you assign a local variable as a buffer, then after your program exits the local program block, the buffer will no longer be valid, and stack problems may occur.

Errors: No checking on Size is done.

See also: Assign (116), Reset (152), Rewrite (152), Append (115)

Listing: refex/ex61.pp

```
Program Example61;

{ Program to demonstrate the SetTextBuf function. }

Var
  Fin,Fout : Text;
  Ch : Char;
```

```
  Bufin , Bufout : Array [1..10000] of byte ;

begin
  Assign ( Fin , paramstr (1));
  Reset ( Fin );
  Assign ( Fout , paramstr (2));
  Rewrite ( Fout );
  { This is harmless before IO has begun }
  { Try this program again on a big file ,
    after commenting out the following 2
    lines and recompiling it . }
  SetTextBuf ( Fin , Bufin );
  SetTextBuf ( Fout , Bufout );
  While not eof( Fin ) do
    begin
    Read ( Fin , ch );
    write ( Fout , ch );
    end;
  Close ( Fin );
  Close ( Fout );
end.
```

Sin

Declaration: `Function Sin (X : Real) : Real;`

Description: `Sin` returns the sine of its argument X, where X is an angle in radians.

If the absolute value of the argument is larger than `2^63`, then the result is undefined.

Errors: None.

See also: Cos (123), Pi (147), Exp (129), Ln (140)

Listing: refex/ex62.pp

```
Program Example62 ;

{ Program to demonstrate the Sin function . }

begin
  Writeln ( Sin( Pi ):0:1);     { Prints 0.0 }
  Writeln ( Sin( Pi /2):0:1); { Prints 1.0 }
end.
```

SizeOf

Declaration: `Function SizeOf (X : Any Type) : Longint;`

Description: `SizeOf` returns the size, in bytes, of any variable or type-identifier.

Remark: This isn't really a RTL function. It's result is calculated at compile-time, and hard-coded in your executable.

Errors: None.

See also: Addr (115)

Listing: refex/ex63.pp

```
Program Example63;

{ Program to demonstrate the SizeOf function. }
Var
  I : Longint;
  S : String [10];

begin
  Writeln (SizeOf(I));   { Prints 4  }
  Writeln (SizeOf(S));   { Prints 11 }
end.
```

Sptr

Declaration: `Function Sptr : Pointer;`

Description: `Sptr` returns the current stack pointer.

Errors: None.

See also: SSeg (161)

Listing: refex/ex64.pp

```
Program Example64;

{ Program to demonstrate the SPtr function. }
Var
  P : Longint;

begin
  P:=Sptr; { P Contains now the current stack position. }
end.
```

Sqr

Declaration: `Function Sqr (X : Real) : Real;`

Description: `Sqr` returns the square of its argument X.

Errors: None.

See also: Sqrt (161), Ln (140), Exp (129)

Listing: refex/ex65.pp

```
Program Example65;

{ Program to demonstrate the Sqr function. }
Var i : Integer;

begin
  For i:=1 to 10 do
    writeln (Sqr(i):3);
end.
```

Sqrt

Declaration: `Function Sqrt (X : Real) : Real;`

Description: `Sqrt` returns the square root of its argument X, which must be positive.

Errors: If X is negative, then a run-time error is generated.

See also: Sqr (160), Ln (140), Exp (129)

Listing: refex/ex66.pp

```
Program Example66;

{ Program to demonstrate the Sqrt function. }

begin
  Writeln (Sqrt(4):0:3); { Prints 2.000 }
  Writeln (Sqrt(2):0:3); { Prints 1.414 }
end.
```

SSeg

Declaration: `Function SSeg : Longint;`

Description: `SSeg` returns the Stack Segment. This function is only supported for compatibility reasons, as `Sptr` returns the correct contents of the stackpointer.

Errors: None.

See also: Sptr (160)

Listing: refex/ex67.pp

```
Program Example67;

{ Program to demonstrate the SSeg function. }
Var W : Longint;

begin
  W:=SSeg;
end.
```

Str

Declaration: `Procedure Str (Var X[:NumPlaces[:Decimals]]; Var S : String);`

Description: `Str` returns a string which represents the value of X. X can be any numerical type. The optional `NumPLaces` and `Decimals` specifiers control the formatting of the string.

Errors: None.

