Chapter 1

Pascal Tokens

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

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

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

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

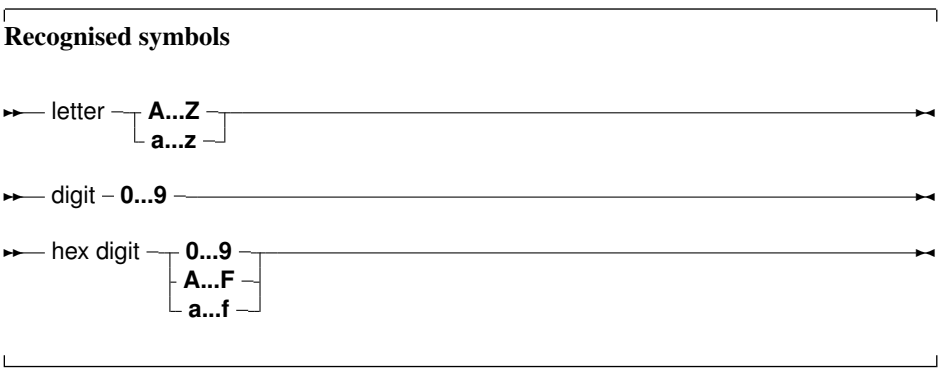
separators This is usually white-space.

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

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

1.1 Symbols

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



The following characters have a special meaning:

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

and the following character pairs too:

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

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

1.2 Comments

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

The following piece of code demonstrates a comment:

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

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

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

The comment can also span multiple lines:

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

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

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

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

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

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

The following are valid ways of nesting comments:

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

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

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

and

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

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

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

1.3 Reserved words

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

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

1.3.1 Turbo Pascal reserved words

The following keywords exist in Turbo Pascal mode

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

1.3.2 Free Pascal reserved words

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

```
dispose          false            true
exit             new
```

1.3.3 Object Pascal reserved words

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

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

1.3.4 Modifiers

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

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

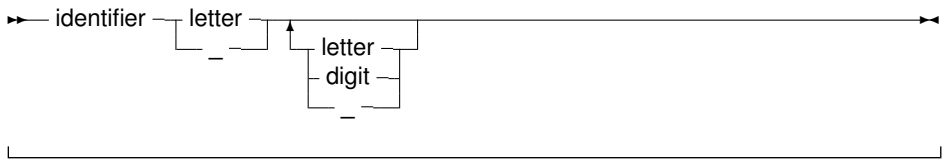
Remark: Predefined types such as `Byte`, `Boolean` and constants such as `maxint` are *not* reserved words. They are identifiers, declared in the system unit. This means that these types can be redefined in other units. The programmer is however not encouraged to do this, as it will cause a lot of confusion.

1.4 Identifiers

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

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

Identifiers



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

```
myprocedure;
```

and

```
MyProcedure;
```

refer to the same procedure.

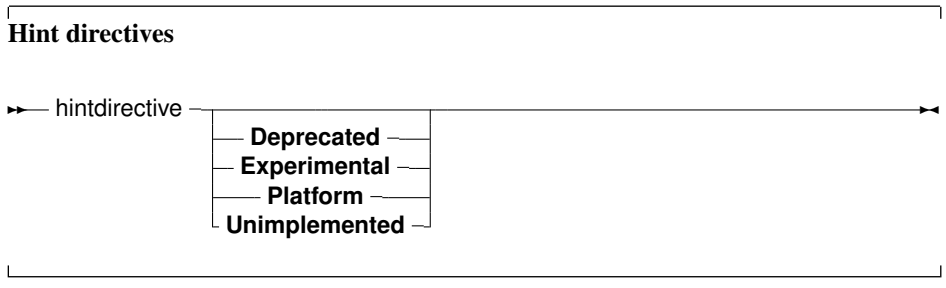
Remark: As of version 2.5.1 it is possible to specify a reserved word as an identifier by prepending it with an ampersand (&). This means that the following is possible:

```
program testdo;

procedure &do;

begin
end;

begin
  &do;
end.
```

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

1.5 Hint directives

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



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

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

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

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

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

The following are examples:

```
Const
  AConst = 12 deprecated;

var
  p : integer platform;

Function Something : Integer; experimental;

begin
  Something:=P+AConst;
end;

begin
  Something;
end.
```

This would result in the following output:

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

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

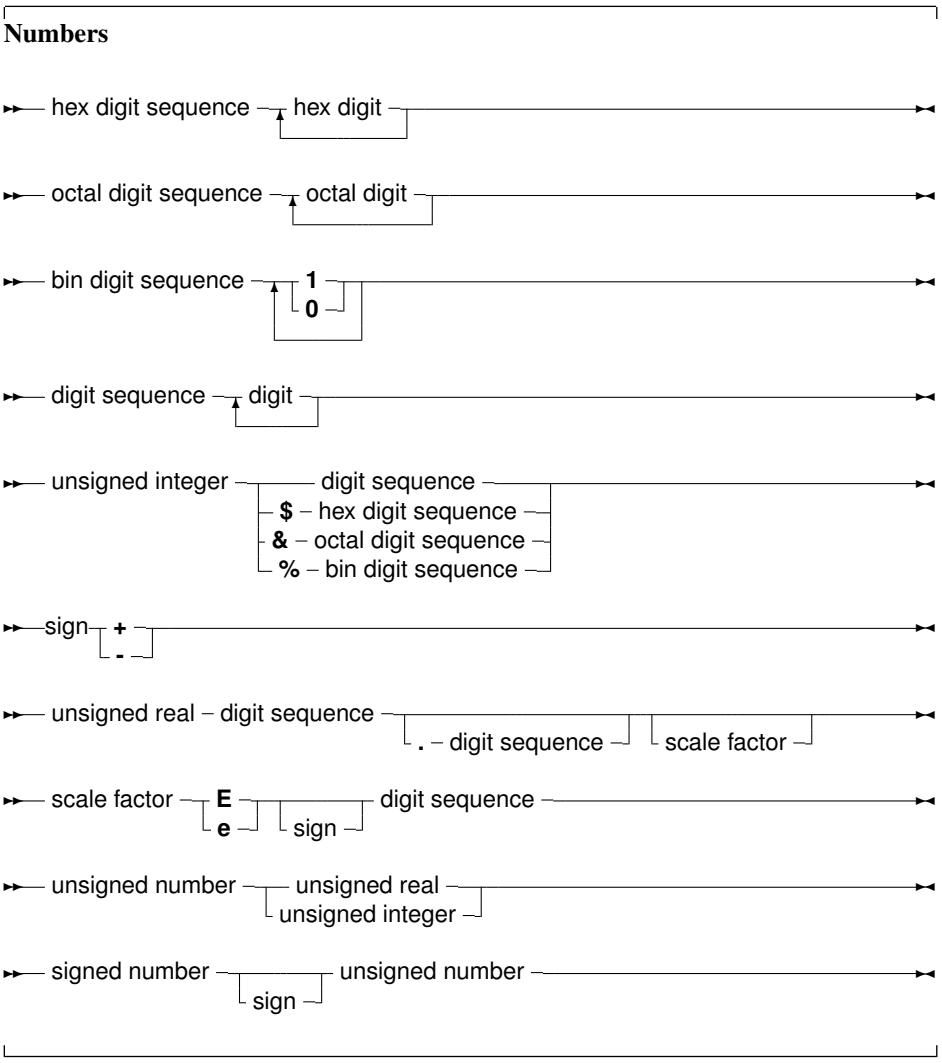
1.6 Numbers

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

For integer type constants, Free Pascal supports 4 formats:

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

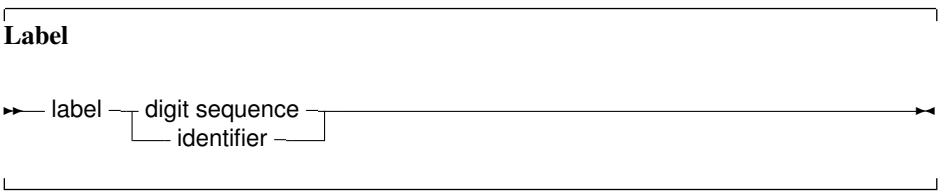
The following diagrams show the syntax for numbers.



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

1.7 Labels

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

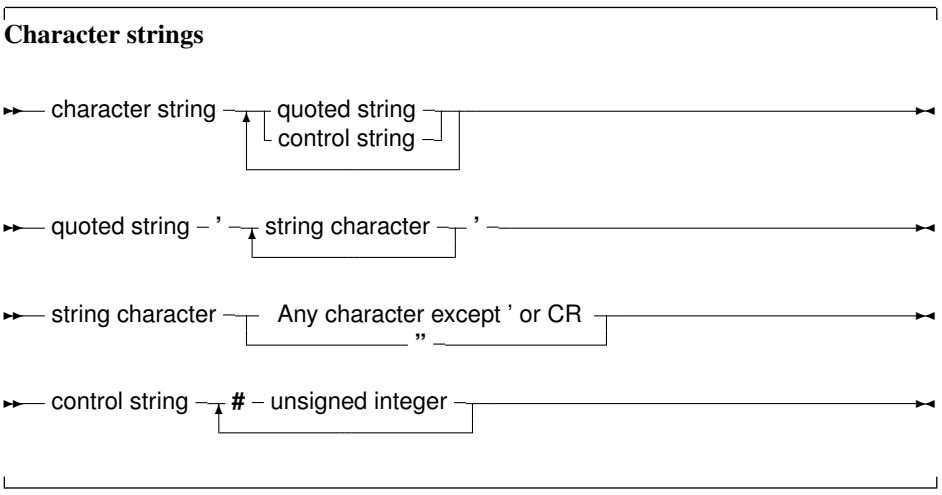


Remark: The `-Sg` or `-Mtp` switches must be specified before labels can be used. By default, Free Pascal doesn't support `label` and `goto` statements. The `{$GOTO ON}` directive can also be used to allow use of labels and the goto statement.

1.8 Character strings

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

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



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

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

The following are valid string constants:

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

The following is an invalid string:

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

The above string must be typed as:

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

or

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

on unices (including Mac OS X), and as

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

on a classic Mac-like operating system.

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

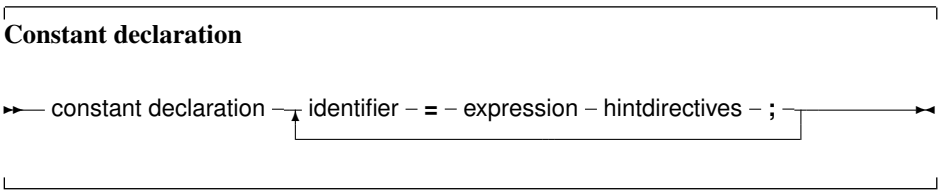
Chapter 2

Constants

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

2.1 Ordinary constants

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



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

Only constants of the following types can be declared: `Ordinal types`, `Real types`, `Char`, and `String`. The following are all valid constant declarations:

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

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

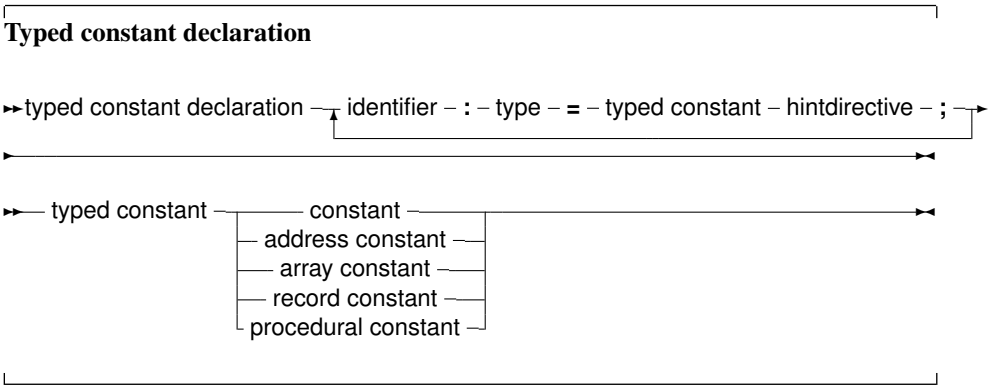
```
  s := 'some other string';
```

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

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

2.2 Typed constants

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



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

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

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

2.3 Resource strings

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

The following is an example of a resourcestring definition:

```
Resourcestring

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

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

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

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

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

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

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

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

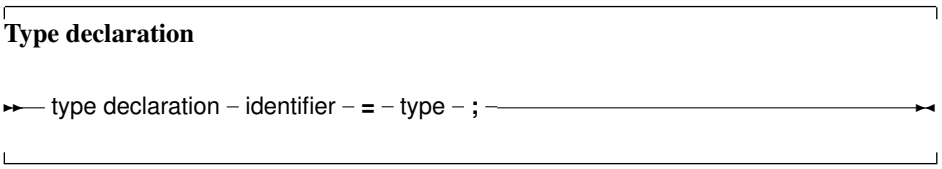
```
resourcestring
  Yes = 'Yes.';
  No = 'No.';

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

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

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

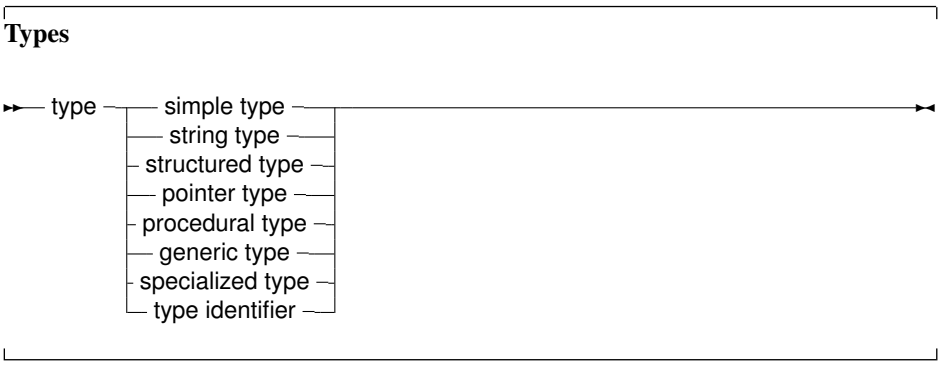
Chapter 3

Types

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



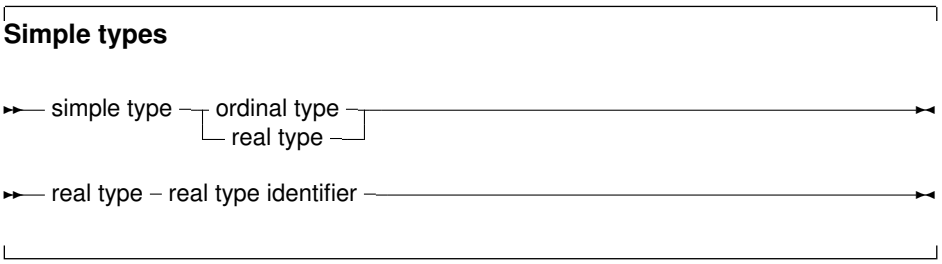
There are 7 major type classes :



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

3.1 Base types

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



3.1.1 Ordinal types

With the exception of `int64`, `qword` and Real types, all base types are ordinal types. Ordinal types have the following characteristics:

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

Integers

A list of pre-defined integer types is presented in table (3.1).

Table 3.1: Predefined integer types

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

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

Table 3.2: Predefined integer types

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

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

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

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

Boolean types

Free Pascal supports the `Boolean` type, with its two pre-defined possible values `True` and `False`. These are the only two values that can be assigned to a `Boolean` type. Of course, any expression that resolves to a `boolean` value, can also be assigned to a boolean type. Free Pascal also supports the `ByteBool`, `WordBool` and `LongBool` types. These are of

Table 3.3: Boolean types

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

type `Byte`, `Word` or `Longint`, but are assignment compatible with a `Boolean`: the value `False` is equivalent to 0 (zero) and any nonzero value is considered `True` when converting to a boolean value. A boolean value of `True` is converted to -1 in case it is assigned to a variable of type `LongBool`.

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

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

Boolean expressions are also used in conditions.

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

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

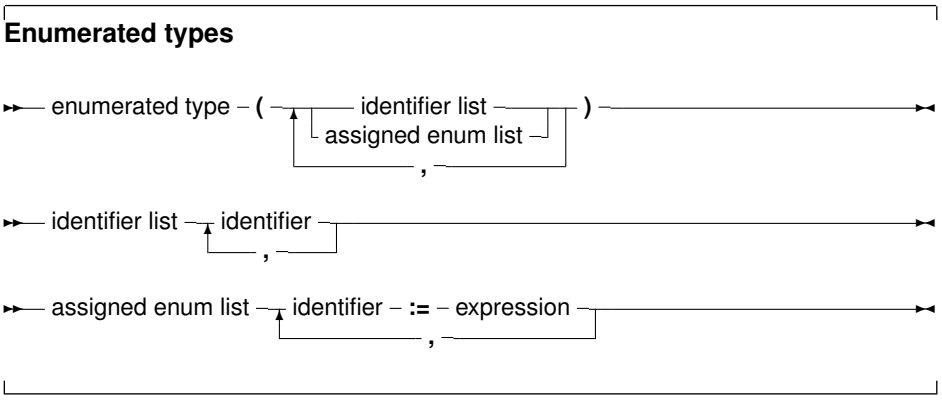
```
...
B := False;
A := B and Func;
```

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

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

Enumeration types

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



(see chapter 9, page 94 for how to use expressions) When using assigned enumerated types, the assigned elements must be in ascending numerical order in the list, or the compiler will complain. The expressions used in assigned enumerated elements must be known at compile time. So the following is a correct enumerated type declaration:

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

A C-style enumeration type looks as follows:

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

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

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

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

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

1. The `Pred` and `Succ` functions cannot be used on this kind of enumeration types. Trying to do this anyhow will result in a compiler error.
2. Enumeration types are stored using a default, independent of the actual number of values: the compiler does not try to optimize for space. This behaviour can be changed with the `{$PACKENUM n}` compiler directive, which tells the compiler the minimal number of bytes to be used for enumeration types. For instance

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

   will, when run, print the following:

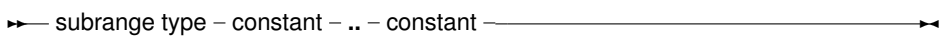
   ```
   Small enum : 1
   Large enum : 4
   ```

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

Subrange types

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

Subrange types



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

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

Subrange types of enumeration types can also be defined:

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

3.1.2 Real types

Free Pascal uses the math coprocessor (or emulation) for all its floating-point calculations. The Real native type is processor dependent, but it is either Single or Double. Only the IEEE floating point types are supported, and these depend on the target processor and emulation options. The true Turbo Pascal compatible types are listed in table (3.4). The `Comp` type is,

Table 3.4: Supported Real types

Type	Range	Significant digits	Size
Real	platform dependant	???	4 or 8
Single	1.5E-45 .. 3.4E38	7-8	4
Double	5.0E-324 .. 1.7E308	15-16	8
Extended	1.9E-4932 .. 1.1E4932	19-20	10
Comp	-2E64+1 .. 2E63-1	19-20	8
Currency	-922337203685477.5808 .. 922337203685477.5807	19-20	8

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

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

3.2 Character types

3.2.1 Char

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

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

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

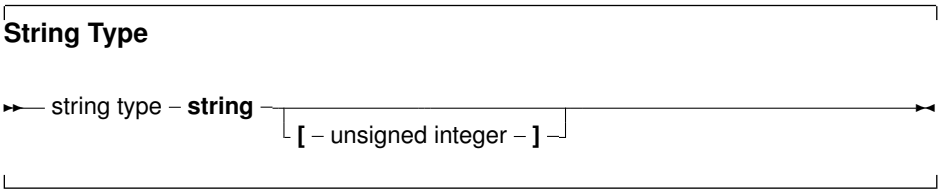
Also, the caret character (`^`) can be used in combination with a letter to specify a character with ASCII value less than 27. Thus `^G` equals `#7` - G is the seventh letter in the alphabet. The compiler is rather sloppy about the characters it allows after the caret, but in general one should assume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

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

3.2.2 Strings

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

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



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

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

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

3.2.3 Short strings

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

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

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

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

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

```
{$H-}

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

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

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

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

For short strings, the length is stored in the character at index 0. Old Turbo Pascal code relies on this, and it is implemented similarly in Free Pascal. Despite this, to write portable code, it is best to set the length of a shortstring with the `SetLength` call, and to retrieve it with the `Length` call. These functions will always work, whatever the internal representation of the shortstrings or other strings in use: this allows easy switching between the various string types.

3.2.4 Ansistrings

Ansistrings are strings that have no length limit. They are reference counted and are guaranteed to be null terminated. Internally, an ansistring is treated as a pointer: the actual content of the string is stored on the heap, as much memory as needed to store the string content is allocated.

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

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

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

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

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

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

```
S2:=S1;
```

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

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

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

This does introduce an overhead. For instance, declaring

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

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

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

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

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

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

```
  S:=T;
  R:=T; // Reference count of T is at least 3
  UniqueString(T);
  // Reference count of T is quaranteed 1
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.5 UnicodeStrings

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

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

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

3.2.6 WideStrings

Widestrings (used to represent unicode character strings in COM applications) are implemented in much the same way as unicodestrings. Unlike the latter, they are *not* reference counted, and on Windows, they are allocated with a special windows function which allows them to be used for OLE automation. This means they are implemented as null-terminated arrays of `WideChars` instead of regular `Chars`. A `WideChar` is a two-byte character (an element of a DBCS: Double Byte Character Set). Mostly the same rules apply for `WideStrings` as for `AnsiStrings`. Similar to unicodestrings, the compiler transparantly converts WideStrings to AnsiStrings and vice versa.

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

3.2.7 Constant strings

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

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

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

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

3.2.8 PChar - Null terminated strings

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

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

Results in the same as

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

These examples also show that it is possible to write *the contents* of the string to a file of type Text. The strings unit contains procedures and functions that manipulate the PChar type as in the standard C library. Since it is equivalent to a pointer to a type Char variable, it is also possible to do the following:

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

This will have the same result as the previous two examples. Null-terminated strings cannot be added as normal Pascal strings. If two PChar strings must be concatenated; the functions from the unit strings must be used.

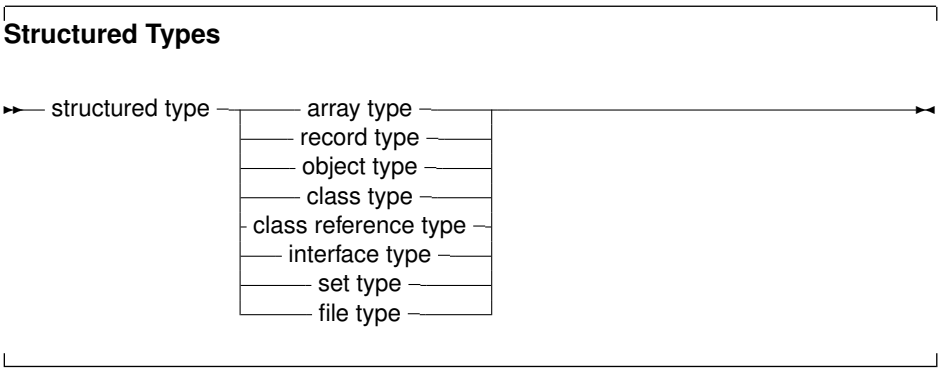
However, it is possible to do some pointer arithmetic. The operators + and - can be used to do operations on PChar pointers. In table (3.5), P and Q are of type PChar, and I is of type Longint.

3.3 Structured Types

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

Table 3.5: `PChar` pointer arithmetic

Operation	Result
`P + I`	Adds `I` to the address pointed to by `P`.
`I + P`	Adds `I` to the address pointed to by `P`.
`P - I`	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`)



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

Packed structured types

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

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

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

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

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

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

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

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

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

There are some more restrictions to elements of bitpacked structures:

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

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

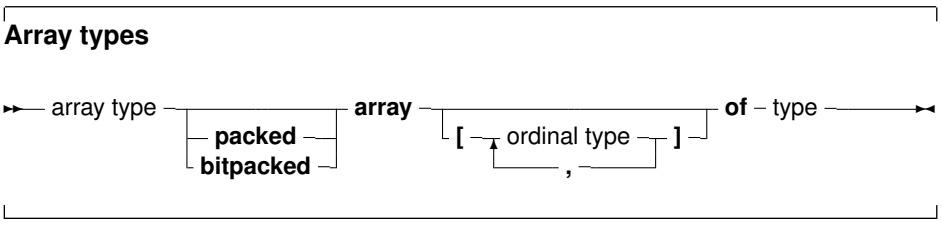
The size of bitpacked records and arrays is limited:

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

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

3.3.1 Arrays

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



Static arrays

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

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

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

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

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

is equivalent to the declaration:

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

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

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

```
program testarray1;

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

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

```
    end;
end.
```

The output of this program will be 2 identical matrices.

Dynamic arrays

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

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

When declaring a variable of a dynamic array type, the initial length of the array is zero. The actual length of the array must be set with the standard `SetLength` function, which will allocate the necessary memory to contain the array elements on the heap. The following example will set the length to 1000:

```
Var
  A : TByteArray;

begin
  SetLength(A,1000);
```

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

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

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

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

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

```
Var
  A,B : TByteArray;

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

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

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

```
program testarray1;

Type
  TA = Array of array of Integer;

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

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

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

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

```
program testarray3;

Type
  TA = array of Integer;

var
  A,B : TA;
  I : Integer;

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

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

The `Copy` function will copy 6 elements of the array to a new array. Starting at the element at index 3 (i.e. the fourth element) of the array.

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

Packing and unpacking an array

Arrays can be packed and bitpacked. 2 array types which have the same index type and element type, but which are differently packed are not assignment compatible.