See also: Val (165)

Listing: refex/ex68.pp

```
Program Example68;

{ Program to demonstrate the Str function. }
Var S : String;

Function IntToStr (I : Longint) : String;

Var S : String;

begin
 Str (I,S);
 IntToStr:=S;
end;

begin
  S:='*'+IntToStr(-233)+'*';
  Writeln (S);
end.
```

StringOfChar

Declaration: `Function StringOfChar(c : char;l : longint) : AnsiString;`

Description: `StringOfChar` creates a new Ansistring of length `l` and fills it with the character `c`.

It is equivalent to the following calls:

```
SetLength(StringOfChar,l);
FillChar(Pointer(StringOfChar)^,Length(StringOfChar),c);
```

Errors: None.

See also: SetLength (157)

Listing: refex/ex97.pp

```
Program Example97;

{$H+}

{ Program to demonstrate the StringOfChar function. }

Var S : String;

begin
  S:=StringOfChar(' ',40)+'Aligned at column 41.';
  Writeln(s);
end.
```

Succ

Declaration: `Function Succ (X : Any ordinal type) : Same type;`

Description: `Succ` returns the element that succeeds the element that was passed to it. If it is applied to the last value of the ordinal type, and the program was compiled with range checking on (`{$R+}`), then a run-time error will be generated.

Errors: Run-time error 201 is generated when the result is out of range.

See also: Ord (146), Pred (148), High (135), Low (141)

for an example, see Ord (146).

Swap

Declaration: `Function Swap (X) : Type of X;`

Description: `Swap` swaps the high and low order bytes of X if X is of type `Word` or `Integer`, or swaps the high and low order words of X if X is of type `Longint` or `Cardinal`. The return type is the type of X

Errors: None.

See also: Lo (140), Hi (135)

Listing: refex/ex69.pp

```
Program Example69;

{ Program to demonstrate the Swap function. }
Var W : Word;
    L : Longint;

begin
  W:=$1234;
  W:=Swap(W);
  if W<>$3412 then
    writeln ('Error when swapping word !');
  L:=$12345678;
  L:=Swap(L);
  if L<>$56781234 then
    writeln ('Error when swapping Longint !');
end.
```

Trunc

Declaration: `Function Trunc (X : Real) :  Longint;`

Description: `Trunc` returns the integer part of X, which is always smaller than (or equal to) X in absolute value.

Errors: None.

See also: Frac (132), Int (138), Round (153)

Listing: refex/ex70.pp

```
Program Example70;

{ Program to demonstrate the Trunc function. }

begin
  Writeln (Trunc(123.456));   { Prints 123  }
  Writeln (Trunc(-123.456));  { Prints -123 }
  Writeln (Trunc(12.3456));   { Prints 12   }
  Writeln (Trunc(-12.3456));  { Prints -12  }
end.
```

Truncate

Declaration: `Procedure Truncate (Var F : file);`

Description: `Truncate` truncates the (opened) file F at the current file position.

Errors: Errors are reported by IOresult.

See also: Append (115), Filepos (129), Seek (154)

Listing: refex/ex71.pp

```
Program Example71;

{ Program to demonstrate the Truncate function. }

Var F : File of longint;
    I,L : Longint;

begin
  Assign (F,'test.dat');
  Rewrite (F);
  For I:=1 to 10 Do
    Write (F,I);
  Writeln ('Filesize before Truncate : ',FileSize(F));
  Close (f);
  Reset (F);
  Repeat
    Read (F,I);
  Until i=5;
  Truncate (F);
  Writeln ('Filesize after Truncate  : ',Filesize(F));
  Close (f);
end.
```

Upcase

Declaration: `Function Upcase (C : Char or string) : Char or String;`

Description: `Upcase` returns the uppercase version of its argument C. If its argument is a string, then the complete string is converted to uppercase. The type of the returned value is the same as the type of the argument.

Errors: None.

See also: Lowercase (142)

Listing: refex/ex72.pp

```
Program Example72;

{ Program to demonstrate the Upcase function. }

Var I : Longint;

begin
  For i:=ord('a') to ord('z') do
```

```
    write (upcase(chr(i)));
  Writeln;
  { This doesn't work in TP, but it does in Free Pascal }
  Writeln (Upcase('abcdefghijklmnopqrstuvwxyz'));
end.
```

Val

Declaration: `Procedure Val (const S : string;var V;var Code : word);`

Description: `Val` converts the value represented in the string `S` to a numerical value, and stores this value in the variable `V`, which can be of type `Longint`, `Real` and `Byte`. If the conversion isn't succesfull, then the parameter `Code` contains the index of the character in `S` which prevented the conversion. The string `S` isn't allowed to contain spaces.