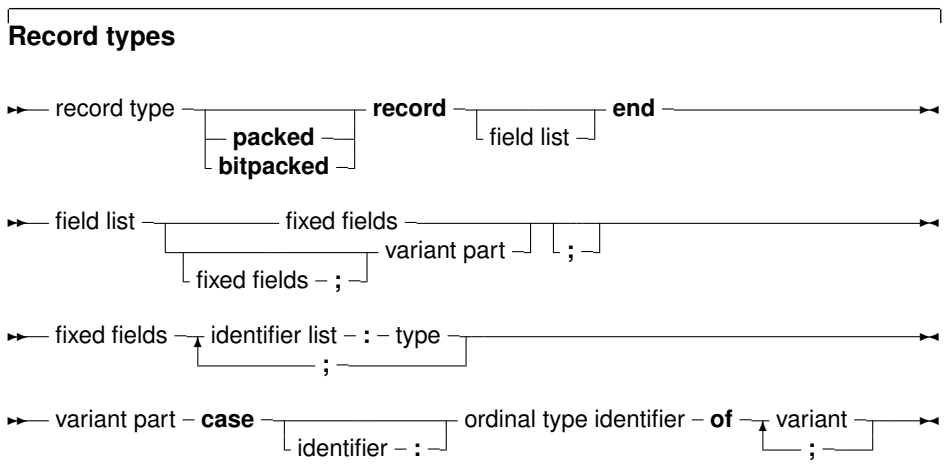
However, it is possible to convert a normal array to a bitpacked array with the `pack` routine. The reverse operation is possible as well; a bitpacked array can be converted to a normally packed array using the `unpack` routine, as in the following example:

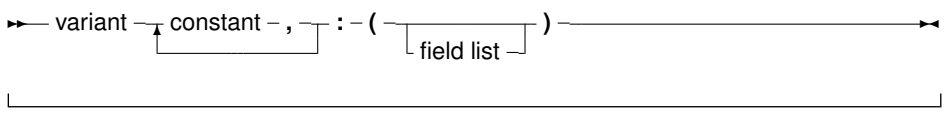
```
Var
  foo : array [ 'a'..'f' ] of Boolean
    = ( false, false, true, false, false, false );
  bar : packed array [ 42..47 ] of Boolean;
  baz : array [ '0'..'5' ] of Boolean;

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

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

3.3.2 Record types

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





So the following are valid record type declarations:

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

The variant part must be last in the record. The optional identifier in the case statement serves to access the tag field value, which otherwise would be invisible to the programmer. It can be used to see which variant is active at a certain time[1]. In effect, it introduces a new field in the record.

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

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

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

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

If a typed file with records, produced by a Turbo Pascal program, must be read, then chances are that attempting to read that file correctly will fail. The reason for this is that by default, elements of a record are aligned at 2-byte boundaries, for performance reasons.

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

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

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

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

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

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

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

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

     {$PackRecords 8}
     Trec8 = Record
       A : Byte;
```

```
      B : Array[1..7] of byte;
      C : byte;
      end;
Var rec1 : Trec1;
    rec2 : Trec2;
    rec3 : TRec3;
    rec4 : TRec4;
    rec5 : Trec5;
    rec6 : TRec6;
    rec7 : TRec7;
    rec8 : TRec8;

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

The output of this program will be :

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

And this is as expected:

- In `Trec1`, since `B` has size 2, it is aligned on a 2 byte boundary, thus leaving an empty byte between `A` and `B`, and making the total size 4. In `Trec2`, `B` is aligned on a 1-byte boundary, right after `A`, hence, the total size of the record is 3.
- For `Trec3`, the sizes of `A,B` are 1, and hence they are aligned on 1 byte boundaries. The same is true for `Trec4`.

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

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

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

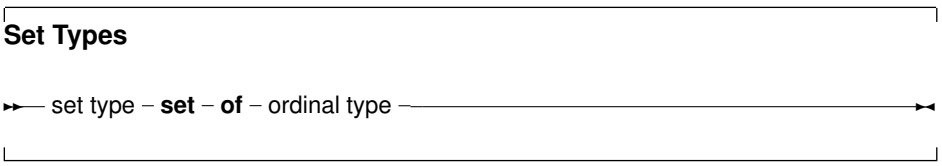
and

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

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

3.3.3 Set types

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



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

```
Type
  Junk = Set of Char;

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

Given these declarations, the following assignment is legal:

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

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

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

3.3.4 File types

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

Here is the type declaration for a file type:

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

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

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

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

3.4 Pointers

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

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

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

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

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

The expression

```
BP^
```

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

```
BP^[23]
```

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

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

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

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

can be considered equivalent to the following array declaration:

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

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

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

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

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

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

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

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

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

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

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

3.5 Forward type declarations

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

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

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

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

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

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

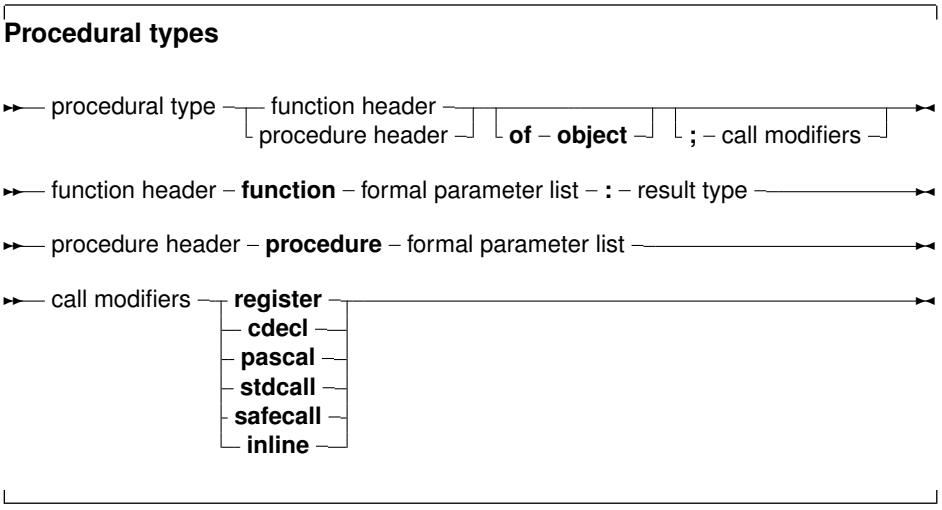
The referenced type should be introduced later in the same `Type` block. No other block may come between the definition of the pointer type and the referenced type. Indeed, even the word `Type` itself may not re-appear: in effect it would start a new type-block, causing the compiler to resolve all pending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

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

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

3.6 Procedural types

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



For a description of formal parameter lists, see chapter 11, page 127. The two following examples are valid type declarations:

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

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

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

4. A method address.

Given these declarations, the following assignments are valid:

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

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

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

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

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

3.7 Variant types

3.7.1 Definition

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

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

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

This means that the following assignments are valid:

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

Var
```

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

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

And of course vice-versa as well.

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

```
Program testv;

uses variants;

Var
  A : Variant;
  I : integer;

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

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

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

3.7.2 Variants in assignments and expressions

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

```
program testv3;

uses Variants;

Var
  V : Variant;
  I : Integer;

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

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

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

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

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

3.7.3 Variants and interfaces

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

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

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

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

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

The line

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

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

Chapter 4

Variables

4.1 Definition

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

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

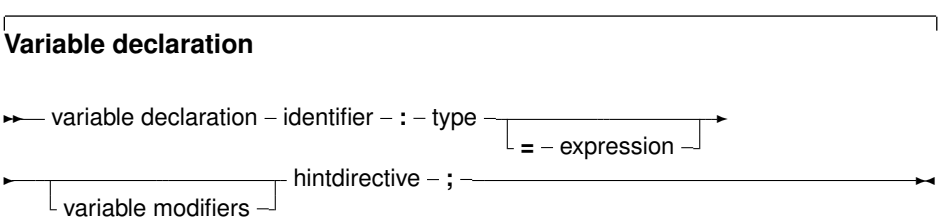
The Free Pascal compiler handles the allocation of these memory locations transparantly, although this location can be influenced in the declaration.

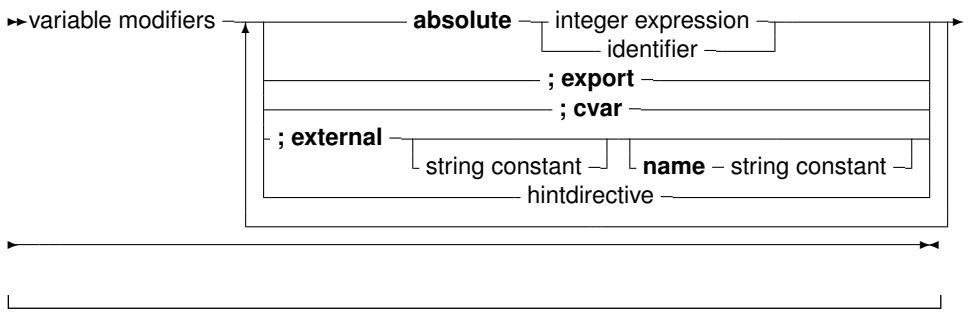
The Free Pascal compiler also handles reading values from or writing values to the variables transparantly. But even this can be explicitly handled by the programmer when using properties.

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

4.2 Declaration

The variables must be declared in a variable declaration section of a unit or a procedure or function. It looks as follows:





This means that the following are valid variable declarations:

```
Var
  curterm1 : integer;

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

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

  curterm6 : integer absolute curterm1;

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

  curterm11 : integer = 1 ;
```

The difference between these declarations is as follows:

1. The first form (`curterm1`) defines a regular variable. The compiler manages everything by itself.
2. The second form (`curterm2`) declares also a regular variable, but specifies that the assembler name for this variable equals the name of the variable as written in the source.
3. The third form (`curterm3`) declares a variable which is located externally: the compiler will assume memory is located elsewhere, and that the assembler name of this location is specified by the name of the variable, as written in the source. The name may not be specified.
4. The fourth form is completely equivalent to the third, it declares a variable which is stored externally, and explicitly gives the assembler name of the location. If `cvar` is not used, the name must be specified.
5. The fifth form is a variant of the fourth form, only the name of the library in which the memory is reserved is specified as well.
6. The sixth form declares a variable (`curterm6`), and tells the compiler that it is stored in the same location as another variable (`curterm1`).

7. The seventh form declares a variable (`curterm7`), and tells the compiler that the assembler label of this variable should be the name of the variable (case sensitive) and must be made public. i.e. it can be referenced from other object files.
8. The eighth form (`curterm8`) is equivalent to the seventh: 'public' is an alias for 'export'.
9. The ninth and tenth form are equivalent: they specify the assembler name of the variable.
10. the elevents form declares a variable (`curterm11`) and initializes it with a value (1 in the above case).

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

4.3 Scope

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

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

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

4.4 Initialized variables

By default, variables in Pascal are not initialized after their declaration. Any assumption that they contain 0 or any other default value is erroneous: They can contain rubbish. To remedy this, the concept of initialized variables exists. The difference with normal variables is that their declaration includes an initial value, as can be seen in the diagram in the previous section.

Given the declaration:

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

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

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

Var
  S : String = SDefault;
```

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

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

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

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

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

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

4.5 Thread Variables

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

```
Threadvar
  IOResult : Integer;
```

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

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

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

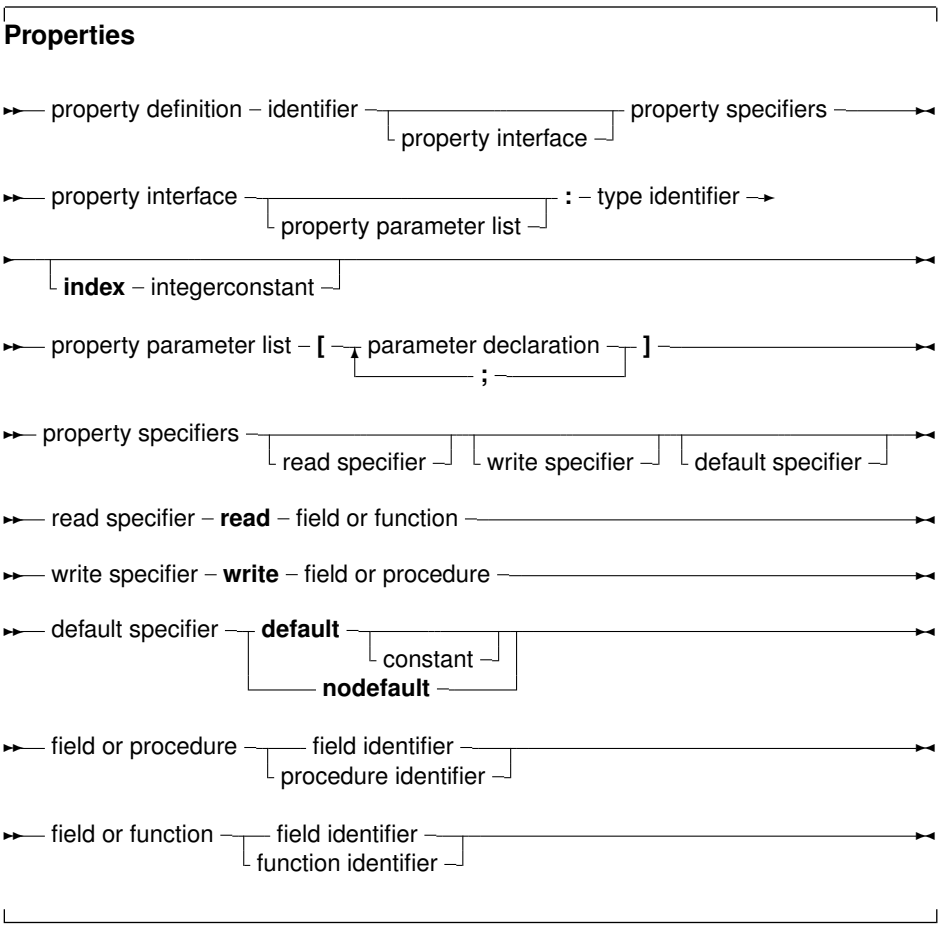
4.6 Properties

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

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

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

The declaration is as follows:



The following is an example:

```
{$mode objfpc}
unit testprop;

Interface

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

Property
  MyProp : Integer Read GetMyInt Write SetMyInt;

Implementation

Uses sysutils;
```

```
Var
  FMyInt : Integer;

Function GetMyInt : Integer;

begin
  Result:=FMyInt;
end;

Procedure SetMyInt(Value : Integer);

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

end.
```

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

```
{$mode objfpc}
unit testrw;

Interface

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

Implementation

Uses sysutils;

Var
  FMyInt : Integer;

Function GetMyInt : Integer;

begin
  Result:=FMyInt;
end;

Procedure SetMyInt(Value : Integer);

begin
  If ((Value mod 2)=1) then
    Raise Exception.Create('Only even values are allowed');
  FMyInt:=Value;
end;

end.
```

The unit testprop would then look like:

```
{$mode objfpc}
unit testprop;

Interface

uses testrw;

Property
  MyProp : Integer Read GetMyInt Write SetMyInt;

Implementation

end.
```

More information about properties can be found in chapter 6, page 67.

Chapter 9

Expressions

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

When using multiple operands in an expression, the precedence rules of table (9.1) are used. When determining the precedence, the compiler uses the following rules:

Table 9.1: Precedence of operators

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

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

Remark: The order in which expressions of the same precedence are evaluated is not guaranteed to be left-to-right. In general, no assumptions on which expression is evaluated first should be made in such a case. The compiler will decide which expression to evaluate first based on optimization rules. Thus, in the following expression:

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

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

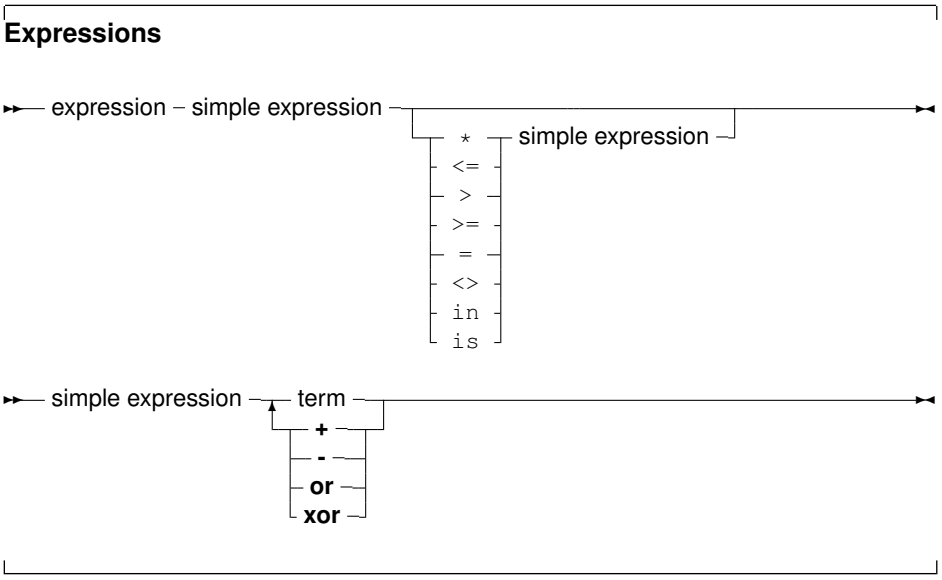
If one expression *must* be executed before the other, it is necessary to split up the statement using temporary results:

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

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

9.1 Expression syntax

An expression applies relational operators to simple expressions. Simple expressions are a series of terms (what a term is, is explained below), joined by adding operators.



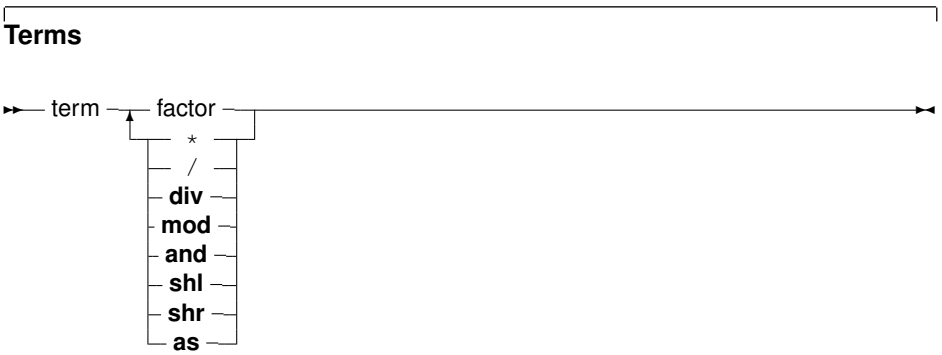
The following are valid expressions:

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

And here are some simple expressions:

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

Terms consist of factors, connected by multiplication operators.

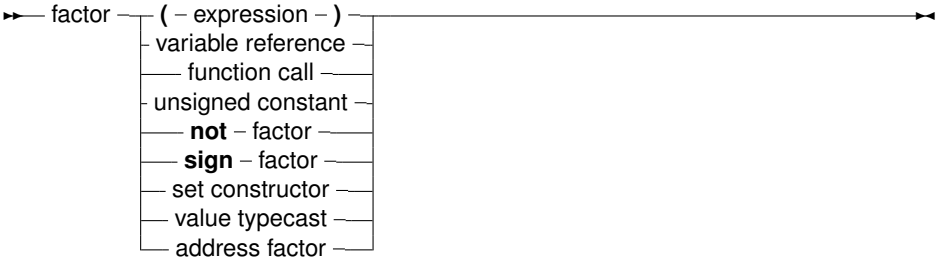


Here are some valid terms:

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

Factors are all other constructions:

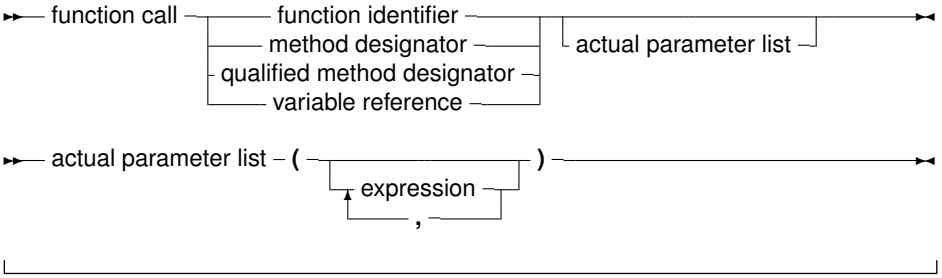
Factors



9.2 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 (unless default parameter values are used).

2. The types of the parameters must be compatible. For variable reference parameters, the parameter types must be exactly the same.

If no matching function is found, then the compiler will generate an error. Which error depends - among other things - on whether the function is overloaded or not: i.e. multiple functions with the same name, but different parameter lists.

There are cases when the compiler will not execute the function call in an expression. This is the case when assigning a value to a procedural type variable, as in the following example in Delphi or Turbo Pascal mode:

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

In the above listing, the 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? In `fpc` and `objfpc` mode this is solved by considering a procedural variable as equivalent to a pointer. Thus the compiler will give a type mismatch error, since `AddOne` is considered a call to a function with integer result, and `F` is a pointer.

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

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

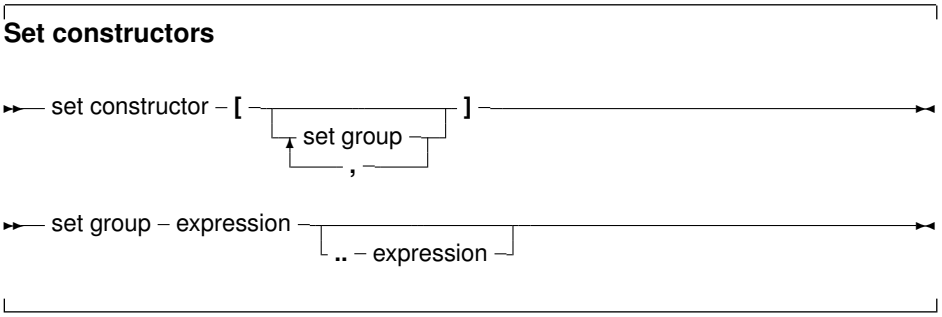
The left hand side of the boolean expression is an address. The right hand side also, and so the compiler compares 2 addresses. How to compare the values that both functions return ? By adding an empty parameter list:

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

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

9.3 Set constructors

When a set-type constant must be entered in an expression, a set constructor must be given. In essence this is the same thing as when a type is defined, only there is no identifier to identify the set with. A set constructor is a comma separated list of expressions, enclosed in square brackets.



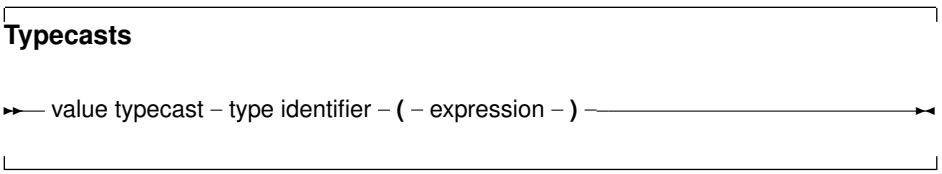
All set groups and set elements must be of the same ordinal type. The empty set is denoted by `[]`, and it can be assigned to any type of set. A set group with a range `[A..Z]` makes all values in the range a set element. The following are valid set constructors:

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

Remark: If the first range specifier has a bigger ordinal value than the second, the resulting set will be empty, e.g., `['Z'..'A']` denotes an empty set. One should be careful when denoting a range.

9.4 Value typecasts

Sometimes it is necessary to change the type of an expression, or a part of the expression, to be able to be assignment compatible. This is done through a value typecast. The syntax diagram for a value typecast is as follows:



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

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

In general, the type size of the expression and the size of the type cast must be the same. However, for ordinal types (byte, char, word, boolean, enumerateds) this is not so, they can be used interchangeably. That is, the following will work, although the sizes do not match.

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

This is compatible with Delphi or Turbo Pascal behaviour.

9.5 Variable typecasts

A variable can be considered a single factor in an expression. It can therefore be typecast as well. A variable can be typecast to any type, provided the type has the same size as the original variable.

It is a bad idea to typecast integer types to real types and vice versa. It's better to rely on type assignment compatibility and using some of the standard type changing functions.

Note that variable typecasts can occur on either side of an assignment, i.e. the following are both valid typecasts:

```
Var
  C : Char;
  B : Byte;

begin
  B:=Byte(C);
  Char(B):=C;
end;
```

Pointer variables can be typecasted to procedural types, but not to method pointers.

A typecast is an expression of the given type, which means the typecast can be followed by a qualifier:

```
Type
  TWordRec = Packed Record
    L,H : Byte;
  end;

Var
  P : Pointer;
  W : Word;
  S : String;

begin
  TWordRec(W).L:=$FF;
  TWordRec(W).H:=0;
  S:=TObject(P).ClassName;
```

9.6 Unaligned typecasts

A special typecast is the `Unaligned` typecast of a variable or expression. This is not a real typecast, but is rather a hint for the compiler that the expression may be misaligned (i.e. not on an aligned memory address). Some processors do not allow direct access to misaligned data structures, and therefor must access the data byte per byte.

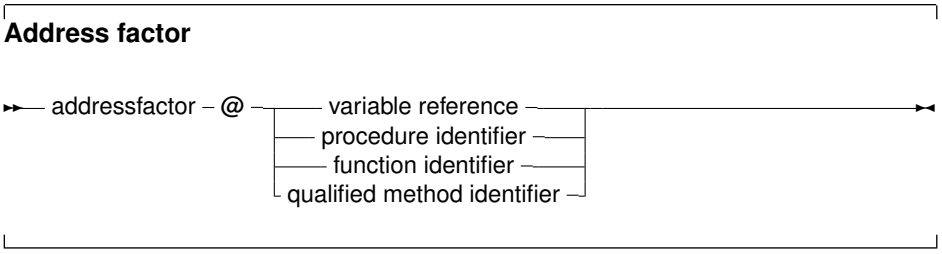
Typecasting an expression with the unaligned keyword signals the compiler that it should access the data byte per byte.

Example:

```
program me;

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

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

9.7 The @ operator

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



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

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

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

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

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

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

This means that the following will work:

```
Procedure MyProc;

begin
end;

Var
  P : PChar;

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

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

9.8 Operators

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

9.8.1 Arithmetic operators

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

Table 9.2: Binary arithmetic operators

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

of `Div` and `Mod`, which accept only integer expressions as operands, all operators accept real and integer expressions as operands.

For binary operators, the result type will be integer if both operands are integer type expressions. If one of the operands is a real type expression, then the result is real.

As an exception, division (`/`) results always in real values.

Table 9.3: Unary arithmetic operators

Operator	Operation
+	Sign identity
−	Sign inversion

For unary operators, the result type is always equal to the expression type. The division (`/`) and `Mod` operator will cause run-time errors if the second argument is zero.

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

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

But it executes faster than the right hand side expression.

9.8.2 Logical operators

Logical operators act on the individual bits of ordinal expressions. Logical operators require operands that are of an integer type, and produce an integer type result. The possible logical operators are listed in table (9.4). The following are valid logical expressions:

Table 9.4: Logical operators

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

```
A shr 1   { same as A div 2, but faster}
Not 1     { equals -2 }
Not 0     { equals -1 }
Not -1    { equals 0  }
B shl 2   { same as B * 4 for integers }
1 or 2    { equals 3 }
3 xor 1   { equals 2 }
```

9.8.3 Boolean operators

Boolean operators can be considered logical operations on a type with 1 bit size. Therefore the `shl` and `shr` operations have little sense. Boolean operators can only have boolean type operands, and the resulting type is always boolean. The possible operators are listed in table (9.5)

Table 9.5: Boolean operators

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

Remark: By default, boolean expressions are evaluated with short-circuit evaluation. This means that from the moment the result of the complete expression is known, evaluation is stopped and the result is returned. For instance, in the following expression:

```
B := True or MaybeTrue;
```

The compiler will never look at the value of `MaybeTrue`, since it is obvious that the expression will always be `True`. As a result of this strategy, if `MaybeTrue` is a function, it will not get called ! (This can have surprising effects when used in conjunction with properties)

9.8.4 String operators

There is only one string operator: `+`. Its action is to concatenate the contents of the two strings (or characters) it acts on. One cannot use `+` to concatenate null-terminated (`PChar`) strings. The following are valid string operations:

```
'This is ' + 'VERY ' + 'easy !'
Dirname+'\'
```

The following is not:

```
Var
  Dirname = Pchar;
...
  Dirname := Dirname+'\';
```

Because `Dirname` is a null-terminated string.