Errors: If the conversion doesn't succeed, the value of `Code` indicates the position where the conversion went wrong.

See also: Str (161)

Listing: refex/ex74.pp

```
Program Example74;

{ Program to demonstrate the Val function. }
Var I, Code : Integer;

begin
  Val (ParamStr (1),I,Code);
  If Code<>0 then
    Writeln ('Error at position ',code,' : ',Paramstr(1)[Code])
  else
    Writeln ('Value : ',I);
end.
```

Write

Declaration: `Procedure Write ([Var F : Any filetype;] V1 [; V2; ... , Vn)];`

Description: `Write` writes the contents of the variables `V1`, `V2` etc. to the file `F`. `F` can be a typed file, or a `Text` file. If `F` is a typed file, then the variables `V1`, `V2` etc. must be of the same type as the type in the declaration of `F`. Untyped files are not allowed. If the parameter `F` is omitted, standard output is assumed. If `F` is of type `Text`, then the necessary conversions are done such that the output of the variables is in human-readable format. This conversion is done for all numerical types. Strings are printed exactly as they are in memory, as well as `PChar` types. The format of the numerical conversions can be influenced through the following modifiers: `OutputVariable : NumChars [: Decimals ]` This will print the value of `OutputVariable` with a minimum of `NumChars` characters, from which `Decimals` are reserved for the decimals. If the number cannot be represented with `NumChars` characters, `NumChars` will be increased, until the representation fits. If the representation requires less than `NumChars` characters then the output is filled up with spaces, to the left of the generated string, thus resulting in a right-aligned representation. If no formatting is specified, then the number is written using its natural length, with nothing in front of it if it's positive, and a minus sign if it's negative. Real numbers are, by default, written in scientific notation.

Errors: If an error occurs, a run-time error is generated. This behavior can be controlled with the `{$i}` switch.

See also: WriteLn (166), Read (150), Readln (151), Blockwrite (119)

WriteLn

Declaration: `Procedure WriteLn [([Var F : Text;] [V1 [; V2; ... , Vn)]];`

Description: `WriteLn` does the same as Write (165) for text files, and emits a Carriage Return - LineFeed character pair after that. If the parameter F is omitted, standard output is assumed. If no variables are specified, a Carriage Return - LineFeed character pair is emitted, resulting in a new line in the file F.

Remark: Under LINUX, the Carriage Return character is omitted, as customary in Unix environments.

Errors: If an error occurs, a run-time error is generated. This behavior can be controlled with the `{$i}` switch.

See also: Write (165), Read (150), Readln (151), Blockwrite (119)

Listing: refex/ex75.pp

```
Program Example75;

{ Program to demonstrate the Write(ln) function. }

Var
  F : File of Longint;
  L : Longint;

begin
  Write ('This is on the first line ! '); { No CR/LF pair! }
  Writeln ('And this too...');
  Writeln ('But this is already on the second line...');
  Assign (f,'test.dat');
  Rewrite (f);
  For L:=1 to 10 do
    write (F,L); { No writeln allowed here ! }
  Close (f);
end.
```

Chapter 14

The OBJPAS unit

The objpas unit is meant for compatibility with Object Pascal as implemented by Delphi. The unit is loaded automatically by the Free Pascal compiler whenever the `Delphi` or `objfpc` more is entered, either through the command line switches `-Sd` or `-Sh` or with the `{$MODE DELPHI}` or `{$MODE OBJFPC}` directives.

It redefines some basic pascal types, introduces some functions for compatibility with Delphi's system unit, and introduces some methods for the management of the resource string tables.

14.1 Types

The objpas unit redefines two integer types, for compatibity with Delphi:

```
type
  smallint = system.integer;
  integer  = system.longint;
```

The resource string tables can be managed with a callback function which the user must provide: `TResourceIterator`.

```
Type
   TResourceIterator =
      Function (Name,Value : AnsiString;Hash : Longint):AnsiString;
```

14.2 Functions and Procedures

AssignFile

Declaration: `Procedure AssignFile(Var f:  FileType;Name:  Character type);`

Description: `AssignFile` is completely equivalent to the system unit's Assign (116) function: It assigns `Name` to a function of any type (`FileType` can be `Text` or a typed or untyped `File` variable). `Name` can be a string, a single character or a `PChar`.

It is most likely introduced to avoid confusion between the regular Assign (116) function and the `Assign` method of `TPersistent` in the Delphi VCL.

Errors: None.

See also: CloseFile (168), Assign (116), Reset (152), Rewrite (152), Append (115)

Listing: refex/ex88.pp

```
Program Example88;

{ Program to demonstrate the AssignFile and CloseFile functions. }

{$MODE Delphi}

Var F : text;

begin
  AssignFile(F,'textfile.txt');
  Rewrite(F);
  Writeln (F,'This is a silly example of AssignFile and CloseFile.');
  CloseFile(F);
end.
```

CloseFile

Declaration: `Procedure CloseFile(Var F: FileType);`

Description: `CloseFile` flushes and closes a file `F` of any file type. `F` can be `Text` or a typed or untyped `File` variable. After a call to `CloseFile`, any attempt to write to the file `F` will result in an error.

It is most likely introduced to avoid confusion between the regular Close (121) function and the `Close` method of `TForm` in the Delphi VCL.

Errors: None.

See also: Close (121), AssignFile (167), Reset (152), Rewrite (152), Append (115)

for an example, see AssignFile (167).

Freemem

Declaration: `Procedure FreeMem(Var p:pointer[;Size:Longint]);`

Description: `FreeMem` releases the memory reserved by a call to GetMem (169). The (optional) `Size` parameter is ignored, since the object pascal version of `GetMem` stores the amount of memory that was requested.

be sure not to release memory that was not obtained with the `Getmem` call in Objpas. Normally, this should not happen, since objpas changes the default memory manager to it's own memory manager.

Errors: None.

See also: Freemem (132), GetMem (169), Getmem (134)

Listing: refex/ex89.pp

```
Program Example89;

{ Program to demonstrate the FreeMem function. }
{$Mode Delphi}
```

```
Var P : Pointer;

begin
  Writeln ('Memory before : ', Memavail);
  GetMem(P,10000);
  FreeMem(P);
  Writeln ('Memory after  : ', Memavail);
end.
```

Getmem

Declaration: `Procedure Getmem(Var P:pointer;Size:Longint);`

Description: `GetMem` reserves `Size` bytes of memory on the heap and returns a pointer to it in `P`. `Size` is stored at offset -4 of the result, and is used to release the memory again. `P` can be a typed or untyped pointer.

Be sure to release this memory with the FreeMem (168) call defined in the objpas unit.

Errors: In case no more memory is available, and no more memory could be obtained from the system a run-time error is triggered.

See also: FreeMem (168), Getmem (134).

For an example, see FreeMem (168).

GetResourceStringCurrentValue

Declaration: `Function GetResourceStringCurrentValue(TableIndex,StringIndex : Longint) : AnsiString;`

Description: `GetResourceStringCurrentValue` returns the current value of the resourcestring in table `TableIndex` with index `StringIndex`.

The current value depends on the system of internationalization that was used, and which language is selected when the program is executed.

Errors: If either `TableIndex` or `StringIndex` are out of range, then a empty string is returned.

See also: SetResourceStrings (173), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173), ResourceStringCount (173)

Listing: refex/ex90.pp

```
Program Example90;

{ Program to demonstrate the GetResourceStringCurrentValue function. }
{$Mode Delphi}

ResourceString

  First  = 'First string';
  Second = 'Second String';

Var I,J : Longint;
```

```
begin
  { Print current values of all resourcestrings }
  For I:=0 to ResourceStringTableCount-1 do
    For J:=0 to ResourceStringCount(i)-1 do
      Writeln (I,',',J,' : ',GetResourceStringCurrentValue(I,J));
end.
```

GetResourceStringDefaultValue

Declaration: `Function GetResourceStringDefaultValue(TableIndex,StringIndex : Longint) : AnsiString`

Description: `GetResourceStringDefaultValue` returns the default value of the resourcestring in table `TableIndex` with index `StringIndex`.