Note that if all strings in a string expressions are short strings, the resulting string is also a short string. Thus, a truncation may occur: there is no automatic upscaling to ansistring.

If all strings in a string expression are ansistrings, then the result is an ansistring.

If the expression contains a mix of ansistrings and shortstrings, the result is an ansistring.

The value of the `{$H}` switch can be used to control the type of constant strings; By default, they are short strings (and thus limited to 255 characters).

9.8.5 Set operators

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

The following program gives some valid examples of set operations:

Table 9.6: Set operators

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

```
Type
  Day = (mon,tue,wed,thu,fri,sat,sun);
  Days = set of Day;

Procedure PrintDays(W : Days);
Const
  DayNames : array [Day] of String[3]
           = ('mon','tue','wed','thu',
              'fri','sat','sun');
Var
  D : Day;
  S : String;
begin
  S:='';
  For D:=Mon to Sun do
    if D in W then
      begin
      If (S<>'') then S:=S+',';
      S:=S+DayNames[D];
      end;
  Writeln('[',S,']');
end;

Var
  W : Days;

begin
   W:=[mon,tue]+[wed,thu,fri]; // equals [mon,tue,wed,thu,fri]
   PrintDays(W);
   W:=[mon,tue,wed]-[wed];     // equals [mon,tue]
   PrintDays(W);
   W:=[mon,tue,wed]-[wed,thu];     // also equals [mon,tue]
   PrintDays(W);
   W:=[mon,tue,wed]*[wed,thu,fri]; // equals [wed]
   PrintDays(W);
   W:=[mon,tue,wed]><[wed,thu,fri]; // equals [mon,tue,thu,fri]
   PrintDays(W);
end.
```

As can be seen, the union is equivalent to a binary OR, while the intersection is equivalent

to a binary AND, and the summetric difference equals a XOR operation.

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

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

is equivalent to

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

and

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

is equivalent to

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

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

9.8.6 Relational operators

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

Table 9.7: Relational operators

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

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

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

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

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

9.8.7 Class operators

Class operators are slightly different from the operators above in the sense that they can only be used in class expressions which return a class. There are only 2 class operators, as can be seen in table (9.8). An expression containing the `is` operator results in a

Table 9.8: Class operators

Operator	Action
`is`	Checks class type
`as`	Conditional typecast

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

```
Object is Class
```

This expression is completely equivalent to

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

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

The following are examples:

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

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

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

```
  Object as Class
```

This is equivalent to the following statements:

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

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

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

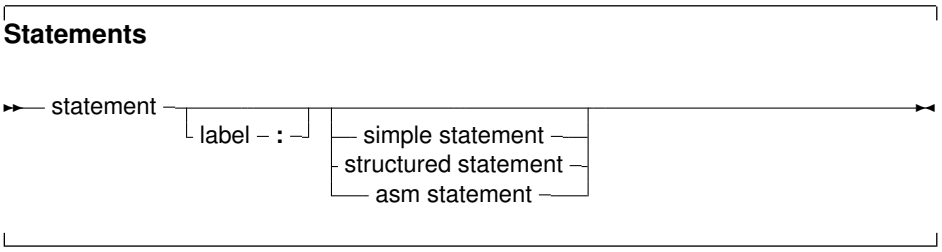
```
Var
  C : TComponent;
  O : TObject;
```

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

Chapter 10

Statements

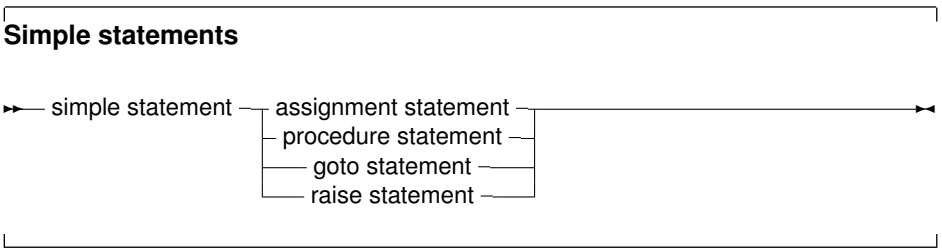
The heart of each algorithm are the actions it takes. These actions are contained in the statements of a program or unit. Each statement can be labeled and jumped to (within certain limits) with `Goto` statements. This can be seen in the following syntax diagram:



A label can be an identifier or an integer digit.

10.1 Simple statements

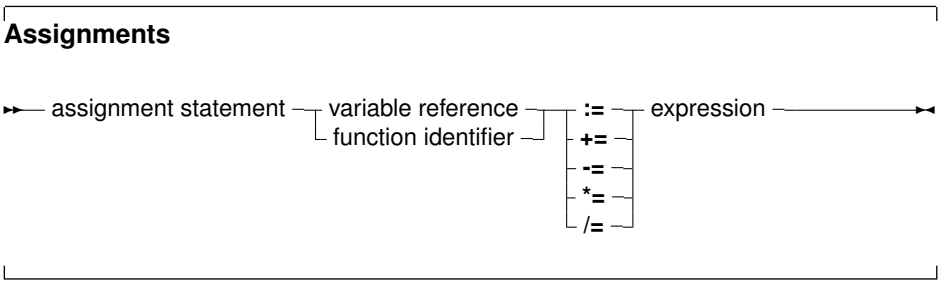
A simple statement cannot be decomposed in separate statements. There are basically 4 kinds of simple statements:



Of these statements, the *raise statement* will be explained in the chapter on Exceptions (chapter 14, page 161)

10.1.1 Assignments

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



In addition to the standard Pascal assignment operator (`:=`), which simply replaces the value of the varable with the value resulting from the expression on the right of the `:=` operator, Free Pascal supports some C-style constructions. All available constructs are listed in table (10.1).

Table 10.1: Allowed C constructs in Free Pascal

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

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

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

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

Keeping in mind that the dereferencing of a typed pointer results in a variable of the type the pointer points to, the following are also valid assignments:

```
Var
  L : ^Longint;
  P : PPChar;

begin
  L^:=3;
  P^^:='A';
```

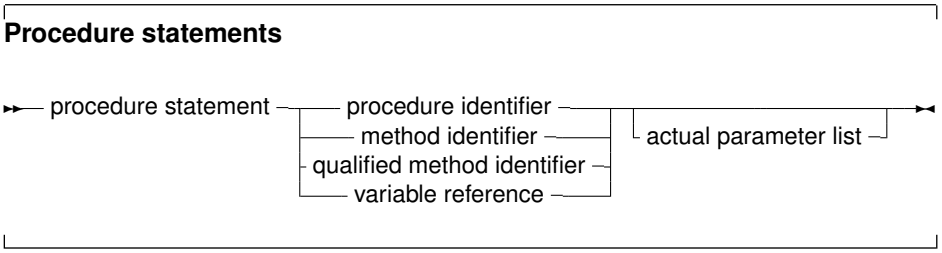
Note the double dereferencing in the second assignment.

10.1.2 Procedure statements

Procedure statements are calls to subroutines. There are different possibilities for procedure calls:

- A normal procedure call.
- An object method call (fully qualified or not).
- Or even a call to a procedural type variable.

All types are present in the following diagram:



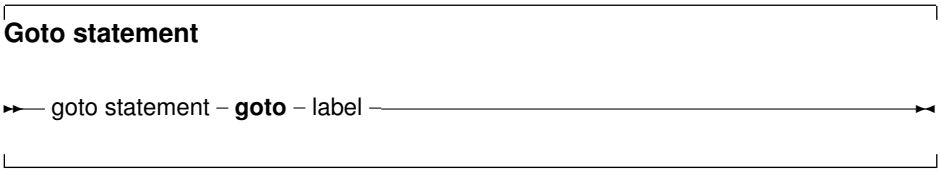
The Free Pascal compiler will look for a procedure with the same name as given in the procedure statement, and with a declared parameter list that matches the actual parameter list. The following are valid procedure statements:

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

Remark: When looking for a function that matches the parameter list of the call, the parameter types should be assignment-compatible for value and const parameters, and should match exactly for parameters that are passed by reference.

10.1.3 Goto statements

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



When using `goto` statements, the following must be kept in mind:

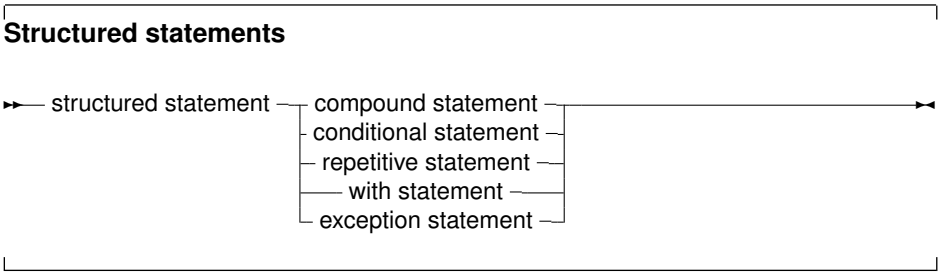
1. The jump label must be defined in the same block as the `Goto` statement.
2. Jumping from outside a loop to the inside of a loop or vice versa can have strange effects.
3. To be able to use the `Goto` statement, the `-Sg` compiler switch must be used, or `{$GOTO ON}` must be used.

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

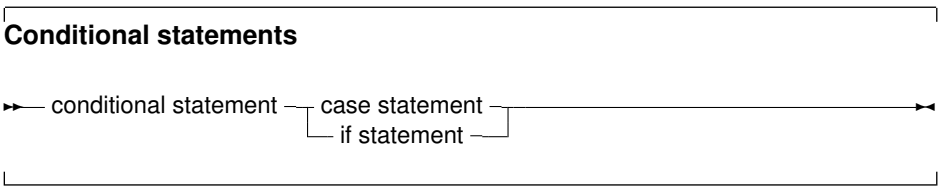
```
label
  jumpto;
...
Jumpto :
  Statement;
...
Goto jumpto;
...
```

10.2 Structured statements

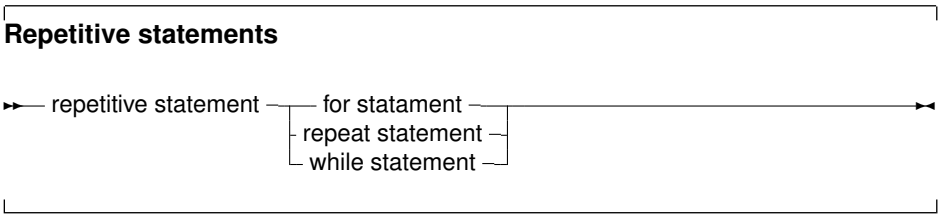
Structured statements can be broken into smaller simple statements, which should be executed repeatedly, conditionally or sequentially:



Conditional statements come in 2 flavours :



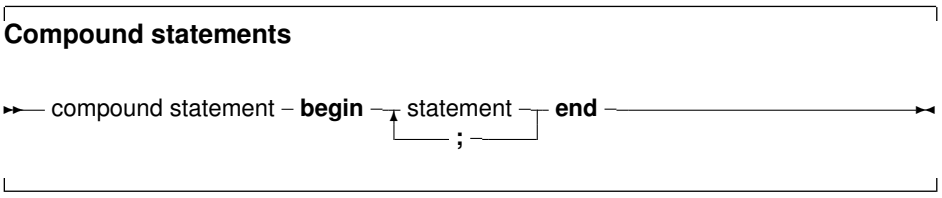
Repetitive statements come in 3 flavours:



The following sections deal with each of these statements.

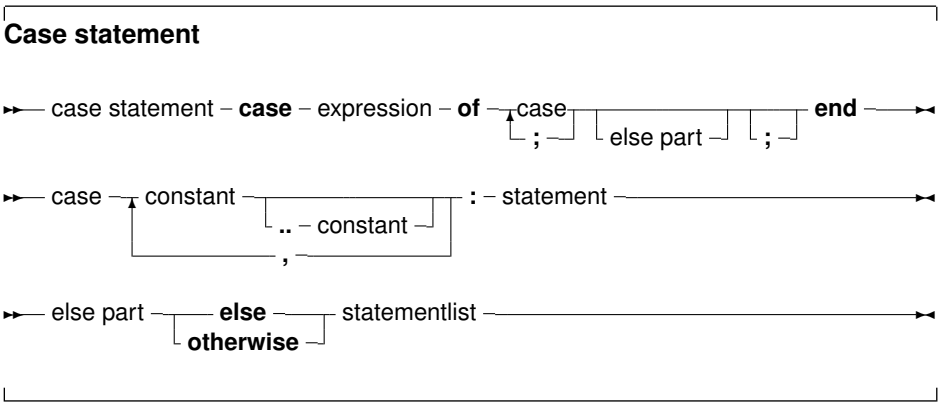
10.2.1 Compound statements

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



10.2.2 The `Case` statement

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



The constants appearing in the various case parts must be known at compile-time, and can be of the following types : enumeration types, Ordinal types (except boolean), and chars. The case expression must be also of this type, or a compiler error will occur. All case constants must have the same type.

The compiler will evaluate the case expression. If one of the case constants' value matches the value of the expression, the statement that follows this constant is executed. After that, the program continues after the final `end`.

If none of the case constants match the expression value, the statement list after the `else` or `otherwise` keyword is executed. This can be an empty statement list. If no else part is present, and no case constant matches the expression value, program flow continues after the final `end`.

The case statements can be compound statements (i.e. a `Begin..End` block).

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

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

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

The following are valid case statements:

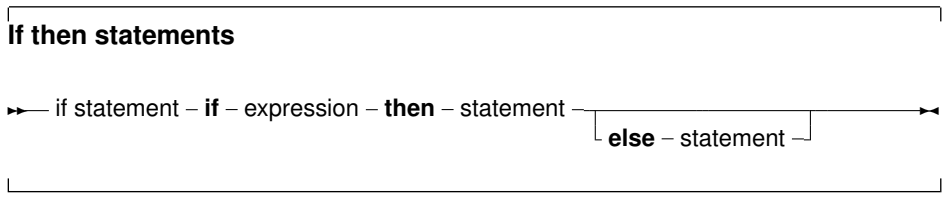
```
Case C of
 'a' : WriteLn ('A pressed');
 'b' : WriteLn ('B pressed');
 'c' : WriteLn ('C pressed');
else
  WriteLn ('unknown letter pressed : ',C);
end;
```

Or

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

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

10.2.3 The `If..then..else` statement

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



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

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

Some points to note:

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

For example:

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

Despite its appearance, the statement is syntactically equivalent to

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

and not to

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

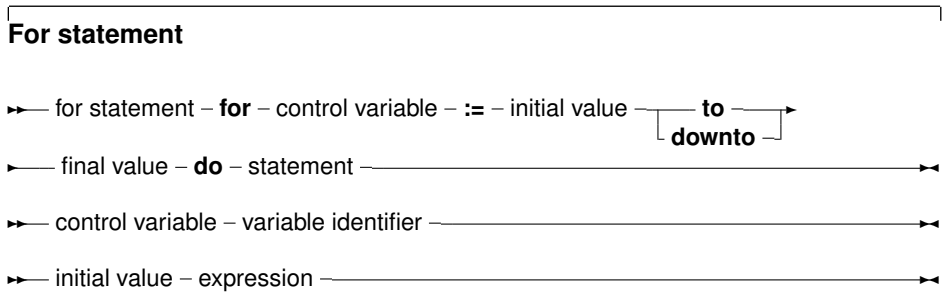
If it is this latter construct which is needed, the `begin` and `end` keywords must be present. When in doubt, it is better to add them.

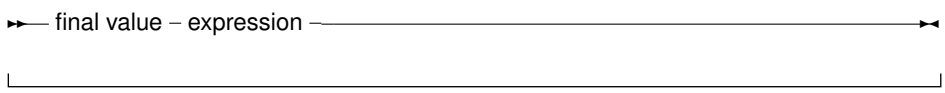
The following is a valid statement:

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

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

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





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

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

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

Remark: Free Pascal always calculates the upper bound before initializing the counter variable with the initial value.

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

The following are valid loops:

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

The following will generate an error:

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

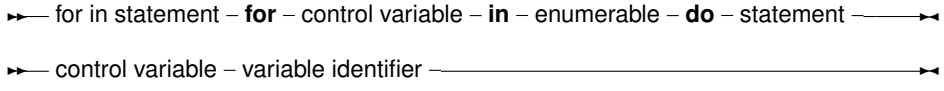
because the loop variable `I` cannot be assigned to inside the loop.

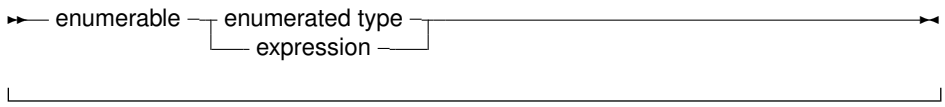
If the statement is a compound statement, then the `Break` and `Continue` reserved words can be used to jump to the end or just after the end of the `For` statement.

10.2.5 The `For..in..do` statement

As of version 2.4.2, Free Pascal supports the `For..in` loop construction. A for..in loop is used in case one wants to calculate something a fixed number of times with an enumerable loop variable. The prototype syntax is as follows:

For statement





Here, `Statement` can be a compound statement. The enumerable must be an expression that consists of a fixed number of elements: the loop variable will be made equal to each of the elements in turn and the statement following the `do` keyword will be executed.

The enumerable expression can be one of 5 cases:

1. An enumeration type identifier. The loop will then be over all elements of the enumeration type. The control variable must be of the enumeration type.
2. A set value. The loop will then be over all elements in the set, the control variable must be of the base type of the set.
3. An array value. The loop will be over all elements in the array, and the control variable must have the same type as an element in the array. As a special case, a string is regarded as an array of characters.
4. An enumeratable class instance. This is an instance of a class that supports the `IEnumerator` and `IENumerable` interfaces. In this case, the control variable's type must equal the type of the `IEnumerator.GetCurrent` return value.
5. Any type for which an `enumerator` operator is defined. The `enumerator` operator must return a class that implements the `IEnumerator` interface. The type of the control variable's type must equal the type of the enumerator class `GetCurrent` return value type.

The simplest case of the `for..in` loop is using an enumerated type:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  d : TWeekday;

begin
  for d in TWeekday do
    writeln(d);
end.
```

This will print all week days to the screen.

The above `for..in` construct is equivalent to the following `for..to` construct:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  d : TWeekday;

begin
  for d:=Low(TWeekday) to High(TWeekday) do
    writeln(d);
end.
```

A second case of `for..in` loop is when the enumerable expression is a set, and then the loop will be executed once for each element in the set:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  Week : set of TWeekDay
       = [monday, tuesday, wednesday, thursday, friday];
  d : TWeekday;

begin
  for d in Week do
    writeln(d);
end.
```

This will print the names of the week days to the screen. Note that the variable `d` is of the same type as the base type of the set.

The above `for..in` construct is equivalent to the following `for..to` construct:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday,
              friday,saturday,sunday);

Var
  Week : set of TWeekDay
       = [monday, tuesday, wednesday, thursday, friday];

  d : TWeekday;

begin
  for d:=Low(TWeekday) to High(TWeekday) do
    if d in Week then
      writeln(d);
end.
```

The third possibility for a `for..in` loop is when the enumerable expression is an array:

```
var
  a : Array[1..7] of string
    = ('monday','tuesday','wednesday','thursday',
       'friday','saturday','sunday');

Var
  S : String;

begin
  For s in a do
    Writeln(s);
end.
```

This will also print all days in the week, and is equivalent to

```
var
  a : Array[1..7] of string
    = ('monday','tuesday','wednesday','thursday',
       'friday','saturday','sunday');

Var
  i : integer;

begin
  for i:=Low(a) to high(a) do
    Writeln(a[i]);
end.
```

A `string` type is equivalent to an `array of char`, and therefor a string can be used in a `for..in` loop. The following will print all letters in the alphabet, each letter on a line:

```
Var
  c : char;

begin
 for c in 'abcdefghijklmnopqrstuvwxyz' do
   writeln(c);
end.
```

The fourth possibility for a `for..in` loop is using classes. A class can implement the `IEnumerable` interface, which is defined as follows:

```
IEnumerable = interface(IInterface)
  function GetEnumerator: IEnumerator;
end;
```

The actual return type of the `GetEnumerator` must not necessarily be an `IEnumerator` interface, instead, it can be a class which implements the methods of `IEnumerator`:

```
IEnumerator = interface(IInterface)
  function GetCurrent: TObject;
  function MoveNext: Boolean;
  procedure Reset;
  property Current: TObject read GetCurrent;
end;
```

The `Current` property and the `MoveNext` method must be present in the class returned by the `GetEnumerator` method. The actual type of the `Current` property need not be a `TObject`. When encountering a `for..in` loop with a class instance as the 'in' operand, the compiler will check each of the following conditions:

- Whether then class in the enumerable expression implements a method `GetEnumerator`
- Whether the result of `GetEnumerator` is a class with the following method:

  ```
  Function MoveNext : Boolean
  ```

- Whether the result of `GetEnumerator` is a class with the following read-only property:

  ```
  Property Current : AType;
  ```

The type of the property must match the type of the control variable of the `for..in` loop.

Neither the `IEnumerator` nor the `IEnumerable` interfaces must actually be declared by the enumerable class: the compiler will detect whether these interfaces are present using the above checks. The interfaces are only defined for Delphi compatibility and are not used internally. (it would also be impossible to enforce their correctness).

The Classes unit contains a number of classes that are enumerable:

TFPList Enumerates all pointers in the list.

TList Enumerates all pointers in the list.

TCollection Enumerates all items in the collection.

TStringList Enumerates all strings in the list.

TComponent enumerates all child components owned by the component.

Thus, the following code will also print all days in the week:

```
{$mode objfpc}
uses classes;

Var
  Days : TStrings;
  D : String;

begin
  Days:=TStringList.Create;
  try
    Days.Add('Monday');
    Days.Add('Tuesday');
    Days.Add('Wednesday');
    Days.Add('Thursday');
    Days.Add('Friday');
    Days.Add('Saturday');
    Days.Add('Sunday');
    For D in Days do
      Writeln(D);
  Finally
    Days.Free;
  end;
end.
```

Note that the compiler enforces type safety: Declaring D as an integer will result in a compiler error:

```
testsl.pp(20,9) Error: Incompatible types: got "AnsiString" expected "LongInt"
```

The above code is equivalent to the following:

```
{$mode objfpc}
uses classes;
```

```
Var
  Days : TStrings;
  D : String;
  E : TStringsEnumerator;

begin
  Days:=TStringList.Create;
  try
    Days.Add('Monday');
    Days.Add('Tuesday');
    Days.Add('Wednesday');
    Days.Add('Thursday');
    Days.Add('Friday');
    Days.Add('Saturday');
    Days.Add('Sunday');
    E:=Days.getEnumerator;
    try
      While E.MoveNext do
        begin
        D:=E.Current;
        Writeln(D);
        end;
    Finally
      E.Free;
    end;
  Finally
    Days.Free;
  end;
end.
```

Both programs will output the same result.

The fifth and last possibility to use a `for..in` loop can be used to enumerate almost any type, using the `enumerator` operator. The `enumerator` operator must return a class that has the same signature as the `IEnumerate` approach above. The following code will define an enumerator for the `Integer` type:

```
Type


TEvenEnumerator = Class
  FCurrent : Integer;
  FMax : Integer;
  Function MoveNext : Boolean;
  Property Current : Integer Read FCurrent;
end;

Function TEvenEnumerator.MoveNext : Boolean;

begin
  FCurrent:=FCurrent+2;
  Result:=FCurrent<=FMax;
end;

operator enumerator(i : integer) : TEvenEnumerator;
```

```
begin
  Result:=TEvenEnumerator.Create;
  Result.FMax:=i;
end;

var
  I : Integer;
  m : Integer = 4;

begin
  For I in M do
    Writeln(i);
end.
```

The loop will print all nonzero even numbers smaller or equal to the enumerable. (2 and 4 in the case of the example).

Care must be taken when defining enumerator operators: the compiler will find and use the first available enumerator operator for the enumerable expression. For classes this also means that the `GetEnumerator` method is not even considered. The following code will define an enumerator operator which extracts the object from a stringlist:

```
{$mode objfpc}
uses classes;

Type
  TDayObject = Class
    DayOfWeek : Integer;
    Constructor Create(ADayOfWeek : Integer);
  end;

  TObjectEnumerator = Class
    FList : TStrings;
    FIndex : Integer;
    Function GetCurrent : TDayObject;
    Function MoveNext: boolean;
    Property Current : TDayObject Read GetCurrent;
  end;

Constructor TDayObject.Create(ADayOfWeek : Integer);

begin
  DayOfWeek:=ADayOfWeek;
end;

Function TObjectEnumerator.GetCurrent : TDayObject;
begin
  Result:=FList.Objects[Findex] as TDayObject;
end;

Function TObjectEnumerator.MoveNext: boolean;

begin
  Inc(FIndex);
```

```
  Result:=(FIndex<FList.Count);
end;

operator enumerator (s : TStrings) : TObjectEnumerator;

begin
  Result:=TObjectEnumerator.Create;
  Result.Flist:=S;
  Result.FIndex:=-1;
end;

Var
  Days : TStrings;
  D : String;
  O : TdayObject;

begin
  Days:=TStringList.Create;
  try
    Days.AddObject('Monday',TDayObject.Create(1));
    Days.AddObject('Tuesday',TDayObject.Create(2));
    Days.AddObject('Wednesday',TDayObject.Create(3));
    Days.AddObject('Thursday',TDayObject.Create(4));
    Days.AddObject('Friday',TDayObject.Create(5));
    Days.AddObject('Saturday',TDayObject.Create(6));
    Days.AddObject('Sunday',TDayObject.Create(7));
    For O in Days do
      Writeln(O.DayOfWeek);
  Finally
    Days.Free;
  end;
end.
```

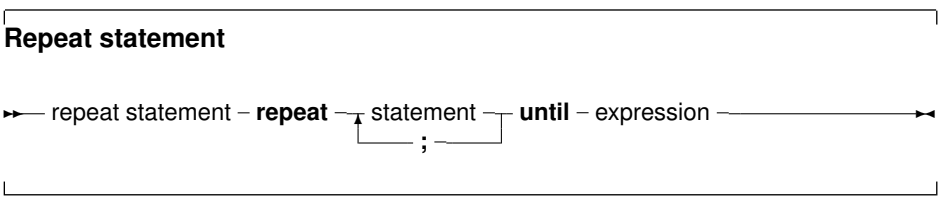
The above code will print the day of the week for each day in the week.

If a class is not enumerable, the compiler will report an error when it is encountered in a `for...in` loop.

Remark: Like the `for..to` loop, it is not allowed to change (i.e. assign a value to) the value of a loop control variable inside the loop.

10.2.6 The `Repeat..until` statement

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



This will execute the statements between `repeat` and `until` up to the moment when `Expression` evaluates to `True`. Since the `expression` is evaluated *after* the execution of the statements, they are executed at least once.

Be aware of the fact that the boolean expression `Expression` will be short-cut evaluated by default, meaning that the evaluation will be stopped at the point where the outcome is known with certainty.

The following are valid `repeat` statements

```
repeat
  WriteLn ('I =',i);
  I := I+2;
until I>100;

repeat
 X := X/2
until x<10e-3;
```

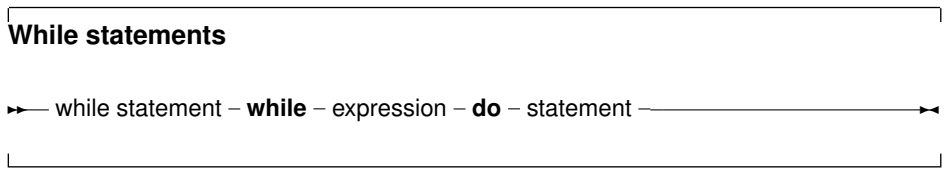
Note that the last statement before the `until` keyword does not need a terminating semicolon, but it is allowed.

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

10.2.7 The `While..do` statement

A `while` statement is used to execute a statement as long as a certain condition holds. In difference with the `repeat` loop, this may imply that the statement is never executed.

The prototype syntax of the `While..do` statement is



This will execute `Statement` as long as `Expression` evaluates to`True`. Since `Expression` is evaluated *before* the execution of `Statement`, it is possible that `Statement` isn't executed at all. `Statement` can be a compound statement.

Be aware of the fact that the boolean expression `Expression` will be short-cut evaluated by default, meaning that the evaluation will be stopped at the point where the outcome is known with certainty.

The following are valid `while` statements:

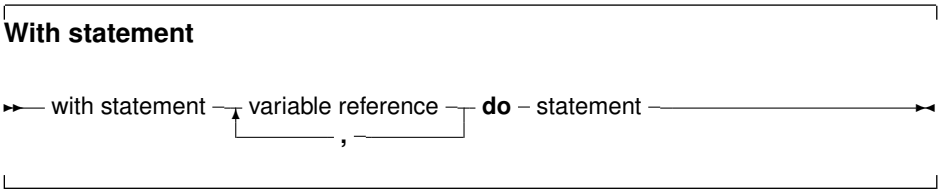
```
I := I+2;
while i<=100 do
  begin
  WriteLn ('I =',i);
  I := I+2;
  end;
X := X/2;
while x>=10e-3 do
  X := X/2;
```

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

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

10.2.8 The `With` statement

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



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

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

Var
  TheCustomer : Passenger;
```

The following statements are completely equivalent:

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

and

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

The statement

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

is equivalent to

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

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

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

Var S,T : Ar;
begin
  S.X := 1;S.Y := 1;
  T.X := 2;T.Y := 2;
  With S,T do
    WriteLn (X,' ',Y);
end.
```

The output of this program is

```
2 2
```

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

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

```
Var p : PAR;

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

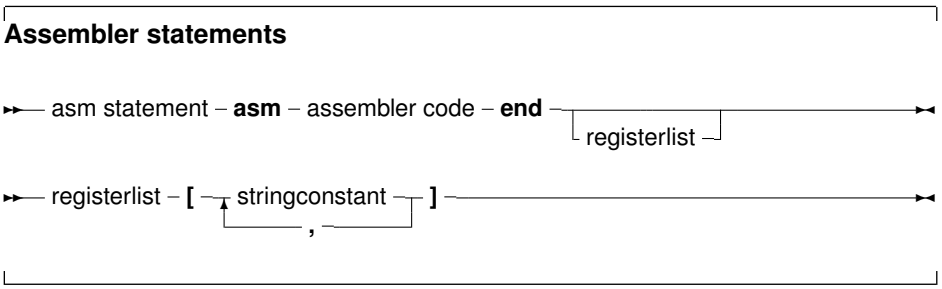
The reason the pointer cannot be changed is that the address is stored by the compiler in a temporary register. Changing the pointer won't change the temporary address. The same is true for classes.

10.2.9 Exception Statements

Free Pascal supports exceptions. Exceptions provide a convenient way to program error and error-recovery mechanisms, and are closely related to classes. Exception support is explained in chapter 14, page 161

10.3 Assembler statements

An assembler statement allows to insert assembler code right in the Pascal code.



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

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

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

Free Pascal supports various styles of assembler syntax. By default, `AT&T` syntax is assumed for the 80386 and compatibles platform. The default assembler style can be changed with the `{$asmmode xxx}` switch in the code, or the `-R` command-line option. More about this can be found in the Programmer's Guide.

Chapter 11

Using functions and procedures

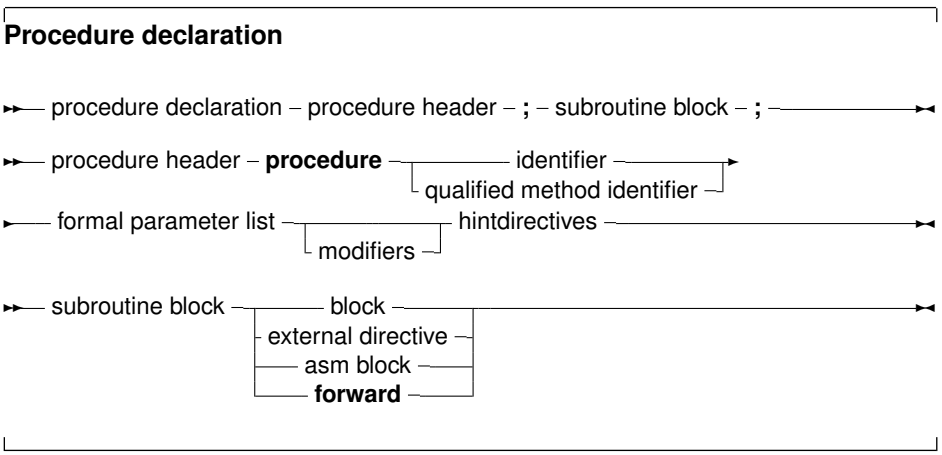
Free Pascal supports the use of functions and procedures. It supports

- Function overloading, i.e. functions with the same name but different parameter lists.
- `Const` parameters.
- Open arrays (i.e. arrays without bounds).
- Variable number of arguments as in C.
- Return-like construct as in C, through the `Exit` keyword.

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

11.1 Procedure declaration

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



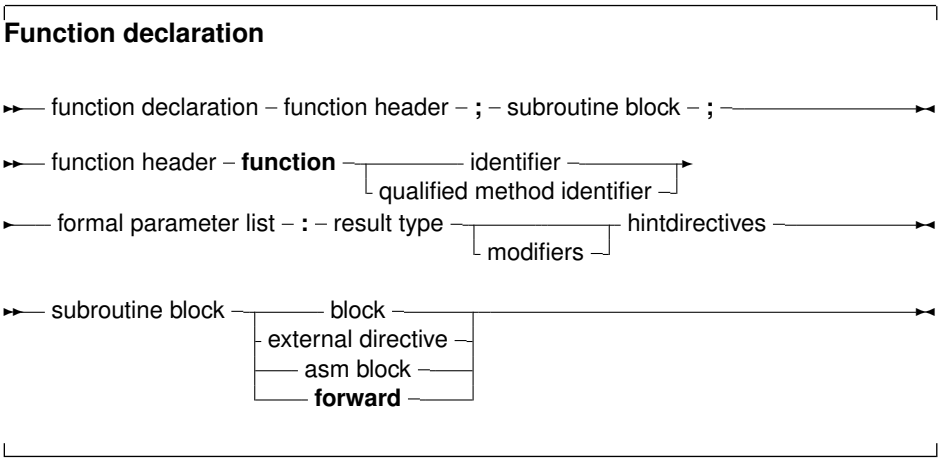
See section 11.4, page 129 for the list of parameters. A procedure declaration that is followed by a block implements the action of the procedure in that block. The following is a valid procedure :

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

Note that it is possible that a procedure calls itself.

11.2 Function declaration

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



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

11.3 Function results

The result of a function can be set by setting the result variable: this can be the function identifier or, (only in ObjFPC or Delphi mode) the special `Result` identifier:

```
Function MyFunction : Integer;

begin
  MyFunction:=12; // Return 12
end;
```

In `Delphi` or `ObjPas` mode, the above can also be coded as:

```
Function MyFunction : Integer;

begin
  Result:=12;
end;
```

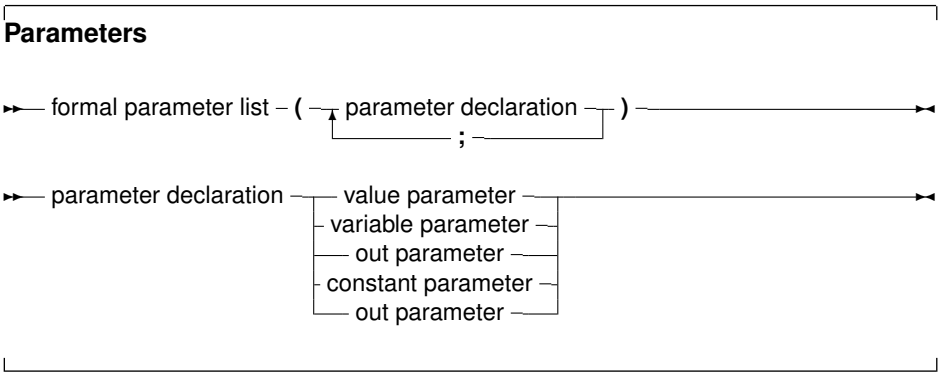
As an extension to Delphi syntax, the ObjFPC mode also supports a special extension of the `Exit` procedure:

```
Function MyFunction : Integer;

begin
  Exit(12);
end;
```

The `Exit` call sets the result of the function and jumps to the final `End` of the function declaration block. It can be seen as the equivalent of the C `return` instruction.

11.4 Parameter lists

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

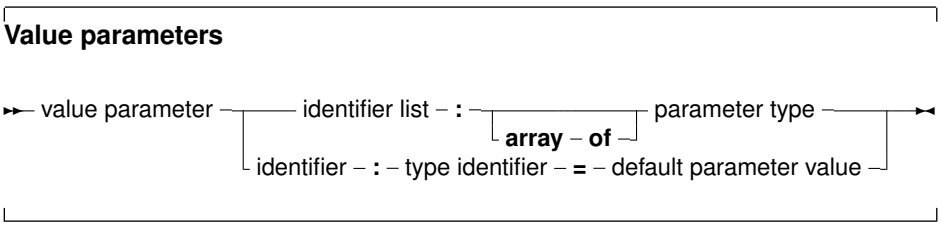


Constant parameters, out parameters and variable parameters can also be `untyped` parameters if they have no type identifier.

As of version 1.1, Free Pascal supports default values for both constant parameters and value parameters, but only for simple types. The compiler must be in `OBJFPC` or `DELPHI` mode to accept default values.

11.4.1 Value parameters

Value parameters are declared as follows:



When parameters are declared as value parameters, the procedure gets *a copy* of the parameters that the calling statement passes. Any modifications to these parameters are purely local to the procedure's block, and do not propagate back to the calling block.

A block that wishes to call a procedure with value parameters must pass assignment compatible parameters to the procedure. This means that the types should not match exactly, but can be converted to the actual parameter types. This conversion code is inserted by the compiler itself.

Care must be taken when using value parameters: Value parameters makes heavy use of the stack, especially when using large parameters. The total size of all parameters in the formal parameter list should be below 32K for portability's sake (the Intel version limits this to 64K).

Open arrays can be passed as value parameters. See section 11.4.5, page 133 for more information on using open arrays.

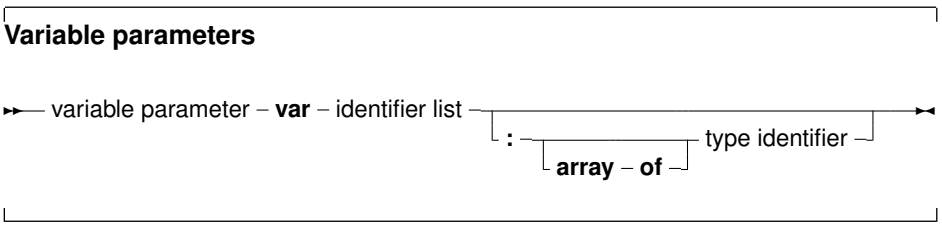
For a parameter of a simple type (i.e. not a structured type), a default value can be specified. This can be an untyped constant. If the function call omits the parameter, the default value will be passed on to the function. For dynamic arrays or other types that can be considered as equivalent to a pointer, the only possible default value is `Nil`.

The following example will print 20 on the screen:

```
program testp;

Const
  MyConst = 20;

Procedure MyRealFunc(I : Integer = MyConst);

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

begin
  MyRealFunc;
end.
```

11.4.2 Variable parameters

Variable parameters are declared as follows:



When parameters are declared as variable parameters, the procedure or function accesses immediatly the variable that the calling block passed in its parameter list. The procedure gets a pointer to the variable that was passed, and uses this pointer to access the variable's value. From this, it follows that any changes made to the parameter, will propagate back to the calling block. This mechanism can be used to pass values back in procedures. Because

of this, the calling block must pass a parameter of *exactly* the same type as the declared parameter's type. If it does not, the compiler will generate an error.

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

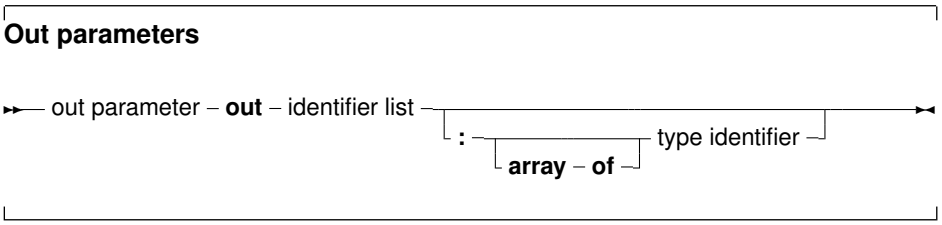
File type variables must always be passed as variable parameters.

Open arrays can be passed as variable parameters. See section 11.4.5, page 133 for more information on using open arrays.

Note that default values are not supported for variable parameters. This would make little sense since it defeats the purpose of being able to pass a value back to the caller.

11.4.3 Out parameters

Out parameters (output parameters) are declared as follows:



The purpose of an `out` parameter is to pass values back to the calling routine: The variable is passed by reference. The initial value of the parameter on function entry is discarded, and should not be used.

If a variable must be used to pass a value to a function and retrieve data from the function, then a variable parameter must be used. If only a value must be retrieved, a `out` parameter can be used.

Needless to say, default values are not supported for `out` parameters.

The difference of out parameters and parameters by reference is very small: the former gives the compiler more information about what happens to the arguments when passed to the procedure: It knows that the variable does not have to be initialized prior to the call. The following example illustrates this:

```
Procedure DoA(Var A : Integer);

begin
  A:=2;
  Writeln('A is ',A);
end;

Procedure DoB(Out B : Integer);

begin
  B:=2;
  Writeln('B is ',B);
end;
```

```
Var
  C,D : Integer;

begin
  DoA(C);
  DoB(D);
end.
```

Both procedures `DoA` and `DoB` do practically the same. But `DoB`'s declaration gives more information to the compiler, allowing it to detect that `D` does not have to initialized before `DoB` is called. Since the parameter `A` in `DoA` can receive a value as well as return one, the compiler notices that `C` was not initialized prior to the call to `DoA`:

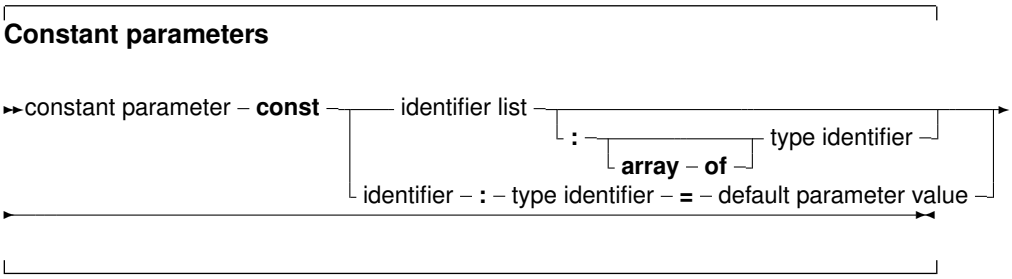
```
home: >fpc -S2 -vwhn testo.pp
testo.pp(19,8) Hint: Variable "C" does not seem to be initialized
```

This shows that it is better to use `out` parameters when the parameter is used only to return a value.

Remark: Out parameters are only supported in Delphi and ObjFPC mode. For the other modes, `out` is a valid identifier.

11.4.4 Constant parameters

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



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

Constant parameters can also be untyped. See section 11.4.2, page 130 for more information about untyped parameters.

As for value parameters, constant parameters can get default values.

Open arrays can be passed as constant parameters. See section 11.4.5, page 133 for more information on using open arrays.

11.4.5 Open array parameters

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

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

would be equivalent to

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

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

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

As of FPC 2.2, it is also possible to pass partial arrays to a function that accepts an open array. This can be done by specifying the range of the array which should be passed to the open array.

Given the declaration

```
Var
  A : Array[1..100];
```

the following call will compute and print the average of the 100 numbers:

```
  Writeln('Average of 100 numbers: ',Average(A));
```

But the following will compute and print the average of the first and second half:

```
  Writeln('Average of first 50 numbers: ',Average(A[1..50]));
  Writeln('Average of last  50 numbers: ',Average(A[51..100]));
```

11.4.6 Array of const

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

This is a special case of the `Open array` construction, where it is allowed to pass any expression in an array to a function or procedure. The expression must have a simple result

type: structures cannot be passed as an argument. This means that all ordinal, float or string types can be passed, as well as pointers, classes and interfaces.

The elements of the array of const are converted to a a special variant record:

```
Type
  PVarRec = ^TVarRec;
  TVarRec = record
     case VType : Ptrint of
       vtInteger    : (VInteger: Longint);
       vtBoolean    : (VBoolean: Boolean);
       vtChar       : (VChar: Char);
       vtWideChar   : (VWideChar: WideChar);
       vtExtended   : (VExtended: PExtended);
       vtString     : (VString: PShortString);
       vtPointer    : (VPointer: Pointer);
       vtPChar      : (VPChar: PChar);
       vtObject     : (VObject: TObject);
       vtClass      : (VClass: TClass);
       vtPWideChar  : (VPWideChar: PWideChar);
       vtAnsiString : (VAnsiString: Pointer);
       vtCurrency   : (VCurrency: PCurrency);
       vtVariant    : (VVariant: PVariant);
       vtInterface  : (VInterface: Pointer);
       vtWideString : (VWideString: Pointer);
       vtInt64      : (VInt64: PInt64);
       vtQWord      : (VQWord: PQWord);
   end;
```

Therefor, inside the procedure body, the `array of const` argument is equivalent to an open array of `TVarRec`:

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

Var I : longint;

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

```
      vtString     :
        Writeln ('ShortString, value :',args[i].VString^);
      vtPointer    :
        Writeln ('Pointer, value : ',Longint(Args[i].VPointer));
      vtPChar      :
        Writeln ('PCHar, value : ',Args[i].VPChar);
      vtObject     :
        Writeln ('Object, name : ',Args[i].VObject.Classname);
      vtClass      :
        Writeln ('Class reference, name :',Args[i].VClass.Classname);
      vtAnsiString :
        Writeln ('AnsiString, value :',AnsiString(Args[I].VAnsiStr
    else
        Writeln ('(Unknown) : ',args[i].vtype);
    end;
    end;
end;
```

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

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

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

```
program testaocc;
{$mode objfpc}

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

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

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

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

11.5 Function overloading

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

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

When the compiler encounters a call to the `Dec` function, it will first search which function it should use. It therefore checks the parameters in a function call, and looks if there is a function definition which matches the specified parameter list. If the compiler finds such a function, a call is inserted to that function. If no such function is found, a compiler error is generated.

functions that have a `cdecl` modifier cannot be overloaded. (Technically, because this modifier prevents the mangling of the function name by the compiler).

Prior to version 1.9 of the compiler, the overloaded functions needed to be in the same unit. Now the compiler will continue searching in other units if it doesn't find a matching version of an overloaded function in one unit, and if the `overload` keyword is present.

If the `overload` keyword is not present, then all overloaded versions must reside in the same unit, and if it concerns methods part of a class, they must be in the same class, i.e. the compiler will not look for overloaded methods in parent classes if the `overload` keyword was not specified.

11.6 Forward defined functions

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

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

```
end.
```

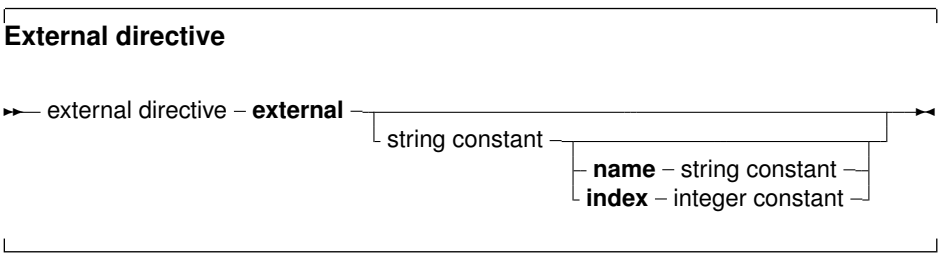
A function can be defined as forward only once. Likewise, in units, it is not allowed to have a forward declared function of a function that has been declared in the interface part. The interface declaration counts as a `forward` declaration. The following unit will give an error when compiled:

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

Reversely, functions declared in the interface section cannot be declared forward in the implementation section. Logically, since they already have been declared.

11.7 External functions

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



It replaces, in effect, the function or procedure code block. As an example:

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

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

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

```
external 'lname';
```

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

The name that the function has in the library can also be specified:

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

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

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

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

Finally, the external directive can be used to specify the external name of the function :

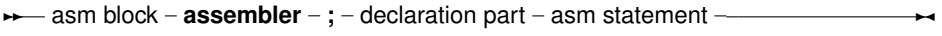
```
external name 'Fname';
{$L myfunc.o}
```

This tells the compiler that the function has the name 'Fname'. The correct library or object file (in this case myfunc.o) must still be linked, ensuring that the function 'Fname' is indeed included in the linking stage.

11.8 Assembler functions

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

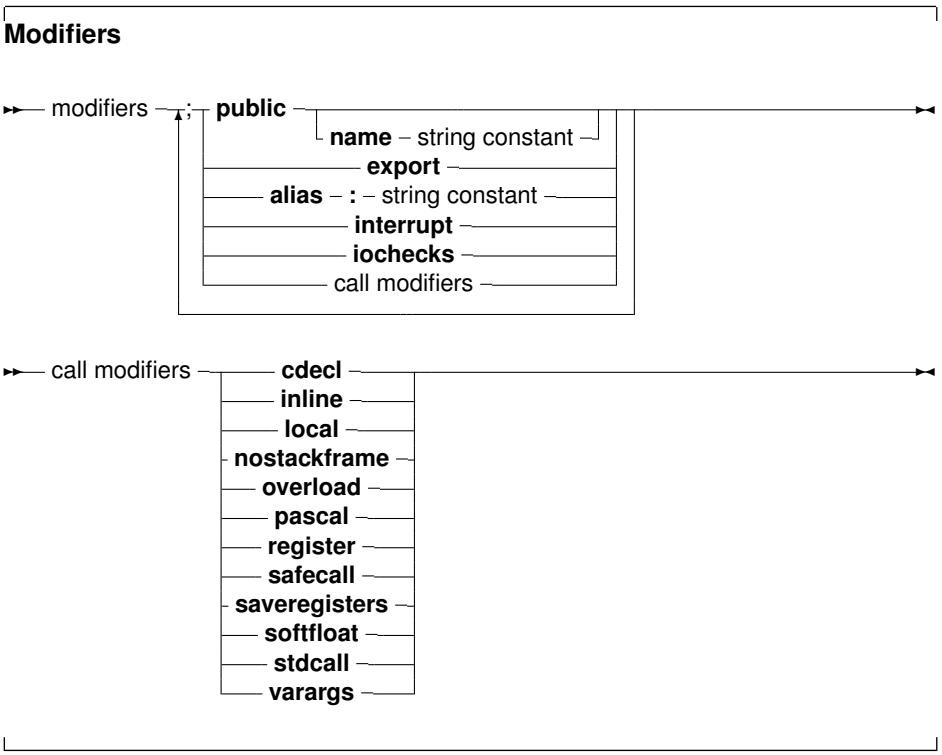
Assembler functions



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

11.9 Modifiers

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



Free Pascal doesn't support all Turbo Pascal modifiers (although it parses them for compatibility), but does support a number of additional modifiers. They are used mainly for assembler and reference to C object files.