The default value is the value of the string that appears in the source code of the programmer, and is compiled into the program.

Errors: If either `TableIndex` or `StringIndex` are out of range, then a empty string is returned.

Errors:

See also: SetResourceStrings (173), GetResourceStringCurrentValue (169), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173), ResourceStringCount (173)

Listing: refex/ex91.pp

```
Program Example91;

{ Program to demonstrate the GetResourceStringDefaultValue function. }
{$Mode Delphi}

ResourceString

  First  = 'First string';
  Second = 'Second String';

Var I,J : Longint;

begin
  { Print default values of all resourcestrings }
  For I:=0 to ResourceStringTableCount-1 do
    For J:=0 to ResourceStringCount(i)-1 do
      Writeln (I,',',J,' : ',GetResourceStringDefaultValue(I,J));
end.
```

GetResourceStringHash

Declaration: `Function GetResourceStringHash(TableIndex,StringIndex : Longint) : Longint;`

Description: `GetResourceStringHash` returns the hash value associated with the resource string in table `TableIndex`, with index `StringIndex`.

The hash value is calculated from the default value of the resource string in a manner that gives the same result as the GNU gettext mechanism. It is stored in the resourcestring tables, so retrieval is faster than actually calculating the hash for each string.

Errors: If either `TableIndex` or `StringIndex` is zero, 0 is returned.

See also: Hash (171) SetResourceStrings (173), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173), ResourceStringCount (173)

For an example, see Hash (171).

GetResourceStringName

Declaration: `Function GetResourceStringName(TableIndex,StringIndex : Longint) : Ansistring;`

Description: `GetResourceStringName` returns the name of the resourcestring in table `TableIndex` with index `StringIndex`. The name of the string is always the unit name in which the string was declared, followed by a period and the name of the constant, all in lowercase.

If a unit MyUnit declares a resourcestring `MyTitle` then the name returned will be `myunit.mytitle`. A resourcestring in the program file will have the name of the program prepended.

The name returned by this function is also the name that is stored in the resourcestring file generated by the compiler.

Strictly speaking, this information isn't necessary for the functioning of the program, it is provided only as a means to easier translation of strings.

Errors: If either `TableIndex` or `StringIndex` is zero, an empty string is returned.

See also: SetResourceStrings (173), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173), ResourceStringCount (173)

Listing: refex/ex92.pp

```
Program Example92;

{ Program to demonstrate the GetResourceStringName function. }
{$Mode Delphi}

ResourceString

  First  = 'First string';
  Second = 'Second String';

Var I,J : Longint;

begin
  { Print names of all resourcestrings }
  For I:=0 to ResourceStringTableCount-1 do
    For J:=0 to ResourceStringCount(i)-1 do
      Writeln (I,',',J,' : ',GetResourceStringName(I,J));
end.
```

Hash

Declaration: `Function Hash(S : AnsiString) : longint;`

Description: `Hash` calculates the hash value of the string S in a manner that is compatible with the GNU gettext hash value for the string. It is the same value that is stored in the Resource string tables, and which can be retrieved with the GetResourceStringHash (170) function call.

Errors: None. In case the calculated hash value should be 0, the returned result will be -1.

See also: GetResourceStringHash (170),

Listing: refex/ex93.pp

```
Program Example93;

{ Program to demonstrate the Hash function. }
{$Mode Delphi}

ResourceString

  First  = 'First string';
  Second = 'Second String';

Var I,J : Longint;

begin
  For I:=0 to ResourceStringTableCount-1 do
    For J:=0 to ResourceStringCount(i)-1 do
      If Hash(GetResourceStringDefaultValue(I,J))
        <>GetResourceStringHash(I,J) then
        Writeln ('Hash mismatch at ',I,',',J)
      else
        Writeln ('Hash (',I,',',J,') matches.');
end.
```

Paramstr

Declaration: `Function ParamStr(Param : Integer) : Ansistring;`

Description: `ParamStr` returns the `Param`-th command-line parameter as an AnsiString. The system unit Paramstr (147) function limits the result to 255 characters.

The zeroeth command-line parameter contains the path of the executable, except on LINUX, where it is the command as typed on the command-line.

Errors: In case `Param` is an invalid value, an empty string is returned.

See also: Paramstr (147)

For an example, see Paramstr (147).

ResetResourceTables

Declaration: `Procedure ResetResourceTables;`

Description: `ResetResourceTables` resets all resource strings to their default (i.e. as in the source code) values.

Normally, this should never be called from a user's program. It is called in the initialization code of the objpas unit. However, if the resourcetables get messed up for some reason, this procedure will fix them again.

Errors: None.

See also: SetResourceStrings (173), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173), ResourceStringCount (173)

ResourceStringCount

Declaration: `Function ResourceStringCount(TableIndex : longint) : longint;`

Description: `ResourceStringCount` returns the number of resourcestrings in the table with index `TableIndex`. The strings in a particular table are numbered from `0` to `ResourceStringCount-1`, i.e. they're zero based.

Errors: If an invalid `TableIndex` is given, `-1` is returned.

See also: SetResourceStrings (173), GetResourceStringCurrentValue (169), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173),

For an example, see GetResourceStringDefaultValue (170)

ResourceStringTableCount

Declaration: `Function ResourceStringTableCount : Longint;`

Description: `ResourceStringTableCount` returns the number of resource string tables; this may be zero if no resource strings are used in a program.

The tables are numbered from 0 to `ResourceStringTableCount-1`, i.e. they're zero based.

Errors:

See also: SetResourceStrings (173), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringCount (173)

For an example, see GetResourceStringDefaultValue (170)

SetResourceStrings

Declaration: `TResourceIterator = Function (Name,Value : AnsiString;Hash : Longint):AnsiString;`

`Procedure SetResourceStrings (SetFunction : TResourceIterator);`

Description: `SetResourceStrings` calls `SetFunction` for all resourcestrings in the resourcestring tables and sets the resourcestring's current value to the value returned by `SetFunction`.

The `Name`,`Value` and `Hash` parameters passed to the iterator function are the values stored in the tables.

Errors: None.

See also: SetResourceStrings (173), GetResourceStringCurrentValue (169), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173), ResourceStringCount (173)

Listing: refex/ex95.pp

```
Program Example95;

{ Program to demonstrate the SetResourceStrings function. }
{$Mode objfpc}

ResourceString

  First  = 'First string';
  Second = 'Second String';

Var I,J : Longint;
    S : AnsiString;

Function Translate (Name,Value : AnsiString; Hash : longint): AnsiString;

begin
  Writeln ('Translate (',Name,') => ',Value);
  Write   ('->');
  Readln  (Result);
end;

begin
  SetResourceStrings(@Translate);
  Writeln ('Translated strings : ');
  For I:=0 to ResourceStringTableCount-1 do
    For J:=0 to ResourceStringCount(i)-1 do
      begin
      Writeln (GetResourceStringDefaultValue(I,J));
      Writeln ('Translates to : ');
      Writeln (GetResourceStringCurrentValue(I,J));
      end;
end.
```

SetResourceStringValue

Declaration: `Function SetResourceStringValue(TableIndex,StringIndex : longint; Value : Ansistring) : Boolean;`

Description: `SetResourceStringValue` assigns `Value` to the resource string in table `TableIndex` with index `StringIndex`.

Errors:

See also: SetResourceStrings (173), GetResourceStringCurrentValue (169), GetResourceStringDefaultValue (170), GetResourceStringHash (170), GetResourceStringName (171), ResourceStringTableCount (173), ResourceStringCount (173)

Listing: refex/ex94.pp

```
Program Example94;

{ Program to demonstrate the SetResourceStringValue function. }
{$Mode Delphi}

ResourceString
```

```
  First  = 'First string';
  Second = 'Second String';

Var I,J : Longint;
    S : AnsiString;

begin
  { Print current values of all resourcestrings }
  For I:=0 to ResourceStringTableCount-1 do
    For J:=0 to ResourceStringCount(i)-1 do
      begin
      Writeln ('Translate => ',GetResourceStringDefaultValue(I,J));
      Write   ('->');
      Readln(S);
      SetResourceStringValue(I,J,S);
      end;
  Writeln ('Translated strings : ');
  For I:=0 to ResourceStringTableCount-1 do
    For J:=0 to ResourceStringCount(i)-1 do
      begin
      Writeln (GetResourceStringDefaultValue(I,J));
      Writeln ('Translates to : ');
      Writeln (GetResourceStringCurrentValue(I,J));
      end;
end.
```

Index