11.9.1 alias

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

```
Program Aliases;

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

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

The `alias` modifier does not make the symbol public to other modules, unless the routine is also declared in the interface part of a unit, or the `public` modifier is used to force it as public. Consider the following:

```
unit testalias;

interface

procedure testroutine;

implementation

procedure testroutine;alias:'ARoutine';
begin
  WriteLn('Hello world');
end;

end.
```

This will make the routine `testroutine` available publicly to external object files under the label name `ARoutine`.

Remark: The `alias` directive is considered deprecated. Please use the `public name` directive. See section 11.9.11, page 143.

11.9.2 cdecl

The `cdecl` modifier can be used to declare a function that uses a C type calling convention. This must be used when accessing functions residing in an object file generated by standard C compilers, but must also be used for Pascal functions that are to be used as callbacks for C libraries.

The `cdecl` modifier allows to use C function in the code. For external C functions, the object file containing the `C` implementation of the function or procedure must be linked in. As an example:

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

When compiling this, and linking to the C-library, the `strlen` function can be called throughout the program. The `external` directive tells the compiler that the function resides in an external object file or library with the 'strlen' name (see 11.7).

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

For functions that are not external, but which are declared using `cdecl`, no external linking is needed. These functions have some restrictions, for instance the `array of const` construct can not be used (due the the way this uses the stack). On the other hand, the `cdecl` modifier allows these functions to be used as callbacks for routines written in C, as the latter expect the 'cdecl' calling convention.

11.9.3 export

The `export` modifier is used to export names when creating a shared library or an executable program. This means that the symbol will be publicly available, and can be imported from other programs. For more information on this modifier, consult the section on Programming dynamic libraries in the Programmer's Guide.

11.9.4 inline

Procedures that are declared `inline` are copied to the places where they are called. This has the effect that there is no actual procedure call, the code of the procedure is just copied to where the procedure is needed, this results in faster execution speed if the function or procedure is used a lot. It is obvious that inlining large functions does not make sense.

By default, `inline` procedures are not allowed. Inline code must be enabled using the command-line switch `-Si` or `{$inline on}` directive.

Remark:

1. `inline` is only a hint for the compiler. This does *not* automatically mean that all calls are inlined; sometimes the compiler may decide that a function simply cannot be inlined, or that a particular call to the function cannot be inlined. If so, the compiler will emit a warning.
2. In old versions of Free Pascal, inline code was *not* exported from a unit. This meant that when calling an inline procedure from another unit, a normal procedure call will be performed. Only inside units, `Inline` procedures are really inlined. As of version 2.0.2, inline works accross units.
3. Recursive inline functions are not allowed. i.e. an inline function that calls itself is not allowed.

11.9.5 interrupt

The `interrupt` keyword is used to declare a routine which will be used as an interrupt handler. On entry to this routine, all the registers will be saved and on exit, all registers will be restored and an interrupt or trap return will be executed (instead of the normal return from subroutine instruction).

On platforms where a return from interrupt does not exist, the normal exit code of routines will be done instead. For more information on the generated code, consult the Programmer's Guide.

11.9.6 iocheck

The `iocheck` keyword is used to declare a routine which causes generation of I/O result checking code within a `{$IOCHECKS ON}` block whenever it is called.

The result is that if a call to this procedure is generated, the compiler will insert I/O checking code if the call is within a `{$IOCHECKS ON}` block.

This modifier is intended for RTL internal routines, not for use in applicaton code.

11.9.7 local

The `local` modifier allows the compiler to optimize the function: a local function cannot be in the interface section of a unit: it is always in the implementation section of the unit. From this it follows that the function cannot be exported from a library.

On Linux, the local directive results in some optimizations. On Windows, it has no effect. It was introduced for Kylix compatibility.

11.9.8 nostackframe

The `nostackframe` modifier can be used to tell the compiler it should not generate a stack frame for this procedure or function. By default, a stack frame is always generated for each procedure or function.

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

11.9.9 overload

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

There is only one case where the `overload` modifier is mandatory: if a function must be overloaded that resides in another unit. Both functions must be declared with the `overload` modifier: the `overload` modifier tells the compiler that it should continue looking for overloaded versions in other units.

The following example illustrates this. Take the first unit:

```
unit ua;

interface

procedure DoIt(A : String); overload;

implementation

procedure DoIt(A : String);

begin
  Writeln('ua.DoIt received ',A)
end;

end.
```

And a second unit, which contains an overloaded version:

```
unit ub;

interface

procedure DoIt(A : Integer); overload;
```

```
implementation

procedure DoIt(A : integer);

begin
  Writeln('ub.DoIt received ',A)
end;

end.
```

And the following program, which uses both units:

```
program uab;

uses ua,ub;

begin
  DoIt('Some string');
end.
```

When the compiler starts looking for the declaration of `DoIt`, it will find one in the ub unit. Without the `overload` directive, the compiler would give an argument mismatch error:

```
home: >fpc uab.pp
uab.pp(6,21) Error: Incompatible type for arg no. 1:
Got "Constant String", expected "SmallInt"
```

With the `overload` directive in place at both locations, the compiler knows it must continue searching for an overloaded version with matching parameter list. Note that *both* declarations must have the `overload` modifier specified; It is not enough to have the modifier in unit ub. This is to prevent unwanted overloading: The programmer who implemented the `ua` unit must mark the procedure as fit for overloading.

11.9.10 pascal

The `pascal` modifier can be used to declare a function that uses the classic Pascal type calling convention (passing parameters from left to right). For more information on the Pascal calling convention, consult the Programmer's Guide.

11.9.11 public

The `Public` keyword is used to declare a function globally in a unit. This is useful if the function should not be accessible from the unit file (i.e. another unit/program using the unit doesn't see the function), but must be accessible from the object file. as an example:

```
Unit someunit;
interface
Function First : Real;
Implementation
Function First : Real;
begin
  First := 0;
```

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

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

The `public` modifier can also be followed by a `name` directive to specify the assembler name, as follows:

```
Unit someunit;
interface
Function First : Real;
Implementation
Function First : Real;
begin
  First := 0;
end;
Function Second : Real; Public name 'second';
begin
  Second := 1;
end;
end.
```

The assembler symbol as specified by the 'public name' directive will be 'second', in all lowercase letters.

11.9.12 register

The `register` keyword is used for compatibility with Delphi. In version 1.0.x of the compiler, this directive has no effect on the generated code. As of the 1.9.X versions, this directive is supported. The first three arguments are passed in registers EAX,ECX and EDX.

11.9.13 safecall

The `safecall` modifier ressembles closely the `stdcall` modifier. It sends parameters from right to left on the stack. Additionally, the called procedure saves and restores all registers.

More information about this modifier can be found in the Programmer's Guide, in the section on the calling mechanism and the chapter on linking.

11.9.14 saveregisters

The `saveregisters` modifier tells the compiler that all CPU registers should be saved prior to calling this routine. Which CPU registers are saved, depends entirely on the CPU.

11.9.15 softfloat

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

11.9.16 stdcall

The `stdcall` modifier pushes the parameters from right to left on the stack, it also aligns all the parameters to a default alignment.

More information about this modifier can be found in the Programmer's Guide, in the section on the calling mechanism and the chapter on linking.

11.9.17 varargs

This modifier can only be used together with the `cdecl` modifier, for external C procedures. It indicates that the procedure accepts a variable number of arguments after the last declared variable. These arguments are passed on without any type checking. It is equivalent to using the `array of const` construction for `cdecl` procedures, without having to declare the `array of const`. The square brackets around the variable arguments do not need to be used when this form of declaration is used.

The following declarations are 2 ways of referring to the same function in the C library:

```
Function PrintF1(fmt : pchar); cdecl; varargs;
                               external 'c' name 'printf';
Function PrintF2(fmt : pchar; Args : Array of const); cdecl;
                               external 'c' name 'printf';
```

But they must be called differently:

```
PrintF1('%d %d\n',1,1);
PrintF2('%d %d\n',[1,1]);
```

11.10 Unsupported Turbo Pascal modifiers

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

Table 11.1: Unsupported modifiers

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

otherwise completely ignore them.

Appendix B

Alphabetical list of reserved words

```
absolute
abstract
and
array
as
asm
assembler
begin
break
case
cdecl
class
const
constructor
continue
cppclass
deprecated
destructor
div
do
downto
else
end
except
exit
export
exports
external
experimental
fail
false
far
file
finally
for
forward
function
goto
if
implementation
in
index
inherited
initialization
inline
interface
interrupt
is
label
library
mod
name
near
nil
not
object
of
on
operator
or
otherwise
packed
popstack
private
procedure
program
property
protected
public
raise
record
reintroduce
repeat
self
set
shl
shr
stdcall
string
then
to
true
try
type
unimplemented
unit
until
uses
var
virtual
while
with
xor
```

Appendix C

Compiler messages

This appendix is meant to list all the compiler messages. The list of messages is generated from he compiler source itself, and should be fairly complete. At this point, only assembler errors are not in the list.

For an explanation of how to control the messages, section 5.1.2, page 25.

C.1 General compiler messages

This section gives the compiler messages which are not fatal, but which display useful information. The number of such messages can be controlled with the various verbosity level `-v` switches.

Compiler: arg1 When the `-vt` switch is used, this line tells you what compiler is used.

Compiler OS: arg1 When the `-vd` switch is used, this line tells you what the source operating system is.

Info: Target OS: arg1 When the `-vd` switch is used, this line tells you what the target operating system is.

Using executable path: arg1 When the `-vt` switch is used, this line tells you where the compiler looks for its binaries.

Using unit path: arg1 When the `-vt` switch is used, this line tells you where the compiler looks for compiled units. You can set this path with the `-Fu` option.

Using include path: arg1 When the `-vt` switch is used, this line tells you where the compiler looks for its include files (files used in `{$I xxx}` statements). You can set this path with the `-Fi` option.

Using library path: arg1 When the `-vt` switch is used, this line tells you where the compiler looks for the libraries. You can set this path with the `-Fl` option.

Using object path: arg1 When the `-vt` switch is used, this line tells you where the compiler looks for object files you link in (files used in `{$L xxx}` statements). You can set this path with the `-Fo` option.

Info: arg1 lines compiled, arg2 sec arg3 When the `-vi` switch is used, the compiler reports the number of lines compiled, and the time it took to compile them (real time, not program time).

Fatal: No memory left The compiler doesn't have enough memory to compile your program. There are several remedies for this:

- If you're using the build option of the compiler, try compiling the different units manually.
- If you're compiling a huge program, split it up into units, and compile these separately.
- If the previous two don't work, recompile the compiler with a bigger heap. (You can use the `-Ch` option for this, `-Ch` (see page 28).)

Info: Writing Resource String Table file: arg1 This message is shown when the compiler writes the Resource String Table file containing all the resource strings for a program.

Error: Writing Resource String Table file: arg1 This message is shown when the compiler encounters an error when writing the Resource String Table file.

Info: Fatal: Prefix for Fatal Errors.

Info: Error: Prefix for Errors.

Info: Warning: Prefix for Warnings.

Info: Note: Prefix for Notes.

Info: Hint: Prefix for Hints.

Error: Path "arg1" does not exist The specified path does not exist.

Fatal: Compilation aborted Compilation was aborted.

bytes code The size of the generated executable code, in bytes.

bytes data The size of the generated program data, in bytes.

Info: arg1 warning(s) issued Total number of warnings issued during compilation.

Info: arg1 hint(s) issued Total number of hints issued during compilation.

Info: arg1 note(s) issued Total number of notes issued during compilation.

C.2 Scanner messages.

This section lists the messages that the scanner emits. The scanner takes care of the lexical structure of the pascal file, i.e. it tries to find reserved words, strings, etc. It also takes care of directives and conditional compilation handling.

Fatal: Unexpected end of file This typically happens in one of the following cases:

- The source file ends before the final `end.` statement. This happens mostly when the `begin` and `end` statements aren't balanced;
- An include file ends in the middle of a statement.
- A comment was not closed.

Fatal: String exceeds line There is a missing closing ' in a string, so it occupies multiple lines.

Fatal: illegal character "arg1" (arg2) An illegal character was encountered in the input file.

Fatal: Syntax error, "arg1" expected but "arg2" found This indicates that the compiler expected a different token than the one you typed. It can occur almost anywhere it is possible to make an error against the Pascal language.

Start reading includefile arg1 When you provide the `-vt` switch, the compiler tells you when it starts reading an included file.

Warning: Comment level arg1 found When the `-vw` switch is used, then the compiler warns you if it finds nested comments. Nested comments are not allowed in Turbo Pascal and Delphi, and can be a possible source of errors.

Note: Ignored compiler switch "arg1" With `-vn` on, the compiler warns if it ignores a switch.

Warning: Illegal compiler switch "arg1" You included a compiler switch (i.e. `{$...}`) which the compiler does not recognise.

Warning: Misplaced global compiler switch The compiler switch is misplaced, and should be located at the start of the unit or program.

Error: Illegal char constant This happens when you specify a character with its ASCII code, as in `#96`, but the number is either illegal, or out of range.

Fatal: Can't open file "arg1" Free Pascal cannot find the program or unit source file you specified on the command line.

Fatal: Can't open include file "arg1" Free Pascal cannot find the source file you specified in a `{$include ..}` statement.

Error: Illegal record alignment specifier "arg1" You are specifying `{$PACKRECORDS n}` or `{$ALIGN n}` with an illegal value for `n`. For $PACKRECORDS valid alignments are 1, 2, 4, 8, 16, 32, C, NORMAL, DEFAULT, and for $ALIGN valid alignments are 1, 2, 4, 8, 16, 32, ON, OFF. Under mode MacPas $ALIGN also supports MAC68K, POWER and RESET.

Error: Illegal enum minimum-size specifier "arg1" You are specifying the `{$PACKENUM n}` with an illegal value for `n`. Only 1,2,4, NORMAL or DEFAULT is valid here.

Error: $ENDIF expected for arg1 arg2 defined in arg3 line arg4 Your conditional compilation statements are unbalanced.

Error: Syntax error while parsing a conditional compiling expression There is an error in the expression following the `{$if ..}`, `{$ifc }` or `{$setc }` compiler directives.

Error: Evaluating a conditional compiling expression There is an error in the expression following the `{$if ..}`, *ifcor*setc compiler directives.

Warning: Macro contents are limited to 255 characters in length The contents of macros cannot be longer than 255 characters.

Error: ENDIF without IF(N)DEF Your `{$IFDEF ..}` and {$ENDIF} statements aren't balanced.

Fatal: User defined: arg1 A user defined fatal error occurred. See also the Programmer's Guide.

Error: User defined: arg1 A user defined error occurred. See also the Programmer's Guide.

Warning: User defined: arg1 A user defined warning occurred. See also the Programmer's Guide.

Note: User defined: arg1 A user defined note was encountered. See also the Programmer's Guide.

Hint: User defined: arg1 A user defined hint was encountered. See also the Programmer's Guide.

Info: User defined: arg1 User defined information was encountered. See also the Programmer's Guide.

Error: Keyword redefined as macro has no effect You cannot redefine keywords with macros.

Fatal: Macro buffer overflow while reading or expanding a macro Your macro or its result was too long for the compiler.

Warning: Expanding of macros exceeds a depth of 16. When expanding a macro, macros have been nested to a level of 16. The compiler will expand no further, since this may be a sign that recursion is used.

Warning: compiler switches aren't supported in // styled comments Compiler switches should be in normal Pascal style comments.

Handling switch "arg1" When you set debugging info on (`-vd`) the compiler tells you when it is evaluating conditional compile statements.

ENDIF arg1 found When you turn on conditional messages (`-vc`), the compiler tells you where it encounters conditional statements.

IFDEF arg1 found, arg2 When you turn on conditional messages (`-vc`), the compiler tells you where it encounters conditional statements.

IFOPT arg1 found, arg2 When you turn on conditional messages (`-vc`), the compiler tells you where it encounters conditional statements.

IF arg1 found, arg2 When you turn on conditional messages (`-vc`), the compiler tells you where it encounters conditional statements.

IFNDEF arg1 found, arg2 When you turn on conditional messages (`-vc`), the compiler tells you where it encounters conditional statements.

ELSE arg1 found, arg2 When you turn on conditional messages (`-vc`), the compiler tells you where it encounters conditional statements.

Skipping until... When you turn on conditional messages (`-vc`), the compiler tells you where it encounters conditional statements, and whether it is skipping or compiling parts.

Info: Press <return> to continue When the `-vi` switch is used, the compiler stops compilation and waits for the `Enter` key to be pressed when it encounters a `{$STOP}` directive.

Warning: Unsupported switch "arg1" When warnings are turned on (`-vw`), the compiler warns you about unsupported switches. This means that the switch is used in Delphi or Turbo Pascal, but not in Free Pascal.

Warning: Illegal compiler directive "arg1" When warnings are turned on (`-vw`), the compiler warns you about unrecognised switches. For a list of recognised switches, see the Programmer's Guide.

Back in arg1 When you use the `-vt` switch, the compiler tells you when it has finished reading an include file.

Warning: Unsupported application type: "arg1" You get this warning if you specify an unknown application type with the directive `{$APPTYPE}`.

Warning: APPTYPE is not supported by the target OS The `{$APPTYPE}` directive is supported by certain operating systems only.

Warning: DESCRIPTION is not supported by the target OS The `{$DESCRIPTION}` directive is not supported on this target OS.

Note: VERSION is not supported by target OS The `{$VERSION}` directive is not supported on this target OS.

Note: VERSION only for exes or DLLs The `{$VERSION}` directive is only used for executable or DLL sources.

Warning: Wrong format for VERSION directive "arg1" The `{$VERSION}` directive format is majorversion.minorversion where majorversion and minorversion are words.

Error: Illegal assembler style specified "arg1" When you specify an assembler mode with the `{$ASMMODE xxx}` directive, the compiler didn't recognize the mode you specified.

Warning: ASM reader switch is not possible inside asm statement, "arg1" will be effective only for next It is not possible to switch from one assembler reader to another inside an assembler block. The new reader will be used for next assembler statements only.

Error: Wrong switch toggle, use ON/OFF or +/- You need to use ON or OFF or a + or - to toggle the switch.

Error: Resource files are not supported for this target The target you are compiling for doesn't support resource files.

Warning: Include environment "arg1" not found in environment The included environment variable can't be found in the environment; it will be replaced by an empty string instead.

Error: Illegal value for FPU register limit Valid values for this directive are 0..8 and NORMAL/DEFAULT.

Warning: Only one resource file is supported for this target The target you are compiling for supports only one resource file. The first resource file found is used, the others are discarded.

Warning: Macro support has been turned off A macro declaration has been found, but macro support is currently off, so the declaration will be ignored. To turn macro support on compile with -Sm on the command line or add {$MACRO ON} in the source.

Error: Illegal interface type specified. Valids are COM, CORBA or DEFAULT. The interface type that was specified is not supported.

Warning: APPID is only supported for PalmOS The `{$APPID}` directive is only supported for the PalmOS target.

Warning: APPNAME is only supported for PalmOS The `{$APPNAME}` directive is only supported for the PalmOS target.

Error: Constant strings can't be longer than 255 chars A single string constant can contain at most 255 chars. Try splitting up the string into multiple smaller parts and concatenate them with a + operator.

Fatal: Including include files exceeds a depth of 16. When including include files the files have been nested to a level of 16. The compiler will expand no further, since this may be a sign that recursion is used.

Fatal: Too many levels of PUSH A maximum of 20 levels is allowed. This error occurs only in mode MacPas.

Error: A POP without a preceding PUSH This error occurs only in mode MacPas.

Error: Macro or compile time variable "arg1" does not have any value Thus the conditional compile time expression cannot be evaluated.

Error: Wrong switch toggle, use ON/OFF/DEFAULT or +/-/* You need to use ON or OFF or DEFAULT or a + or - or * to toggle the switch.

Error: Mode switch "arg1" not allowed here A mode switch has already been encountered, or, in the case of option -Mmacpas, a mode switch occurs after UNIT.

Error: Compile time variable or macro "arg1" is not defined. Thus the conditional compile time expression cannot be evaluated. Only in mode MacPas.

Error: UTF-8 code greater than 65535 found Free Pascal handles UTF-8 strings internally as widestrings, i.e. the char codes are limited to 65535.

Error: Malformed UTF-8 string The given string isn't a valid UTF-8 string.

UTF-8 signature found, using UTF-8 encoding The compiler found a UTF-8 encoding signature ($ef, $bb, $bf) at the beginning of a file, so it interprets it as a UTF-8 file.

Error: Compile time expression: Wanted arg1 but got arg2 at arg3 The type-check of a compile time expression failed.

Note: APPTYPE is not supported by the target OS The `{$APPTYPE}` directive is supported by certain operating systems only.

Error: Illegal optimization specified "arg1" You specified an optimization with the `{$OPTIMIZATION xxx}` directive, and the compiler didn't recognize the optimization you specified.

Warning: SETPEFLAGS is not supported by the target OS The `{$SETPEFLAGS}` directive is not supported by the target OS.

Warning: IMAGEBASE is not supported by the target OS The `{$IMAGEBASE}` directive is not supported by the target OS.

Warning: MINSTACKSIZE is not supported by the target OS The `{$MINSTACKSIZE}` directive is not supported by the target OS.

Warning: MAXSTACKSIZE is not supported by the target OS The `{$MAXSTACKSIZE}` directive is not supported by the target OS.

Error: Illegal state for $WARN directive Only ON and OFF can be used as state with a `{$WARN}` compiler directive.

Error: Illegal set packing value Only 0, 1, 2, 4, 8, DEFAULT and NORMAL are allowed as packset parameters.

Warning: PIC directive or switch ignored Several targets, such as WINDOWS, do not support nor need PIC, so the PIC directive and switch are ignored.

Warning: The switch "arg1" is not supported by the currently selected target Some compiler switches like $E are not supported by all targets.

Warning: Framework-related options are only supported for Darwin/Mac OS X Frameworks are not a known concept, or at least not supported by FPC, on operating systems other than Darwin/Mac OS X.

Error: Illegal minimal floating point constant precision "arg1" Valid minimal precisions for floating point constants are default, 32 and 64, which mean respectively minimal (usually 32 bit), 32 bit and 64 bit precision.

Warning: Overriding name of "main" procedure multiple times, was previously set to "arg1" The name for the main entry procedure is specified more than once. Only the last name will be used.

C.3 Parser messages

This section lists all parser messages. The parser takes care of the semantics of you language, i.e. it determines if your Pascal constructs are correct.

Error: Parser - Syntax Error An error against the Turbo Pascal language was encountered. This typically happens when an illegal character is found in the source file.

Error: INTERRUPT procedure can't be nested An `INTERRUPT` procedure must be global.

Warning: Procedure type "arg1" ignored The specified procedure directive is ignored by FPC programs.

Error: Not all declarations of "arg1" are declared with OVERLOAD When you want to use overloading using the `OVERLOAD` directive, then all declarations need to have `OVERLOAD` specified.

Error: Duplicate exported function name "arg1" Exported function names inside a specific DLL must all be different.

Error: Duplicate exported function index arg1 Exported function indexes inside a specific DLL must all be different.

Error: Invalid index for exported function DLL function index must be in the range `1..$FFFF`.

Warning: Relocatable DLL or executable arg1 debug info does not work, disabled. It is currently not possible to include debug information in a relocatable DLL.

Warning: To allow debugging for win32 code you need to disable relocation with -WN option Stabs debug info is wrong for relocatable DLL or EXES. Use -WN if you want to debug win32 executables.

Error: Constructor name must be INIT You are declaring an object constructor with a name which is not `init`, and the `-Ss` switch is in effect. See the switch `-Ss` (see page 33).

Error: Destructor name must be DONE You are declaring an object destructor with a name which is not `done`, and the `-Ss` switch is in effect. See the switch `-Ss` (see page 33).

Error: Procedure type INLINE not supported You tried to compile a program with C++ style inlining, and forgot to specify the `-Si` option (`-Si` (see page 32)). The compiler doesn't support C++ styled inlining by default.

Warning: Constructor should be public Constructors must be in the 'public' part of an object (class) declaration.

Warning: Destructor should be public Destructors must be in the 'public' part of an object (class) declaration.

Note: Class should have one destructor only You can declare only one destructor for a class.

Error: Local class definitions are not allowed Classes must be defined globally. They cannot be defined inside a procedure or function.

Fatal: Anonymous class definitions are not allowed An invalid object (class) declaration was encountered, i.e. an object or class without methods that isn't derived from another object or class. For example:

```
Type o = object
         a : longint;
         end;
```

will trigger this error.

Note: The object "arg1" has no VMT This is a note indicating that the declared object has no virtual method table.

Error: Illegal parameter list You are calling a function with parameters that are of a different type than the declared parameters of the function.

Error: Wrong number of parameters specified for call to "arg1" There is an error in the parameter list of the function or procedure – the number of parameters is not correct.

Error: overloaded identifier "arg1" isn't a function The compiler encountered a symbol with the same name as an overloaded function, but it is not a function it can overload.

Error: overloaded functions have the same parameter list You're declaring overloaded functions, but with the same parameter list. Overloaded function must have at least 1 different parameter in their declaration.

Error: function header doesn't match the previous declaration "arg1" You declared a function with the same parameters but different result type or function modifiers.

Error: function header "arg1" doesn't match forward : var name changes arg2 => arg3 You declared the function in the `interface` part, or with the `forward` directive, but defined it with a different parameter list.

Note: Values in enumeration types have to be ascending Free Pascal allows enumeration constructions as in C. Examine the following two declarations:

```
type a = (A_A,A_B,A_E:=6,A_UAS:=200);
type a = (A_A,A_B,A_E:=6,A_UAS:=4);
```

The second declaration would produce an error. `A_UAS` needs to have a value higher than `A_E`, i.e. at least 7.

Error: With cannot be used for variables in a different segment With stores a variable locally on the stack, but this is not possible if the variable belongs to another segment.

Error: function nesting > 31 You can nest function definitions only 31 levels deep.

Error: range check error while evaluating constants The constants are out of their allowed range.

Warning: range check error while evaluating constants The constants are out of their allowed range.

Error: duplicate case label You are specifying the same label 2 times in a `case` statement.

Error: Upper bound of case range is less than lower bound The upper bound of a `case` label is less than the lower bound and this is useless.

Error: typed constants of classes or interfaces are not allowed You cannot declare a constant of type class or object.

Error: functions variables of overloaded functions are not allowed You are trying to assign an overloaded function to a procedural variable. This is not allowed.

Error: string length must be a value from 1 to 255 The length of a shortstring in Pascal is limited to 255 characters. You are trying to declare a string with length less than 1 or greater than 255.

Warning: use extended syntax of NEW and DISPOSE for instances of objects If you have a pointer `a` to an object type, then the statement `new(a)` will not initialize the object (i.e. the constructor isn't called), although space will be allocated. You should issue the `new(a,init)` statement. This will allocate space, and call the constructor of the object.

Warning: use of NEW or DISPOSE for untyped pointers is meaningless

Error: use of NEW or DISPOSE is not possible for untyped pointers You cannot use `new(p)` or `dispose(p)` if `p` is an untyped pointer because no size is associated to an untyped pointer. It is accepted for compatibility in `TP` and `DELPHI` modes, but the compiler will still warn you if it finds such a construct.

Error: class identifier expected This happens when the compiler scans a procedure declaration that contains a dot, i.e., an object or class method, but the type in front of the dot is not a known type.

Error: type identifier not allowed here You cannot use a type inside an expression.

Error: method identifier expected This identifier is not a method. This happens when the compiler scans a procedure declaration that contains a dot, i.e., an object or class method, but the procedure name is not a procedure of this type.

Error: function header doesn't match any method of this class "arg1" This identifier is not a method. This happens when the compiler scans a procedure declaration that contains a dot, i.e., an object or class method, but the procedure name is not a procedure of this type.

procedure/function arg1 When using the `-vd` switch, the compiler tells you when it starts processing a procedure or function implementation.

Error: Illegal floating point constant The compiler expects a floating point expression, and gets something else.

Error: FAIL can be used in constructors only You are using the `fail` keyword outside a constructor method.

Error: Destructors can't have parameters You are declaring a destructor with a parameter list. Destructor methods cannot have parameters.

Error: Only class methods can be referred with class references This error occurs in a situation like the following:

```
Type :
   Tclass = Class of Tobject;

Var C : TClass;

begin
...
C.free
```

`Free` is not a class method and hence cannot be called with a class reference.

Error: Only class methods can be accessed in class methods This is related to the previous error. You cannot call a method of an object from inside a class method. The following code would produce this error:

```
class procedure tobject.x;

begin
  free
```

Because free is a normal method of a class it cannot be called from a class method.

Error: Constant and CASE types do not match One of the labels is not of the same type as the case variable.

Error: The symbol can't be exported from a library You can only export procedures and functions when you write a library. You cannot export variables or constants.

Warning: An inherited method is hidden by "arg1" A method that is declared `virtual` in a parent class, should be overridden in the descendant class with the `override` directive. If you don't specify the `override` directive, you will hide the parent method; you will not override it.

Error: There is no method in an ancestor class to be overridden: "arg1" You are trying to `override` a virtual method of a parent class that does not exist.

Error: No member is provided to access property You specified no `read` directive for a property.

Warning: Stored property directive is not yet implemented This message is no longer used, as the `stored` directive has been implemented.

Error: Illegal symbol for property access There is an error in the `read` or `write` directives for an array property. When you declare an array property, you can only access it with procedures and functions. The following code would cause such an error.

```
tmyobject = class
  i : integer;
  property x [i : integer]: integer read I write i;
```

Error: Cannot access a protected field of an object here Fields that are declared in a `protected` section of an object or class declaration cannot be accessed outside the module where the object is defined, or outside descendent object methods.

Error: Cannot access a private field of an object here Fields that are declared in a `private` section of an object or class declaration cannot be accessed outside the module where the class is defined.

Error: Overridden methods must have the same return type: "arg2" is overriden by "arg1" which has another return
If you declare overridden methods in a class definition, they must have the same return type.

Error: EXPORT declared functions can't be nested You cannot declare a function or procedure within a function or procedure that was declared as an export procedure.

Error: Methods can't be EXPORTed You cannot declare a procedure that is a method for an object as `export`ed.

Error: Call by var for arg no. arg1 has to match exactly: Got "arg2" expected "arg3" When calling a function declared with `var` parameters, the variables in the function call must be of exactly the same type. There is no automatic type conversion.

Error: Class isn't a parent class of the current class When calling inherited methods, you are trying to call a method of a non-related class. You can only call an inherited method of a parent class.

Error: SELF is only allowed in methods You are trying to use the `self` parameter outside an object's method. Only methods get passed the `self` parameters.

Error: Methods can be only in other methods called direct with type identifier of the class A construction like `sometype.somemethod` is only allowed in a method.

Error: Illegal use of ':' You are using the format `:` (colon) 2 times on an expression that is not a real expression.

Error: range check error in set constructor or duplicate set element The declaration of a set contains an error. Either one of the elements is outside the range of the set type, or two of the elements are in fact the same.

Error: Pointer to object expected You specified an illegal type in a `new` statement. The extended syntax of `new` needs an object as a parameter.

Error: Expression must be constructor call When using the extended syntax of `new`, you must specify the constructor method of the object you are trying to create. The procedure you specified is not a constructor.

Error: Expression must be destructor call When using the extended syntax of `dispose`, you must specify the destructor method of the object you are trying to dispose of. The procedure you specified is not a destructor.

Error: Illegal order of record elements When declaring a constant record, you specified the fields in the wrong order.

Error: Expression type must be class or record type A `with` statement needs an argument that is of the type `record` or `class`. You are using `with` on an expression that is not of this type.

Error: Procedures can't return a value In Free Pascal, you can specify a return value for a function when using the `exit` statement. This error occurs when you try to do this with a procedure. Procedures cannot return a value.

Error: constructors and destructors must be methods You're declaring a procedure as destructor or constructor, when the procedure isn't a class method.

Error: Operator is not overloaded You're trying to use an overloaded operator when it is not overloaded for this type.

Error: Impossible to overload assignment for equal types You cannot overload assignment for types that the compiler considers as equal.

Error: Impossible operator overload The combination of operator, arguments and return type are incompatible.

Error: Re-raise isn't possible there You are trying to re-raise an exception where it is not allowed. You can only re-raise exceptions in an `except` block.

Error: The extended syntax of new or dispose isn't allowed for a class You cannot generate an instance of a class with the extended syntax of `new`. The constructor must be used for that. For the same reason, you cannot call `dispose` to de-allocate an instance of a class, the destructor must be used for that.

Error: Procedure overloading is switched off When using the `-So` switch, procedure overloading is switched off. Turbo Pascal does not support function overloading.

Error: It is not possible to overload this operator. Related overloadable operators (if any) are: arg1 You are trying to overload an operator which cannot be overloaded. The following operators can be overloaded :

```
+, -, *, /, =, >, <, <=, >=, is, as, in, **, :=
```

Error: Comparative operator must return a boolean value When overloading the = operator, the function must return a boolean value.

Error: Only virtual methods can be abstract You are declaring a method as abstract, when it is not declared to be virtual.

Fatal: Use of unsupported feature! You're trying to force the compiler into doing something it cannot do yet.

Error: The mix of different kind of objects (class, object, interface, etc) isn't allowed You cannot derive `objects`, `classes`, `cppclasses` and `interfaces` intertwined. E.g. a class cannot have an object as parent and vice versa.

Warning: Unknown procedure directive had to be ignored: "arg1" The procedure directive you specified is unknown.

Error: absolute can only be associated to one variable You cannot specify more than one variable before the `absolute` directive. Thus, the following construct will provide this error:

```
Var Z : Longint;
    X,Y : Longint absolute Z;
```

Error: absolute can only be associated with a var or const The address of an `absolute` directive can only point to a variable or constant. Therefore, the following code will produce this error:

```
Procedure X;

var p : longint absolute x;
```

Error: Only one variable can be initialized You cannot specify more than one variable with a initial value in Delphi mode.

Error: Abstract methods shouldn't have any definition (with function body) Abstract methods can only be declared, you cannot implement them. They should be overridden by a descendant class.

Error: This overloaded function can't be local (must be exported) You are defining an overloaded function in the implementation part of a unit, but there is no corresponding declaration in the interface part of the unit.

Warning: Virtual methods are used without a constructor in "arg1" If you declare objects or classes that contain virtual methods, you need to have a constructor and destructor to initialize them. The compiler encountered an object or class with virtual methods that doesn't have a constructor/destructor pair.

Macro defined: arg1 When `-vc` is used, the compiler tells you when it defines macros.

Macro undefined: arg1 When `-vc` is used, the compiler tells you when it undefines macros.

Macro arg1 set to arg2 When `-vc` is used, the compiler tells you what values macros get.

Info: Compiling arg1 When you turn on information messages (`-vi`), the compiler tells you what units it is recompiling.

Parsing interface of unit arg1 This tells you that the reading of the interface of the current unit has started

Parsing implementation of arg1 This tells you that the code reading of the implementation of the current unit, library or program starts

Compiling arg1 for the second time When you request debug messages (`-vd`) the compiler tells you what units it recompiles for the second time.

Error: No property found to override You want to override a property of a parent class, when there is, in fact, no such property in the parent class.

Error: Only one default property is allowed You specified a property as `Default`, but the class already has a default property, and a class can have only one default property.

Error: The default property must be an array property Only array properties of classes can be made `default` properties.

Error: Virtual constructors are only supported in class object model You cannot have virtual constructors in objects. You can only have them in classes.

Error: No default property available You are trying to access a default property of a class, but this class (or one of its ancestors) doesn't have a default property.

Error: The class can't have a published section, use the {$M+} switch If you want a `published` section in a class definition, you must use the `{$M+}` switch, which turns on generation of type information.

Error: Forward declaration of class "arg1" must be resolved here to use the class as ancestor To be able to use an object as an ancestor object, it must be defined first. This error occurs in the following situation:

```
Type ParentClas = Class;
     ChildClass = Class(ParentClass)
       ...
     end;
```

where `ParentClass` is declared but not defined.

Error: Local operators not supported You cannot overload locally, i.e. inside procedures or function definitions.

Error: Procedure directive "arg1" not allowed in interface section This procedure directive is not allowed in the `interface` section of a unit. You can only use it in the `implementation` section.

Error: Procedure directive "arg1" not allowed in implementation section This procedure directive is not allowed in the `implementation` section of a unit. You can only use it in the `interface` section.

Error: Procedure directive "arg1" not allowed in procvar declaration This procedure directive cannot be part of a procedural or function type declaration.

Error: Function is already declared Public/Forward "arg1" You will get this error if a function is defined as `forward` twice. Or if it occurs in the `interface` section, and again as a `forward` declaration in the `implementation` section.

Error: Can't use both EXPORT and EXTERNAL These two procedure directives are mutually exclusive.

Warning: "arg1" not yet supported inside inline procedure/function Inline procedures don't support this declaration.

Warning: Inlining disabled Inlining of procedures is disabled.

Info: Writing Browser log arg1 When information messages are on, the compiler warns you when it writes the browser log (generated with the `{$Y+ }` switch).

Hint: may be pointer dereference is missing The compiler thinks that a pointer may need a dereference.

Fatal: Selected assembler reader not supported The selected assembler reader (with `{$ASMMODE xxx}` is not supported. The compiler can be compiled with or without support for a particular assembler reader.

Error: Procedure directive "arg1" has conflicts with other directives You specified a procedure directive that conflicts with other directives. For instance `cdecl` and `pascal` are mutually exclusive.

Error: Calling convention doesn't match forward This error happens when you declare a function or procedure with e.g. `cdecl;` but omit this directive in the implementation, or vice versa. The calling convention is part of the function declaration, and must be repeated in the function definition.

Error: Property can't have a default value Set properties or indexed properties cannot have a default value.

Error: The default value of a property must be constant The value of a `default` declared property must be known at compile time. The value you specified is only known at run time. This happens e.g. if you specify a variable name as a default value.

Error: Symbol can't be published, can be only a class Only class type variables can be in a `published` section of a class if they are not declared as a property.

Error: This kind of property can't be published Properties in a `published` section cannot be array properties. They must be moved to public sections. Properties in a `published` section must be an ordinal type, a real type, strings or sets.

Error: An import name is required Some targets need a name for the imported procedure or a `cdecl` specifier.

Error: Division by zero A division by zero was encounted.

Error: Invalid floating point operation An operation on two real type values produced an overflow or a division by zero.

Error: Upper bound of range is less than lower bound The upper bound of an array declaration is less than the lower bound and this is not possible.

Warning: string "arg1" is longer than "arg2" The size of the constant string is larger than the size you specified in string type definition.

Error: string length is larger than array of char length The size of the constant string is larger than the size you specified in the `Array[x..y] of char` definition.

Error: Illegal expression after message directive Free Pascal supports only integer or string values as message constants.

Error: Message handlers can take only one call by ref. parameter A method declared with the `message` directive as message handler can take only one parameter which must be declared as call by reference. Parameters are declared as call by reference using the `var`-directive.

Error: Duplicate message label: "arg1" A label for a message is used twice in one object/class.

Error: Self can only be an explicit parameter in methods which are message handlers The `Self` parameter can only be passed explicitly to a method which is declared as message handler.

Error: Threadvars can be only static or global Threadvars must be static or global; you can't declare a thread local to a procedure. Local variables are always local to a thread, because every thread has its own stack and local variables are stored on the stack.

Fatal: Direct assembler not supported for binary output format You can't use direct assembler when using a binary writer. Choose an other output format or use another assembler reader.

Warning: Don't load OBJPAS unit manually, use \{\$mode objfpc\} or \{\$mode delphi\} instead You are trying to load the ObjPas unit manually from a `uses` clause. This is not a good idea. Use the `{$MODE OBJFPC}` or `{$mode delphi}` directives which load the unit automatically.

Error: OVERRIDE can't be used in objects `Override` is not supported for objects, use `virtual` instead to override a method of a parent object.

Error: Data types which require initialization/finalization can't be used in variant records Some data types (e.g. `ansistring`) need initialization/finalization code which is implicitly generated by the compiler. Such data types can't be used in the variant part of a record.

Error: Resourcestrings can be only static or global Resourcestring cannot be declared local, only global or using the static directive.

Error: Exit with argument can't be used here An exit statement with an argument for the return value can't be used here. This can happen for example in `try..except` or `try..finally` blocks.

Error: The type of the storage symbol must be boolean If you specify a storage symbol in a property declaration, it must be a boolean type.

Error: This symbol isn't allowed as storage symbol You can't use this type of symbol as storage specifier in property declaration. You can use only methods with the result type boolean, boolean class fields or boolean constants.

Error: Only classes which are compiled in $M+ mode can be published A class-typed field in the published section of a class can only be a class which was compiled in `{$M+}` or which is derived from such a class. Normally such a class should be derived from `TPersistent`.

Error: Procedure directive expected This error is triggered when you have a `{$Calling}` directive without a calling convention specified. It also happens when declaring a procedure in a const block and you used a ; after a procedure declaration which must be followed by a procedure directive. Correct declarations are:

```
const
  p : procedure;stdcall=nil;
  p : procedure stdcall=nil;
```

Error: The value for a property index must be of an ordinal type The value you use to index a property must be of an ordinal type, for example an integer or enumerated type.

Error: Procedure name too short to be exported The length of the procedure/function name must be at least 2 characters long. This is because of a bug in dlltool which doesn't parse the .def file correctly with a name of length 1.

Error: No DEFFILE entry can be generated for unit global vars

Error: Compile without -WD option You need to compile this file without the -WD switch on the command line.

Fatal: You need ObjFpc (-S2) or Delphi (-Sd) mode to compile this module You need to use `{$MODE OBJFPC}` or `{$MODE DELPHI}` to compile this file. Or use the corresponding command line switch, either `-Mobjfpc` or `-MDelphi`.

Error: Can't export with index under arg1 Exporting of functions or procedures with a specified index is not supported on this target.

Error: Exporting of variables is not supported under arg1 Exporting of variables is not supported on this target.

Error: Improper GUID syntax The GUID indication does not have the proper syntax. It should be of the form

```
{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}
```

Where each `X` represents a hexadecimal digit.

Warning: Procedure named "arg1" not found that is suitable for implementing the arg2.arg3 The compiler cannot find a suitable procedure which implements the given method of an interface. A procedure with the same name is found, but the arguments do not match.

Error: interface identifier expected This happens when the compiler scans a `class` declaration that contains `interface` function name mapping code like this:

```
type
  TMyObject = class(TObject, IDispatch)
    function IUnknown.QueryInterface=MyQueryInterface;
    ....
```

and the `interface` before the dot is not listed in the inheritance list.

Error: Type "arg1" can't be used as array index type Types like `qword` or `int64` aren't allowed as array index type.

Error: Con- and destructors aren't allowed in interfaces Constructor and destructor declarations aren't allowed in interfaces. In the most cases method `QueryInterface` of `IUnknown` can be used to create a new interface.

Error: Access specifiers can't be used in INTERFACES The access specifiers `public`, `private`, `protected` and `published` can't be used in interfaces because all methods of an interface must be public.

Error: An interface can't contain fields Declarations of fields aren't allowed in interfaces. An interface can contain only methods and properties with method read/write specifiers.

Error: Can't declare local procedure as EXTERNAL Declaring local procedures as external is not possible. Local procedures get hidden parameters that will make the chance of errors very high.

Warning: Some fields coming before "arg1" weren't initialized In Delphi mode, not all fields of a typed constant record have to be initialized, but the compiler warns you when it detects such situations.

Error: Some fields coming before "arg1" weren't initialized In all syntax modes but Delphi mode, you can't leave some fields uninitialized in the middle of a typed constant record.

Warning: Some fields coming after "arg1" weren't initialized You can leave some fields at the end of a type constant record uninitialized (The compiler will initialize them to zero automatically). This may be the cause of subtle problems.

Error: VarArgs directive (or '...' in MacPas) without CDecl/CPPDecl/MWPascal and External The varargs directive (or the "..." varargs parameter in MacPas mode) can only be used with procedures or functions that are declared with `external` and one of `cdecl`, `cppdecl` and `mwpascal`. This functionality is only supported to provide a compatible interface to C functions like printf.

Error: Self must be a normal (call-by-value) parameter You can't declare `Self` as a const or var parameter, it must always be a call-by-value parameter.

Error: Interface "arg1" has no interface identification When you want to assign an interface to a constant, then the interface must have a GUID value set.

Error: Unknown class field or method identifier "arg1" Properties must refer to a field or method in the same class.

Warning: Overriding calling convention "arg1" with "arg2" There are two directives in the procedure declaration that specify a calling convention. Only the last directive will be used.

Error: Typed constants of the type "procedure of object" can only be initialized with NIL You can't assign the address of a method to a typed constant which has a 'procedure of object' type, because such a constant requires two addresses: that of the method (which is known at compile time) and that of the object or class instance it operates on (which cannot be known at compile time).

Error: Default value can only be assigned to one parameter It is not possible to specify a default value for several parameters at once. The following is invalid:

```
Procedure MyProcedure (A,B : Integer = 0);
```

Instead, this should be declared as

```
Procedure MyProcedure (A : Integer = 0; B : Integer = 0);
```

Error: Default parameter required for "arg1" The specified parameter requires a default value.

Warning: Use of unsupported feature! You're trying to force the compiler into doing something it cannot do yet.

Hint: C arrays are passed by reference Any array passed to a C function is passed by a pointer (i.e. by reference).

Error: C array of const must be the last argument You cannot add any other argument after an `array of const` for `cdecl` functions, as the size pushed on stack for this argument is not known.

Hint: Type "arg1" redefinition This is an indicator that a previously declared type is being redefined as something else. This may, or may not be, a potential source of errors.

Warning: cdecl'ared functions have no high parameter Functions declared with the `cdecl` modifier do not pass an extra implicit parameter.

Warning: cdecl'ared functions do not support open strings Openstring is not supported for functions that have the `cdecl` modifier.

Error: Cannot initialize variables declared as threadvar Variables declared as threadvar cannot be initialized with a default value. The variables will always be filled with zero at the start of a new thread.

Error: Message directive is only allowed in Classes The message directive is only supported for Class types.

Error: Procedure or Function expected A class method can only be specified for procedures and functions.

Warning: Calling convention directive ignored: "arg1" Some calling conventions are supported only by certain CPUs. I.e. most non-i386 ports support only the standard ABI calling convention of the CPU.

Error: REINTRODUCE can't be used in objects `reintroduce` is not supported for objects.

Error: Each argument must have its own location If locations for arguments are specified explicitly as it is required by some syscall conventions, each argument must have its own location. Things like

```
procedure p(i,j : longint 'r1');
```

aren't allowed.

Error: Each argument must have an explicit location If one argument has an explicit argument location, all arguments of a procedure must have one.

Error: Unknown argument location The location specified for an argument isn't recognized by the compiler.

Error: 32 Bit-Integer or pointer variable expected The libbase for MorphOS/AmigaOS can be given only as `longint`, `dword` or any pointer variable.

Error: Goto statements aren't allowed between different procedures It isn't allowed to use `goto` statements referencing labels outside the current procedure. The following example shows the problem:

```
...
  procedure p1;
  label
    l1;

    procedure p2;
    begin
      goto l1; // This goto ISN'T allowed
    end;

  begin
    p2
  l1:
  end;
...
```

Fatal: Procedure too complex, it requires too many registers Your procedure body is too long for the compiler. You should split the procedure into multiple smaller procedures.

Error: Illegal expression This can occur under many circumstances. Usually when trying to evaluate constant expressions.

Error: Invalid integer expression You made an expression which isn't an integer, and the compiler expects the result to be an integer.

Error: Illegal qualifier One of the following is happening :

- You're trying to access a field of a variable that is not a record.
- You're indexing a variable that is not an array.
- You're dereferencing a variable that is not a pointer.

Error: High range limit < low range limit You are declaring a subrange, and the high limit is less than the low limit of the range.

Error: Exit's parameter must be the name of the procedure it is used in Non local exit is not allowed. This error occurs only in mode MacPas.

Error: Illegal assignment to for-loop variable "arg1" The type of a `for` loop variable must be an ordinal type. Loop variables cannot be reals or strings. You also cannot assign values to loop variables inside the loop (Except in Delphi and TP modes). Use a while or repeat loop instead if you need to do something like that, since those constructs were built for that.

Error: Can't declare local variable as EXTERNAL Declaring local variables as external is not allowed. Only global variables can reference external variables.

Error: Procedure is already declared EXTERNAL The procedure is already declared with the EXTERNAL directive in an interface or forward declaration.

Warning: Implicit uses of Variants unit The Variant type is used in the unit without any used unit using the Variants unit. The compiler has implicitly added the Variants unit to the uses list. To remove this warning the Variants unit needs to be added to the uses statement.

Error: Class and static methods can't be used in INTERFACES The specifier `class` and directive `static` can't be used in interfaces because all methods of an interface must be public.

Error: Overflow in arithmetic operation An operation on two integer values produced an overflow.

Error: Protected or private expected `strict` can be only used together with `protected` or `private`.

Error: SLICE can't be used outside of parameter list `slice` can be used only for arguments accepting an open array parameter.

Error: A DISPINTERFACE can't have a parent class A DISPINTERFACE is a special type of interface which can't have a parent class.

Error: A DISPINTERFACE needs a guid A DISPINTERFACE always needs an interface identification (a GUID).

Warning: Overridden methods must have a related return type. This code may crash, it depends on a Delphi parser bu If you declare overridden methods in a class definition, they must have the same return type. Some versions of Delphi allow you to change the return type of interface methods, and even to change procedures into functions, but the resulting code may crash depending on the types used and the way the methods are called.

Error: Dispatch IDs must be ordinal constants The `dispid` keyword must be followed by an ordinal constant (the dispid index).

Error: The range of the array is too large Regardless of the size taken up by its elements, an array cannot have more than high(ptrint) elements. Additionally, the range type must be a subrange of ptrint.

Error: The address cannot be taken of bit packed array elements and record fields If you declare an array or record as `packed` in Mac Pascal mode (or as `packed` in any mode with `{$bitpacking on}`), it will be packed at the bit level. This means it becomes impossible to take addresses of individual array elements or record fields. The only exception to this rule is in the case of packed arrays elements whose packed size is a multple of 8 bits.

Error: Dynamic arrays cannot be packed Only regular (and possibly in the future also open) arrays can be packed.

Error: Bit packed array elements and record fields cannot be used as loop variables If you declare an array or record as `packed` in Mac Pascal mode (or as `packed` in any mode with `{$bitpacking on}`), it will be packed at the bit level. For performance reasons, they cannot be used as loop variables.

Error: VAR and TYPE are allowed only in generics The usage of VAR and TYPE to declare new types inside an object is allowed only inside generics.

Error: This type can't be a generic Only Classes, Objects, Interfaces and Records are allowed to be used as generic.

Warning: Don't load LINEINFO unit manually, Use the -gl compiler switch instead Do not use the lineinfo unit directly, Use the `-gl` switch which automatically adds the correct unit for reading the selected type of debugging information. The unit that needs to be used depends on the type of debug information used when compiling the binary.

Error: No function result type specified for function "arg1" The first time you declare a function you have to declare it completely, including all parameters and the result type.

Error: Specialization is only supported for generic types Types which are not generics can't be specialized.

Error: Generics can't be used as parameters when specializing generics When specializing a generic, only non-generic types can be used as parameters.

Error: Constants of objects containing a VMT aren't allowed If an object requires a VMT either because it contains a constructor or virtual methods, it's not allowed to create constants of it. In TP and Delphi mode this is allowed for compatibility reasons.

Error: Taking the address of labels defined outside the current scope isn't allowed It isn't allowed to take the address of labels outside the current procedure.

Error: Cannot initialize variables declared as external Variables declared as external cannot be initialized with a default value.

Error: Illegal function result type Some types like file types cannot be used as function result.

Error: No common type possible between "arg1" and "arg2" To perform an operation on integers, the compiler converts both operands to their common type, which appears to be an invalid type. To determine the common type of the operands, the compiler takes the minimum of the minimal values of both types, and the maximum of the maximal values of both types. The common type is then minimum..maximum.

Error: Generics without specialization cannot be used as a type for a variable Generics must be always specialized before being used as variable type.

Warning: Register list is ignored for pure assembler routines When using pure assembler routines, the list with modified registers is ignored.

Error: Implements property must have class or interface type A property which implements an interface must be of type class or interface.

Error: Implements-property must implement interface of correct type, found "arg1" expected "arg2" A property which implements an interface actually implements a different interface.

Error: Implements-property must have read specifier A property which implements an interface must have at least a read specifier.

Error: Implements-property must not have write-specifier A property which implements an interface may not have a write specifier.

Error: Implements-property must not have stored-specifier A property which implements an interface may not have a stored specifier.

Error: Implements-property used on unimplemented interface: "arg1" The interface which is implemented by a property is not an interface implemented by the class.

Error: Floating point not supported for this target The compiler parsed a floating point expression, but it is not supported.

Error: Class "arg1" does not implement interface "arg2" The delegated interface is not implemented by the class given in the implements clause.

Error: Type used by implements must be an interface The `implements` keyword must be followed by an interface type.

Error: Variables cannot be exported with a different name on this target, add the name to the declaration using the "ex On most targets it is not possible to change the name under which a variable is exported inside the `exports` statement of a library. In that case, you have to specify the export name at the point where the variable is declared, using the `export` and `alias` directives.

Error: Weak external symbols are not supported for the current target A "weak external" symbol is a symbol which may or may not exist at (either static or dynamic) link time. This concept may not be available (or implemented yet) on the current cpu/OS target.

Error: Forward type definition does not match Classes and interfaces being defined forward must have the same type when being implemented. A forward interface cannot be changed into a class.

Note: Virtual method "arg1" has a lower visibility (arg2) than parent class arg3 (arg4) The virtual method overrides an method that is declared with a higher visibility. This might give unexpected results. In case the new visibility is private than it might be that a call to inherited in a new child class will call the higher visible method in a parent class and ignores the private method.

Error: Fields cannot appear after a method or property definition, start a new visibility section first Once a method or property has been defined in a class or object, you cannot define any fields afterwards without starting a new visibility section (such as `public`, `private`, etc.). The reason is that otherwise the source code can appear ambiguous to the compiler, since it is possible to use modifiers such as `default` and `register` also as field names.

Error: Parameters cannot contain local type definitions. Use a separate type definition in a type block. In Pascal, types are not considered to be identical simply because they are semantically equivalent. Two variables or parameters are only considered to be of the same type if they refer to the same type definition. As a result, it is not allowed to define new types inside parameter lists, because then it is impossible to refer to the same type definition in the procedure headers of the interface and implementation of a unit (both procedure headers would define a separate type). Keep in mind that expressions such as “file of byte” or “string[50]” also define a new type.

Error: ABSTRACT and SEALED conflict ABSTRACT and SEALED cannot be used together in one declaration

Error: Cannot create a descendant of the sealed class "arg1" Sealed means that class cannot be derived by another class.

Error: SEALED class cannot have an ABSTRACT method Sealed means that class cannot be derived. Therefore no one class is able to override an abstract method in a sealed class.

Error: Only virtual methods can be final You are declaring a method as final, when it is not declared to be virtual.

Error: Final method cannot be overridden: "arg1" You are trying to `override` a virtual method of a parent class that does not exist.

Error: Invalid enumerator identifier: "arg1" Only "MoveNext" and "Current" enumerator identifiers are supported.

Error: Enumerator identifier required "MoveNext" or "Current" identifier must follow the `enumerator` modifier.

Error: Enumerator MoveNext pattern method is not valid. Method must be a function with the Boolean return type an "MoveNext" enumerator pattern method must be a function with Boolean return type and no required arguments

Error: Enumerator Current pattern property is not valid. Property must have a getter. "Current" enumerator pattern property must have a getter

Error: Only one enumerator MoveNext method is allowed per class/object Class or Object can have only one enumerator MoveNext declaration.

Error: Only one enumerator Current property is allowed per class/object Class or Object can have only one enumerator Current declaration.

Error: For in loop cannot be used for the type "arg1" For in loop can be used not for all types. For example it cannot be used for the enumerations with jumps.

C.4 Type checking errors

This section lists all errors that can occur when type checking is performed.

Error: Type mismatch This can happen in many cases:

- The variable you're assigning to is of a different type than the expression in the assignment.
- You are calling a function or procedure with parameters that are incompatible with the parameters in the function or procedure definition.

Error: Incompatible types: got "arg1" expected "arg2" There is no conversion possible between the two types. Another possiblity is that they are declared in different declarations:

```
Var
   A1 : Array[1..10] Of Integer;
   A2 : Array[1..10] Of Integer;

Begin
   A1:=A2; { This statement also gives this error. It
             is due to the strict type checking of Pascal }
End.
```

Error: Type mismatch between "arg1" and "arg2" The types are not equal.

Error: Type identifier expected The identifier is not a type, or you forgot to supply a type identifier.

Error: Variable identifier expected This happens when you pass a constant to a routine (such as `Inc` var or `Dec`) when it expects a variable. You can only pass variables as arguments to these functions.

Error: Integer expression expected, but got "arg1" The compiler expects an expression of type integer, but gets a different type.

Error: Boolean expression expected, but got "arg1" The expression must be a boolean type. It should be return `True` or `False`.

Error: Ordinal expression expected The expression must be of ordinal type, i.e., maximum a `Longint`. This happens, for instance, when you specify a second argument to `Inc` or `Dec` that doesn't evaluate to an ordinal value.

Error: pointer type expected, but got "arg1" The variable or expression isn't of the type `pointer`. This happens when you pass a variable that isn't a pointer to `New` or `Dispose`.

Error: class type expected, but got "arg1" The variable or expression isn't of the type `class`. This happens typically when

1. The parent class in a class declaration isn't a class.
2. An exception handler (`On`) contains a type identifier that isn't a class.

Error: Can't evaluate constant expression This error can occur when the bounds of an array you declared do not evaluate to ordinal constants.

Error: Set elements are not compatible You are trying to perform an operation on two sets, when the set element types are not the same. The base type of a set must be the same when taking the union.

Error: Operation not implemented for sets several binary operations are not defined for sets. These include: `div`, `mod`, `**`, `>=` and `<=`. The last two may be defined for sets in the future.

Warning: Automatic type conversion from floating type to COMP which is an integer type An implicit type conversion from a real type to a `comp` is encountered. Since `comp` is a 64 bit integer type, this may indicate an error.

Hint: use DIV instead to get an integer result When hints are on, then an integer division with the '/' operator will produce this message, because the result will then be of type real.

Error: string types doesn't match, because of $V+ mode When compiling in `{$V+}` mode, the string you pass as a parameter should be of the exact same type as the declared parameter of the procedure.

Error: succ or pred on enums with assignments not possible If you declare an enumeration type which has C-like assignments in it, such as in the following:

```
Tenum = (a,b,e:=5);
```

then you cannot use the `Succ` or `Pred` functions with this enumeration.

Error: Can't read or write variables of this type You are trying to `read` or `write` a variable from or to a file of type text, which doesn't support that variable's type. Only integer types, reals, pchars and strings can be read from or written to a text file. Booleans can only be written to text files.

Error: Can't use readln or writeln on typed file `readln` and `writeln` are only allowed for text files.

Error: Can't use read or write on untyped file. `read` and `write` are only allowed for text or typed files.

Error: Type conflict between set elements There is at least one set element which is of the wrong type, i.e. not of the set type.

Warning: lo/hi(dword/qword) returns the upper/lower word/dword Free Pascal supports an overloaded version of `lo/hi` for `longint/dword/int64/qword` which returns the lower/upper word/dword of the argument. Turbo Pascal always uses a 16 bit `lo/hi` which always returns bits 0..7 for `lo` and the bits 8..15 for `hi`. If you want the Turbo Pascal behavior you have to type cast the argument to a `word` or `integer`.

Error: Integer or real expression expected The first argument to `str` must be a real or integer type.

Error: Wrong type "arg1" in array constructor You are trying to use a type in an array constructor which is not allowed.

Error: Incompatible type for arg no. arg1: Got "arg2", expected "arg3" You are trying to pass an invalid type for the specified parameter.

Error: Method (variable) and Procedure (variable) are not compatible You can't assign a method to a procedure variable or a procedure to a method pointer.

Error: Illegal constant passed to internal math function The constant argument passed to a `ln` or `sqrt` function is out of the definition range of these functions.

Error: Can't take the address of constant expressions It is not possible to get the address of a constant expression, because they aren't stored in memory. You can try making it a typed constant. This error can also be displayed if you try to pass a property to a var parameter.

Error: Argument can't be assigned to Only expressions which can be on the left side of an assignment can be passed as call by reference arguments.

Remark: Properties can be used on the left side of an assignment, nevertheless they cannot be used as arguments.

Error: Can't assign local procedure/function to procedure variable It's not allowed to assign a local procedure/function to a procedure variable, because the calling convention of a local procedure/function is different. You can only assign local procedure/function to a void pointer.

Error: Can't assign values to an address It is not allowed to assign a value to an address of a variable, constant, procedure or function. You can try compiling with -So if the identifier is a procedure variable.

Error: Can't assign values to const variable It's not allowed to assign a value to a variable which is declared as a const. This is normally a parameter declared as const. To allow changing the value, pass the parameter by value, or a parameter by reference (using var).

Error: Array type required If you are accessing a variable using an index '[<x>]' then the type must be an array. In FPC mode a pointer is also allowed.

Error: interface type expected, but got "arg1" The compiler expected to encounter an interface type name, but got something else. The following code would produce this error:

```
Type
  TMyStream = Class(TStream,Integer)
```

Hint: Mixing signed expressions and longwords gives a 64bit result If you divide (or calculate the modulus of) a signed expression by a longword (or vice versa), or if you have overflow and/or range checking turned on and use an arithmetic expression (+, -, *, div, mod) in which both signed numbers and longwords appear, then everything has to be evaluated in 64-bit arithmetic which is slower than normal 32-bit arithmetic. You can avoid this by typecasting one operand so it matches the result type of the other one.

Warning: Mixing signed expressions and cardinals here may cause a range check error If you use a binary operator (and, or, xor) and one of the operands is a longword while the other one is a signed expression, then, if range checking is turned on, you may get a range check error because in such a case both operands are converted to longword before the operation is carried out. You can avoid this by typecasting one operand so it matches the result type of the other one.

Error: Typecast has different size (arg1 -> arg2) in assignment Type casting to a type with a different size is not allowed when the variable is used in an assignment.

Error: enums with assignments can't be used as array index When you declared an enumeration type which has C-like assignments, such as in the following:

```
Tenum = (a,b,e:=5);
```

you cannot use it as the index of an array.

Error: Class or Object types "arg1" and "arg2" are not related There is a typecast from one class or object to another while the class/object are not related. This will probably lead to errors.

Warning: Class types "arg1" and "arg2" are not related There is a typecast from one class to another while the classes are not related. This will probably lead to errors.

Error: Class or interface type expected, but got "arg1" The compiler expected a class or interface name, but got another type or identifier.

Error: Type "arg1" is not completely defined This error occurs when a type is not complete: i.e. a pointer type which points to an undefined type.

Warning: String literal has more characters than short string length The size of the constant string, which is assigned to a shortstring, is longer than the maximum size of the shortstring (255 characters).

Warning: Comparison is always false due to range of values There is a comparison between an unsigned value and a signed constant which is less than zero. Because of type promotion, the statement will always evaluate to false. Explicitly typecast the constant to the correct range to avoid this problem.

Warning: Comparison is always true due to range of values There is a comparison between an unsigned value and a signed constant which is less than zero. Because of type promotion, the statement will always evaluate to true. Explicitly typecast the constant to the correct range to avoid this problem.

Warning: Constructing a class "arg1" with abstract method "arg2" An instance of a class is created which contains non-implemented abstract methods. This will probably lead to a runtime error 211 in the code if that routine is ever called. All abstract methods should be overriden.

Hint: The left operand of the IN operator should be byte sized The left operand of the `in` operator is not an ordinal or enumeration which fits within 8 bits. This may lead to range check errors. The `in` operator currently only supports a left operand which fits within a byte. In the case of enumerations, the size of an element of an enumeration can be controlled with the `{$PACKENUM}` or `{$Zn}` switches.

Warning: Type size mismatch, possible loss of data / range check error There is an assignment to a smaller type than the source type. This means that this may cause a range-check error, or may lead to possible loss of data.

Hint: Type size mismatch, possible loss of data / range check error There is an assignment to a smaller type than the source type. This means that this may cause a range-check error, or may lead to possible loss of data.

Error: The address of an abstract method can't be taken An abstract method has no body, so the address of an abstract method can't be taken.

Error: Assignments to formal parameters and open arrays are not possible You are trying to assign a value to a formal (untyped var, const or out) parameter, or to an open array.

Error: Constant Expression expected The compiler expects an constant expression, but gets a variable expression.

Error: Operation "arg1" not supported for types "arg2" and "arg3" The operation is not allowed for the supplied types.

Error: Illegal type conversion: "arg1" to "arg2" When doing a type-cast, you must take care that the sizes of the variable and the destination type are the same.

Hint: Conversion between ordinals and pointers is not portable If you typecast a pointer to a longint (or vice-versa), this code will not compile on a machine using 64 bits addressing.

Warning: Conversion between ordinals and pointers is not portable If you typecast a pointer to an ordinal type of a different size (or vice-versa), this can cause problems. This is a warning to help in finding the 32-bit specific code where cardinal/longint is used to typecast pointers to ordinals. A solution is to use the ptrint/ptruint types instead.

Error: Can't determine which overloaded function to call You're calling overloaded functions with a parameter that doesn't correspond to any of the declared function parameter lists. e.g. when you have declared a function with parameters `word` and `longint`, and then you call it with a parameter which is of type `integer`.

Error: Illegal counter variable The type of a `for` loop variable must be an ordinal type. Loop variables cannot be reals or strings.

Warning: Converting constant real value to double for C variable argument, add explicit typecast to prevent this. In C, constant real values are double by default. For this reason, if you pass a constant real value to a variable argument part of a C function, FPC by default converts this constant to double as well. If you want to prevent this from happening, add an explicit typecast around the constant.

Error: Class or COM interface type expected, but got "arg1" Some operators, such as the AS operator, are only applicable to classes or COM interfaces.

Error: Constant packed arrays are not yet supported You cannot declare a (bit)packed array as a typed constant.

Error: Incompatible type for arg no. arg1: Got "arg2" expected "(Bit)Packed Array" The compiler expects a (bit)packed array as the specified parameter.

Error: Incompatible type for arg no. arg1: Got "arg2" expected "(not packed) Array" The compiler expects a regular (i.e., not packed) array as the specified parameter.

Error: Elements of packed arrays cannot be of a type which need to be initialised Support for packed arrays of types that need initialization (such as ansistrings, or records which contain ansistrings) is not yet implemented.

Error: Constant packed records and objects are not yet supported You cannot declare a (bit)packed array as a typed constant at this time.

Warning: Arithmetic "arg1" on untyped pointer is unportable to {$T+}, suggest typecast Addition/subtraction from an untyped pointer may work differently in `{$T+}`. Use a typecast to a typed pointer.

Error: Can't take address of a subroutine marked as local The address of a subroutine marked as local can't be taken.

Error: Can't export subroutine marked as local from a unit A subroutine marked as local can't be exported from a unit.

Error: Type is not automatable: "arg1" Only byte, integer, longint, smallint, currency, single, double, ansistring, widestring, tdatetime, variant, olevariant, wordbool and all interfaces are automatable.

Hint: Converting the operands to "arg1" before doing the add could prevent overflow errors. Adding two types can cause overflow errors. Since you are converting the result to a larger type, you could prevent such errors by converting the operands to this type before doing the addition.

Hint: Converting the operands to "arg1" before doing the subtract could prevent overflow errors. Subtracting two types can cause overflow errors. Since you are converting the result to a larger type, you could prevent such errors by converting the operands to this type before doing the subtraction.

Hint: Converting the operands to "arg1" before doing the multiply could prevent overflow errors. Multiplying two types can cause overflow errors. Since you are converting the result to a larger type, you could prevent such errors by converting the operands to this type before doing the multiplication.

Warning: Converting pointers to signed integers may result in wrong comparison results and range errors, use an unsig The virtual address space on 32-bit machines runs from $00000000 to $ffffffff. Many operating systems allow you to allocate memory above $80000000. For example both WINDOWS and LINUX allow pointers in the range $0000000 to $bfffffff. If you convert pointers to signed types, this can cause overflow and range check errors, but also $80000000 < $7fffffff. This can cause random errors in code like "if p>q".

Error: Interface type arg1 has no valid GUID When applying the as-operator to an interface or class, the desired interface (i.e. the right operand of the as-operator) must have a valid GUID.

Error: Invalid selector name An Objective-C selector cannot be empty, must be a valid identifier or a single colon, and if it contains at least one colon it must also end in one.

Error: Expected Objective-C method, but got arg1 A selector can only be created for Objective-C methods, not for any other kind of procedure/function/method.

Error: Expected Objective-C method or constant method name A selector can only be created for Objective-C methods, either by specifying the name using a string constant, or by using an Objective-C method identifier that is visible in the current scope.

Error: No type info available for this type Type information is not generated for some types, such as enumerations with gaps in their value range (this includes enumerations whose lower bound is different from zero).

C.5 Symbol handling

This section lists all the messages that concern the handling of symbols. This means all things that have to do with procedure and variable names.

Error: Identifier not found "arg1" The compiler doesn't know this symbol. Usually happens when you misspell the name of a variable or procedure, or when you forget to declare a variable.

Fatal: Internal Error in SymTableStack() An internal error occurred in the compiler; If you encounter such an error, please contact the developers and try to provide an exact description of the circumstances in which the error occurs.

Error: Duplicate identifier "arg1" The identifier was already declared in the current scope.

Hint: Identifier already defined in arg1 at line arg2 The identifier was already declared in a previous scope.

Error: Unknown identifier "arg1" The identifier encountered has not been declared, or is used outside the scope where it is defined.

Error: Forward declaration not solved "arg1" This can happen in two cases:

- You declare a function in the `interface` part, or with a `forward` directive, but do not implement it.
- You reference a type which isn't declared in the current `type` block.

Error: Error in type definition There is an error in your definition of a new array type. One of the range delimiters in an array declaration is erroneous. For example, `Array [1..1.25]` will trigger this error.

Error: Forward type not resolved "arg1" A symbol was forward defined, but no declaration was encountered.

Error: Only static variables can be used in static methods or outside methods A static method of an object can only access static variables.

Fatal: record or class type expected The variable or expression isn't of the type `record` or `class`.

Error: Instances of classes or objects with an abstract method are not allowed You are trying to generate an instance of a class which has an abstract method that wasn't overridden.

Warning: Label not defined "arg1" A label was declared, but not defined.

Error: Label used but not defined "arg1" A label was declared and used, but not defined.

Error: Illegal label declaration This error should never happen; it occurs if a label is defined outside a procedure or function.

Error: GOTO and LABEL are not supported (use switch -Sg) You must use the -Sg switch to compile a program which has `labels` and `goto` statements. By default, `label` and `goto` aren't supported.

Error: Label not found A `goto label` was encountered, but the label wasn't declared.

Error: identifier isn't a label The identifier specified after the `goto` isn't of type label.

Error: label already defined You are defining a label twice. You can define a label only once.

Error: illegal type declaration of set elements The declaration of a set contains an invalid type definition.

Error: Forward class definition not resolved "arg1" You declared a class, but you did not implement it.

Hint: Unit "arg1" not used in arg2 The unit referenced in the `uses` clause is not used.

Hint: Parameter "arg1" not used The identifier was declared (locally or globally) but was not used (locally or globally).

Note: Local variable "arg1" not used You have declared, but not used, a variable in a procedure or function implementation.

Hint: Value parameter "arg1" is assigned but never used The identifier was declared (locally or globally) and assigned to, but is not used (locally or globally) after the assignment.

Note: Local variable "arg1" is assigned but never used The variable in a procedure or function implementation is declared and assigned to, but is not used after the assignment.

Hint: Local arg1 "arg2" is not used A local symbol is never used.

Note: Private field "arg1.arg2" is never used The indicated private field is defined, but is never used in the code.

Note: Private field "arg1.arg2" is assigned but never used The indicated private field is declared and assigned to, but never read.

Note: Private method "arg1.arg2" never used The indicated private method is declared but is never used in the code.

Error: Set type expected The variable or expression is not of type `set`. This happens in an `in` statement.

Warning: Function result does not seem to be set You can get this warning if the compiler thinks that a function return value is not set. This will not be displayed for assembler procedures, or procedures that contain assembler blocks.

Warning: Type "arg1" is not aligned correctly in current record for C Arrays with sizes not multiples of 4 will be wrongly aligned for C structures.

Error: Unknown record field identifier "arg1" The field doesn't exist in the record/object definition.

Warning: Local variable "arg1" does not seem to be initialized This message is displayed if the compiler thinks that a variable will be used (i.e. it appears in the right-hand side of an expression) when it was not initialized first (i.e. appeared in the left-hand side of an assignment).

Warning: Variable "arg1" does not seem to be initialized This message is displayed if the compiler thinks that a variable will be used (i.e. it appears in the right-hand side of an expression) when it was not initialized first (i.e. appeared in the left-hand side of an assignment).

Error: identifier idents no member "arg1" This error is generated when an identifier of a record, field or method is accessed while it is not defined.

Hint: Found declaration: arg1 You get this when you use the `-vh` switch.In the case of an overloaded procedure not being found. Then all candidate overloaded procedures are listed, with their parameter lists.

Error: Data element too large You get this when you declare a data element whose size exceeds the prescribed limit (2 Gb on 80386+/68020+ processors).

Error: No matching implementation for interface method "arg1" found There was no matching method found which could implement the interface method. Check argument types and result type of the methods.

Warning: Symbol "arg1" is deprecated This means that a symbol (a variable, routine, etc...) which is declared as `deprecated` is used. Deprecated symbols may no longer be available in newer versions of the unit / library. Use of this symbol should be avoided as much as possible.

Warning: Symbol "arg1" is not portable This means that a symbol (a variable, routine, etc...) which is declared as `platform` is used. This symbol's value, use and availability is platform specific and should not be used if the source code must be portable.

Warning: Symbol "arg1" is not implemented This means that a symbol (a variable, routine, etc...) which is declared as `unimplemented` is used. This symbol is defined, but is not yet implemented on this specific platform.

Error: Can't create unique type from this type Only simple types like ordinal, float and string types are supported when redefining a type with `type newtype = type oldtype;`.

Hint: Local variable "arg1" does not seem to be initialized This message is displayed if the compiler thinks that a variable will be used (i.e. it appears in the right-hand side of an expression) when it was not initialized first (i.e. it did not appear in the left-hand side of an assignment).

Hint: Variable "arg1" does not seem to be initialized This message is displayed if the compiler thinks that a variable will be used (i.e. it appears in the right-hand side of an expression) when it was not initialized first (i.e. t did not appear in the left-hand side of an assignment).

Warning: Function result variable does not seem to initialized This message is displayed if the compiler thinks that the function result variable will be used (i.e. it appears in the right-hand side of an expression) before it is initialized (i.e. before it appeared in the left-hand side of an assignment).

Hint: Function result variable does not seem to be initialized This message is displayed if the compiler thinks that the function result variable will be used (i.e. it appears in the right-hand side of an expression) before it is initialized (i.e. it appears in the left-hand side of an assignment)

Warning: Variable "arg1" read but nowhere assigned You have read the value of a variable, but nowhere assigned a value to it.

Hint: Found abstract method: arg1 When getting a warning about constructing a class/object with abstract methods you get this hint to assist you in finding the affected method.

Warning: Symbol "arg1" is experimental This means that a symbol (a variable, routine, etc...) which is declared as `experimental` is used. Experimental symbols might disappear or change semantics in future versions. Usage of this symbol should be avoided as much as possible.

Warning: Forward declaration "arg1" not resolved, assumed external This happens if you declare a function in the `interface` of a unit in macpas mode, but do not implement it.

Warning: Symbol "arg1" is belongs to a library This means that a symbol (a variable, routine, etc...) which is declared as `library` is used. Library symbols may not be available in other libraries.

Warning: Symbol "arg1" is deprecated: "arg2" This means that a symbol (a variable, routine, etc...) which is declared as `deprecated` is used. Deprecated symbols may no longer be available in newer versions of the unit / library. Use of this symbol should be avoided as much as possible.

Error: Cannot find an enumerator for the type "arg1" This means that compiler cannot find an apropriate enumerator to use in the for-in loop. To create an enumerator you need to defind an operator enumerator or add a public or published GetEnumerator method to the class or object definition.

Error: Cannot find a "MoveNext" method in enumerator "arg1" This means that compiler cannot find a public MoveNext method with the Boolean return type in the enumerator class or object definition.

Error: Cannot find a "Current" property in enumerator "arg1" This means that compiler cannot find a public Current property in the enumerator class or object definition.

C.6 Code generator messages

This section lists all messages that can be displayed if the code generator encounters an error condition.

Error: Parameter list size exceeds 65535 bytes The I386 processor limits the parameter list to 65535 bytes. (The `RET` instruction causes this.)

Error: File types must be var parameters You cannot specify files as value parameters, i.e., they must always be declared `var` parameters.

Error: The use of a far pointer isn't allowed there Free Pascal doesn't support far pointers, so you cannot take the address of an expression which has a far reference as a result. The `mem` construct has a far reference as a result, so the following code will produce this error:

```
var p : pointer;
...
p:=@mem[a000:000];
```

Error: EXPORT declared functions can't be called No longer in use.

Warning: Possible illegal call of constructor or destructor The compiler detected that a constructor or destructor is called within a a method. This will probably lead to problems, since constructors / destructors require parameters on entry.

Note: Inefficient code Your statement seems dubious to the compiler.

Warning: unreachable code You specified a construct which will never be executed. Example:

```
while false do
  begin
  {.. code ...}
  end;
```

Error: Abstract methods can't be called directly You cannot call an abstract method directly. Instead, you must call an overriding child method, because an abstract method isn't implemented.

Register arg1 weight arg2 arg3 Debugging message. Shown when the compiler considers a variable for keeping in the registers.

Stack frame is omitted Some procedure/functions do not need a complete stack-frame, so it is omitted. This message will be displayed when the -vd switch is used.

Error: Object or class methods can't be inline. You cannot have inlined object methods.

Error: Procvar calls cannot be inline. A procedure with a procedural variable call cannot be inlined.

Error: No code for inline procedure stored The compiler couldn't store code for the inline procedure.

Error: Element zero of an ansi/wide- or longstring can't be accessed, use (set)length instead You should use `setlength` to set the length of an ansi/wide/longstring and `length` to get the length of such string type.

Error: Constructors or destructors cannot be called inside a 'with' clause Inside a `with` clause you cannot call a constructor or destructor for the object you have in the `with` clause.

Error: Cannot call message handler methods directly A message method handler method cannot be called directly if it contains an explicit `Self` argument.

Error: Jump in or outside of an exception block It is not allowed to jump in or outside of an exception block like `try..finally..end;`. For example, the following code will produce this error:

```
label 1;

...

try
   if not(final) then
     goto 1;   // this line will cause an error
finally
  ...
end;
1:
...
```

Error: Control flow statements aren't allowed in a finally block It isn't allowed to use the control flow statements `break`, `continue` and `exit` inside a finally statement. The following example shows the problem:

```
...
  try
     p;
  finally
     ...
     exit;  // This exit ISN'T allowed
  end;
...
```

If the procedure `p` raises an exception the finally block is executed. If the execution reaches the exit, it's unclear what to do: exit the procedure or search for another exception handler.

Warning: Parameters size exceeds limit for certain cpu's This indicates that you are declaring more than 64K of parameters, which might not be supported on other processor targets.

Warning: Local variable size exceed limit for certain cpu's This indicates that you are declaring more than 32K of local variables, which might not be supported on other processor targets.

Error: Local variables size exceeds supported limit This indicates that you are declaring more than 32K of local variables, which is not supported by this processor.

Error: BREAK not allowed You're trying to use `break` outside a loop construction.

Error: CONTINUE not allowed You're trying to use `continue` outside a loop construction.

Fatal: Unknown compilerproc "arg1". Check if you use the correct run time library. The compiler expects that the runtime library contains certain subroutines. If you see this error and you didn't change the runtime library code, it's very likely that the runtime library you're using doesn't match the compiler in use. If you changed the runtime library this error means that you removed a subroutine which the compiler needs for internal use.

Fatal: Cannot find system type "arg1". Check if you use the correct run time library. The compiler expects that the runtime library contains certain type definitions. If you see this error and you didn't change the runtime library code, it's very likely that the runtime library you're using doesn't match the compiler in use. If you changed the runtime library this error means that you removed a type which the compiler needs for internal use.

Hint: Inherited call to abstract method ignored This message appears only in Delphi mode when you call an abstract method of a parent class via `inherited;`. The call is then ignored.

Error: Goto label "arg1" not defined or optimized away The label used in the goto definition is not defined or optimized away by the unreachable code elemination.

C.7 Errors of assembling/linking stage

This section lists errors that occur when the compiler is processing the command line or handling the configuration files.

Warning: Source operating system redefined The source operating system is redefined.

Info: Assembling (pipe) arg1 Assembling using a pipe to an external assembler.

Error: Can't create assembler file: arg1 The mentioned file can't be created. Check if you have access permissions to create this file.

Error: Can't create object file: arg1 The mentioned file can't be created. Check if you have got access permissions to create this file.

Error: Can't create archive file: arg1 The mentioned file can't be created. Check if you have access permissions to create this file.

Error: Assembler arg1 not found, switching to external assembling The assembler program was not found. The compiler will produce a script that can be used to assemble and link the program.

Using assembler: arg1 An informational message saying which assembler is being used.

Error: Error while assembling exitcode arg1 There was an error while assembling the file using an external assembler. Consult the documentation of the assembler tool to find out more information on this error.

Error: Can't call the assembler, error arg1 switching to external assembling An error occurred when calling an external assembler. The compiler will produce a script that can be used to assemble and link the program.

Info: Assembling arg1 An informational message stating which file is being assembled.

Info: Assembling with smartlinking arg1 An informational message stating which file is being assembled using smartlinking.

Warning: Object arg1 not found, Linking may fail ! One of the object files is missing, and linking will probably fail. Check your paths.

Warning: Library arg1 not found, Linking may fail ! One of the library files is missing, and linking will probably fail. Check your paths.

Error: Error while linking Generic error while linking.

Error: Can't call the linker, switching to external linking An error occurred when calling an external linker. The compiler will produce a script that can be used to assemble and link the program.

Info: Linking arg1 An informational message, showing which program or library is being linked.

Error: Util arg1 not found, switching to external linking An external tool was not found. The compiler will produce a script that can be used to assemble and link or postprocess the program.

Using util arg1 An informational message, showing which external program (usually a postprocessor) is being used.

Error: Creation of Executables not supported Creating executable programs is not supported for this platform, because it was not yet implemented in the compiler.

Error: Creation of Dynamic/Shared Libraries not supported Creating dynamically loadable libraries is not supported for this platform, because it was not yet implemented in the compiler.

Info: Closing script arg1 Informational message showing when writing of the external assembling and linking script is finished.

Error: resource compiler "arg1" not found, switching to external mode An external resource compiler was not found. The compiler will produce a script that can be used to assemble, compile resources and link or postprocess the program.

Info: Compiling resource arg1 An informational message, showing which resource is being compiled.

unit arg1 can't be statically linked, switching to smart linking Static linking was requested, but a unit which is not statically linkable was used.

unit arg1 can't be smart linked, switching to static linking Smart linking was requested, but a unit which is not smart-linkable was used.

unit arg1 can't be shared linked, switching to static linking Shared linking was requested, but a unit which is not shared-linkable was used.

Error: unit arg1 can't be smart or static linked Smart or static linking was requested, but a unit which cannot be used for either was used.

Error: unit arg1 can't be shared or static linked Shared or static linking was requested, but a unit which cannot be used for either was used.

Calling resource compiler "arg1" with "arg2" as command line An informational message showing which command line is used for the resource compiler.

Error: Error while compiling resources The resource compiler or converter returned an error.

Error: Can't call the resource compiler "arg1", switching to external mode An error occurred when calling a resource compiler. The compiler will produce a script that can be used to assemble, compile resources and link or postprocess the program.

Error: Can't open resource file "arg1" An error occurred resource file cannot be opened.

Error: Can't write resource file "arg1" An error occurred resource file cannot be written.

C.8 Executable information messages.

This section lists all messages that the compiler emits when an executable program is produced, and only when the internal linker is used.

Fatal: Can't post process executable arg1 Fatal error when the compiler is unable to post-process an executable.

Fatal: Can't open executable arg1 Fatal error when the compiler cannot open the file for the executable.

Size of Code: arg1 bytes Informational message showing the size of the produced code section.

Size of initialized data: arg1 bytes Informational message showing the size of the initialized data section.

Size of uninitialized data: arg1 bytes Informational message showing the size of the uninitialized data section.

Stack space reserved: arg1 bytes Informational message showing the stack size that the compiler reserved for the executable.

Stack space committed: arg1 bytes Informational message showing the stack size that the compiler committed for the executable.

C.9 Linker messages

This section lists messages produced by internal linker.

Fatal: Executable image size is too big for arg1 target. Fatal error when resulting executable is too big.

Warning: Object file "arg1" contains 32-bit absolute relocation to symbol "arg2". Warning when 64-bit object file contains 32-bit absolute relocations. In such case an executable image can be loaded into lower 4Gb of address space only.

C.10 Unit loading messages.

This section lists all messages that can occur when the compiler is loading a unit from disk into memory. Many of these messages are informational messages.

Unitsearch: arg1 When you use the `-vt` option, the compiler tells you where it tries to find unit files.

PPU Loading arg1 When the `-vt` switch is used, the compiler tells you what units it loads.

PPU Name: arg1 When you use the `-vu` flag, the unit name is shown.

PPU Flags: arg1 When you use the `-vu` flag, the unit flags are shown.

PPU Crc: arg1 When you use the `-vu` flag, the unit CRC check is shown.

PPU Time: arg1 When you use the `-vu` flag, the time the unit was compiled is shown.

PPU File too short The ppufile is too short, not all declarations are present.

PPU Invalid Header (no PPU at the begin) A unit file contains as the first three bytes the ASCII codes of the characters `PPU`.

PPU Invalid Version arg1 This unit file was compiled with a different version of the compiler, and cannot be read.

PPU is compiled for another processor This unit file was compiled for a different processor type, and cannot be read.

PPU is compiled for an other target This unit file was compiled for a different target, and cannot be read.

PPU Source: arg1 When you use the `-vu` flag, the unit source file name is shown.

Writing arg1 When you specify the `-vu` switch, the compiler will tell you where it writes the unit file.

Fatal: Can't Write PPU-File An error occurred when writing the unit file.

Fatal: Error reading PPU-File This means that the unit file was corrupted, and contains invalid information. Recompilation will be necessary.

Fatal: unexpected end of PPU-File Unexpected end of file. This may mean that the PPU file is corrupted.

Fatal: Invalid PPU-File entry: arg1 The unit the compiler is trying to read is corrupted, or generated with a newer version of the compiler.

Fatal: PPU Dbx count problem There is an inconsistency in the debugging information of the unit.

Error: Illegal unit name: arg1 The name of the unit does not match the file name.

Fatal: Too much units Free Pascal has a limit of 1024 units in a program. You can change this behavior by changing the `maxunits` constant in the fmodule.pas file of the compiler, and recompiling the compiler.

Fatal: Circular unit reference between arg1 and arg2 Two units are using each other in the interface part. This is only allowed in the `implementation` part. At least one unit must contain the other one in the `implementation` section.

Fatal: Can't compile unit arg1, no sources available A unit was found that needs to be recompiled, but no sources are available.

Fatal: Can't find unit arg1 used by arg2 You tried to use a unit of which the PPU file isn't found by the compiler. Check your configuration file for the unit paths.

Warning: Unit arg1 was not found but arg2 exists This error message is no longer used.

Fatal: Unit arg1 searched but arg2 found `DOS` truncation of 8 letters for unit PPU files may lead to problems when unit name is longer than 8 letters.

Warning: Compiling the system unit requires the -Us switch When recompiling the system unit (it needs special treatment), the `-Us` switch must be specified.

Fatal: There were arg1 errors compiling module, stopping When the compiler encounters a fatal error or too many errors in a module then it stops with this message.

Load from arg1 (arg2) unit arg3 When you use the `-vu` flag, which unit is loaded from which unit is shown.

Recompiling arg1, checksum changed for arg2 The unit is recompiled because the checksum of a unit it depends on has changed.

Recompiling arg1, source found only When you use the -vu flag, these messages tell you why the current unit is recompiled.

Recompiling unit, static lib is older than ppufile When you use the -vu flag, the compiler warns if the static library of the unit is older than the unit file itself.

Recompiling unit, shared lib is older than ppufile When you use the -vu flag, the compiler warns if the shared library of the unit is older than the unit file itself.

Recompiling unit, obj and asm are older than ppufile When you use the -vu flag, the compiler warns if the assembler or object file of the unit is older than the unit file itself.

Recompiling unit, obj is older than asm When you use the -vu flag, the compiler warns if the assembler file of the unit is older than the object file of the unit.

Parsing interface of arg1 When you use the -vu flag, the compiler warns that it starts parsing the interface part of the unit.

Parsing implementation of arg1 When you use the -vu flag, the compiler warns that it starts parsing the implementation part of the unit.

Second load for unit arg1 When you use the -vu flag, the compiler warns that it starts recompiling a unit for the second time. This can happen with interdependent units.

PPU Check file arg1 time arg2 When you use the -vu flag, the compiler shows the filename and date and time of the file on which a recompile depends.

Warning: Can't recompile unit arg1, but found modifed include files A unit was found to have modified include files, but some source files were not found, so recompilation is impossible.

File arg1 is newer than PPU file arg2 A modified source file for a compiler unit was found.

Trying to use a unit which was compiled with a different FPU mode Trying to compile code while using units which were not compiled with the same floating point format mode. Either all code should be compiled with FPU emulation on, or with FPU emulation off.

Loading interface units from arg1 When you use the -vu flag, the compiler warns that it is starting to load the units defined in the interface part of the unit.

Loading implementation units from arg1 When you use the -vu flag, the compiler warns that it is starting to load the units defined in the implementation part of the unit.

Interface CRC changed for unit arg1 When you use the -vu flag, the compiler warns that the CRC calculated for the interface has been changed after the implementation has been parsed.

Implementation CRC changed for unit arg1 When you use the -vu flag, the compiler warns that the CRC calculated has been changed after the implementation has been parsed.

Finished compiling unit arg1 When you use the -vu flag, the compiler warns that it has finished compiling the unit.

Add dependency of arg1 to arg2 When you use the -vu flag, the compiler warns that it has added a dependency between the two units.

No reload, is caller: arg1 When you use the -vu flag, the compiler warns that it will not reload the unit because it is the unit that wants to load this unit.

No reload, already in second compile: arg1 When you use the -vu flag, the compiler warns that it will not reload the unit because it is already in a second recompile.

Flag for reload: arg1 When you use the -vu flag, the compiler warns that it has to reload the unit.

Forced reloading When you use the `-vu` flag, the compiler warns that it is reloading the unit because it was required.

Previous state of arg1: arg2 When you use the `-vu` flag, the compiler shows the previous state of the unit.

Already compiling arg1, setting second compile When you use the `-vu` flag, the compiler warns that it is starting to recompile a unit for the second time. This can happen with interdependent units.

Loading unit arg1 When you use the `-vu` flag, the compiler warns that it starts loading the unit.

Finished loading unit arg1 When you use the `-vu` flag, the compiler warns that it finished loading the unit.

Registering new unit arg1 When you use the `-vu` flag, the compiler warns that it has found a new unit and is registering it in the internal lists.

Re-resolving unit arg1 When you use the `-vu` flag, the compiler warns that it has to recalculate the internal data of the unit.

Skipping re-resolving unit arg1, still loading used units When you use the `-vu` flag, the compiler warns that it is skipping the recalculation of the internal data of the unit because there is no data to recalculate.

Unloading resource unit arg1 (not needed) When you use the `-vu` flag, the compiler warns that it is unloading the resource handling unit, since no resources are used.

Error: Unit arg1 was compiled using a different whole program optimization feedback input (arg2, arg3); recompile it When a unit has been compiled using a particular whole program optimization (wpo) feedback file (`-FW<x> -OW<x>`), this compiled version of the unit is specialised for that particular compilation scenario and cannot be used in any other context. It has to be recompiled before you can use it in another program or with another wpo feedback input file.

C.11 Command line handling errors

This section lists errors that occur when the compiler is processing the command line or handling the configuration files.

Warning: Only one source file supported, changing source file to compile from "arg1" into "arg2" You can specify only one source file on the command line. The last one will be compiled, others will be ignored. This may indicate that you forgot a `'-'` sign.

Warning: DEF file can be created only for OS/2 This option can only be specified when you're compiling for OS/2.

Error: nested response files are not supported You cannot nest response files with the `@file` command line option.

Fatal: No source file name in command line The compiler expects a source file name on the command line.

Note: No option inside arg1 config file The compiler didn't find any option in that config file.

Error: Illegal parameter: arg1 You specified an unknown option.

Hint: -? writes help pages When an unknown option is given, this message is diplayed.

Fatal: Too many config files nested You can only nest up to 16 config files.

Fatal: Unable to open file arg1 The option file cannot be found.

Reading further options from arg1 Displayed when you have notes turned on, and the compiler switches to another options file.

Warning: Target is already set to: arg1 Displayed if more than one `-T` option is specified.

Warning: Shared libs not supported on DOS platform, reverting to static If you specify `-CD` for the `DOS` platform, this message is displayed. The compiler supports only static libraries under `DOS`.

Fatal: In options file arg1 at line arg2 too many \var{\#IF(N)DEFs} encountered The `#IF(N)DEF` statements in the options file are not balanced with the `#ENDIF` statements.

Fatal: In options file arg1 at line arg2 unexpected \var{\#ENDIFs} encountered The `#IF(N)DEF` statements in the options file are not balanced with the `#ENDIF` statements.

Fatal: Open conditional at the end of the options file The `#IF(N)DEF` statements in the options file are not balanced with the `#ENDIF` statements.

Warning: Debug information generation is not supported by this executable It is possible to have a compiler executable that doesn't support the generation of debugging info. If you use such an executable with the `-g` switch, this warning will be displayed.

Hint: Try recompiling with -dGDB It is possible to have a compiler executable that doesn't support the generation of debugging info. If you use such an executable with the `-g` switch, this warning will be displayed.

Warning: You are using the obsolete switch arg1 This warns you when you use a switch that is not needed/supported anymore. It is recommended that you remove the switch to overcome problems in the future, when the meaning of the switch may change.

Warning: You are using the obsolete switch arg1, please use arg2 This warns you when you use a switch that is not supported anymore. You must now use the second switch instead. It is recommended that you change the switch to overcome problems in the future, when the meaning of the switch may change.

Note: Switching assembler to default source writing assembler This notifies you that the assembler has been changed because you used the -a switch, which can't be used with a binary assembler writer.

Warning: Assembler output selected "arg1" is not compatible with "arg2"

Warning: "arg1" assembler use forced The assembler output selected cannot generate object files with the correct format. Therefore, the default assembler for this target is used instead.

Reading options from file arg1 Options are also read from this file.

Reading options from environment arg1 Options are also read from this environment string.

Handling option "arg1" Debug info that an option is found and will be handled.

***** press enter ***** Message shown when help is shown page per page. When pressing the ENTER Key, the next page of help is shown. If you press q and then ENTER, the compiler exits.

Hint: Start of reading config file arg1 Start of configuration file parsing.

Hint: End of reading config file arg1 End of configuration file parsing.

interpreting option "arg1" The compiler is interpreting an option

interpreting firstpass option "arg1" The compiler is interpreting an option for the first time.

interpreting file option "arg1" The compiler is interpreting an option which it read from the configuration file.

Reading config file "arg1" The compiler is starting to read the configuration file.

found source file name "arg1" Additional information about options. Displayed when you have the debug option turned on.

Error: Unknown code page An unknown code page for the source files was requested. The compiler is compiled with support for several code pages built-in. The requested code page is not in that list. You will need to recompile the compiler with support for the codepage you need.

Fatal: Config file arg1 is a directory Directories cannot be used as configuration files.

Warning: Assembler output selected "arg1" cannot generate debug info, debugging disabled The selected assembler output cannot generate debugging information, debugging option is therefore disabled.

Warning: Use of ppc386.cfg is deprecated, please use fpc.cfg instead Using ppc386.cfg is still supported for historical reasons, however, for a multiplatform system the naming makes no sense anymore. Please continue to use fpc.cfg instead.

Fatal: In options file arg1 at line arg2 \var{\#ELSE} directive without \var{\#IF(N)DEF} found An `#ELSE` statement was found in the options file without a matching `#IF(N)DEF` statement.

Fatal: Option "arg1" is not, or not yet, supported on the current target platform Not all options are supported or implemented for all target platforms. This message informs you that a chosen option is incompatible with the currently selected target platform.

Fatal: The feature "arg1" is not, or not yet, supported on the selected target platform Not all features are supported or implemented for all target platforms. This message informs you that a chosen feature is incompatible with the currently selected target platform.

Note: DWARF debug information cannot be used with smart linking on this target, switching to static linking Smart linking is currently incompatble with DWARF debug information on most platforms, so smart linking is disabled in such cases.

Warning: Option "arg1" is ignored for the current target platform. Not all options are supported or implemented for all target platforms. This message informs you that a chosen option is ignored for the currently selected target platform.

C.12 Whole program optimization messages

This section lists errors that occur when the compiler is performing whole program optimization.

Fatal: Cannot open whole program optimization feedback file "arg1" The compiler cannot open the specified feedback file with whole program optimization information.

Processing whole program optimization information in wpo feedback file "arg1" The compiler starts processing whole program optimization information found in the named file.

Finished processing the whole program optimization information in wpo feedback file "arg1" The compiler has finished processing the whole program optimization information found in the named file.

Error: Expected section header, but got "arg2" at line arg1 of wpo feedback file The compiler expected a section header in the whole program optimization file (starting with %), but did not find it.

Warning: No handler registered for whole program optimization section "arg2" at line arg1 of wpo feedback file, ignor
The compiler has no handler to deal with the mentioned whole program optimization information section, and will therefore ignore it and skip to the next section.

Found whole program optimization section "arg1" with information about "arg2" The compiler encountered a section with whole program optimization information, and according to its handler this section contains information usable for the mentioned purpose.

Fatal: The selected whole program optimizations require a previously generated feedback file (use -Fw to specify)
The compiler needs information gathered during a previous compilation run to perform the selected whole program optimizations. You can specify the location of the feedback file containing this information using the -Fw switch.

Error: No collected information necessary to perform "arg1" whole program optimization found
While you pointed the compiler to a file containing whole program optimization feedback, it did not contain the information necessary to perform the selected optimizations. You most likely have to recompile the program using the appropate -OWxxx switch.

Fatal: Specify a whole program optimization feedback file to store the generated info in (using -FW)
You have to specify the feedback file in which the compiler has to store the whole program optimization feedback that is generated during the compilation run. This can be done using the -FW switch.

Error: Not generating any whole program optimization information, yet a feedback file was specified (using -FW)
The compiler was instructed to store whole program optimization feedback into a file specified using -FW, but not to actually generated any whole program optimization feedback. The classes of to be generated information can be speciied using -OWxxx.

Error: Not performing any whole program optimizations, yet an input feedback file was specified (using -Fw)
The compiler was not instructed to perform any whole program optimizations (no -Owxxx parameters), but nevertheless an input file with such feedback was specified (using -Fwyyy). Since this can indicate that you forgot to specify an -Owxxx parameter, the compiler generates an error in this case.

Skipping whole program optimization section "arg1", because not needed by the requested optimizations
The whole program optimization feedback file contains a section with information that is not required by the selected whole program optimizations.

Warning: Overriding previously read information for "arg1" from feedback input file using information in section "ar
The feedback file contains multiple sections that provide the same class of information (e.g., information about which virtual methods can be devirtualized). In this case, the information in last encountered section is used. Turn on debugging output (-vd) to see which class of information is provided by each section.

Error: Cannot extract symbol liveness information from program when stripping symbols, use -Xs-
Certain symbol liveness collectors extract the symbol information from the linked program. If the symbol information is stripped (option -Xs), this is not possible.

Error: Cannot extract symbol liveness information from program when when not linking Certain symbol liveness collectors extract the symbol information from the linked program. If the program is not linked by the compiler, this is not possible.

Fatal: Cannot find "arg1" or "arg2" to extract symbol liveness information from linked program Certain symbol liveness collectors need a helper program to extract the symbol information from the linked program. This helper program is normally 'nm', which is part of the GNU binutils.

Error: Error during reading symbol liveness information produced by "arg1" An error occurred during the reading of the symbol liveness file that was generated using the 'nm' or 'objdump' program. The reason can be that it was shorter than expected, or that its format was not understood.

Fatal: Error executing "arg1" (exitcode: arg2) to extract symbol information from linked program Certain symbol liveness collectors need a helper program to extract the symbol information from the linked program. The helper program produced the reported error code when it was run on the linked program.

Error: Collection of symbol liveness information can only help when using smart linking, use -CX -XX Whether or not a symbol is live is determined by looking whether it exists in the final linked program. Without smart linking/dead code stripping, all symbols are always included, regardless of whether they are actually used or not. So in that case all symbols will be seen as live, which makes this optimization ineffective.

Error: Cannot create specified whole program optimisation feedback file "arg1" The compiler is unable to create the file specified using the -FW parameter to store the whole program optimisation information.

C.13 Assembler reader errors.

This section lists the errors that are generated by the inline assembler reader. They are *not* the messages of the assembler itself.

C.13.1 General assembler errors

Divide by zero in asm evaluator This fatal error is reported when a constant assembler expression performs a division by zero.

Evaluator stack overflow, Evaluator stack underflow These fatal error is reported when a constant assembler expression is too big to be evaluated by the constant parser. Try reducing the number of terms.

Invalid numeric format in asm evaluator This fatal error is reported when a non-numeric value is detected by the constant parser. Normally this error should never occur.

Invalid Operator in asm evaluator This fatal error is reported when a mathematical operator is detected by the constant parser. Normally this error should never occur.

Unknown error in asm evaluator This fatal error is reported when an internal error is detected by the constant parser. Normally this error should never occur.

Invalid numeric value This warning is emitted when a conversion from octal, binary or hexadecimal to decimal is outside of the supported range.

Escape sequence ignored This error is emitted when a non ANSI C escape sequence is detected in a C string.

Asm syntax error - Prefix not found This occurs when trying to use a non-valid prefix instruction.

Asm syntax error - Trying to add more than one prefix This occurs when you try to add more than one prefix instruction.

Asm syntax error - Opcode not found You have tried to use an unsupported or unknown opcode.

Constant value out of bounds This error is reported when the constant parser determines that the value you are using is out of bounds, either with the opcode or with the constant declaration used.

Non-label pattern contains @ This only applied to the m68k and Intel styled assembler. This is reported when you try to use a non-label identifier with an '@' prefix.

Internal error in Findtype()

Internal Error in ConcatOpcode()

Internal Errror converting binary

Internal Errror converting hexadecimal

Internal Errror converting octal

Internal Error in BuildScaling()

Internal Error in BuildConstant()

internal error in BuildReference()

internal error in HandleExtend()

Internal error in ConcatLabeledInstr() These errors should never occur. If they do then you have found a new bug in the assembler parsers. Please contact one of the developers.

Opcode not in table, operands not checked This warning only occurs when compiling the system unit, or related files. No checking is performed on the operands of the opcodes.

@CODE and @DATA not supported This Turbo Pascal construct is not supported.

SEG and OFFSET not supported This Turbo Pascal construct is not supported.

Modulo not supported Modulo constant operation is not supported.

Floating point binary representation ignored

Floating point hexadecimal representation ignored

Floating point octal representation ignored These warnings occur when a floating point constant is declared in a base other than decimal. No conversion can be done on these formats. You should use a decimal representation instead.

Identifier supposed external This warning occurs when a symbol is not found in the symbol table. It is therefore considered external.

Functions with void return value can't return any value in asm code Only routines with a return value can have a return value set.

Error in binary constant

Error in octal constant

Error in hexadecimal constant

Error in integer constant These errors are reported when you tried using a constant expression that is invalid or whose value is out of range.

Invalid labeled opcode

Asm syntax error - error in reference

Invalid Opcode

Invalid combination of opcode and operands

Invalid size in reference

Invalid middle sized operand

Invalid three operand opcode

Assembler syntax error

Invalid operand type You tried using an invalid combination of opcode and operands. Check the syntax and if you are sure it is correct, please contact one of the developers.

Unknown identifier The identifier you are trying to access does not exist, or is not within the current scope.

Trying to define an index register more than once

Trying to define a segment register twice

Trying to define a base register twice You are trying to define an index/segment register more than once.

Invalid field specifier The record or object field you are trying to access does not exist, or is incorrect.

Invalid scaling factor

Invalid scaling value

Scaling value only allowed with index Allowed scaling values are 1,2,4 or 8.

Cannot use SELF outside a method You are trying to access the SELF identifier for objects outside a method.

Invalid combination of prefix and opcode This opcode cannot be prefixed by this instruction.

Invalid combination of override and opcode This opcode cannot be overriden by this combination.

Too many operands on line At most three operand instructions exist on the m68k, and i386, you are probably trying to use an invalid syntax for this opcode.

Duplicate local symbol You are trying to redefine a local symbol, such as a local label.

Unknown label identifer

Undefined local symbol

local symbol not found inside asm statement This label does not seem to have been defined in the current scope.

Assemble node syntax error

Not a directive or local symbol The assembler statement is invalid, or you are not using a recognized directive.

C.13.2 I386 specific errors

repeat prefix and a segment override on <= i386 ... A problem with interrupts and a prefix instruction may occur and may cause false results on 386 and earlier computers.

Fwait can cause emulation problems with emu387 This warning is reported when using the FWAIT instruction. It can cause emulation problems on systems which use the em387.dxe emulator.

You need GNU as version >= 2.81 to compile this MMX code MMX assembler code can only be compiled using GAS v2.8.1 or later.

NEAR ignored

FAR ignored `NEAR` and `FAR` are ignored in the Intel assemblers, but are still accepted for compatiblity with the 16-bit code model.

Invalid size for MOVSX/MOVZX

16-bit base in 32-bit segment

16-bit index in 32-bit segment 16-bit addressing is not supported. You must use 32-bit addressing.

Constant reference not allowed It is not allowed to try to address a constant memory address in protected mode.

Segment overrides not supported Intel style (eg: rep ds stosb) segment overrides are not supported by the assembler parser.

Expressions of the form [sreg:reg...] are currently not supported To access a memory operand in a different segment, you should use the sreg:[reg...] snytax instead of [sreg:reg...]

Size suffix and destination register do not match In intel AT&T syntax, you are using a register size which does not concord with the operand size specified.

Invalid assembler syntax. No ref with brackets

Trying to use a negative index register

Local symbols not allowed as references

Invalid operand in bracket expression

Invalid symbol name:

Invalid Reference syntax

Invalid string as opcode operand:

Null label references are not allowed

Using a defined name as a local label

Invalid constant symbol

Invalid constant expression

/ at beginning of line not allowed

NOR not supported

Invalid floating point register name

Invalid floating point constant:

Asm syntax error - Should start with bracket

Asm syntax error - register:

Asm syntax error - in opcode operand

Invalid String expression

Constant expression out of bounds

Invalid or missing opcode

Invalid real constant expression

Parenthesis are not allowed

Invalid Reference

Cannot use __SELF outside a method

Cannot use __OLDEBP outside a nested procedure

Invalid segment override expression

Strings not allowed as constants

Switching sections is not allowed in an assembler block

Invalid global definition

Line separator expected

Invalid local common definition

Invalid global common definition

assembler code not returned to text

invalid opcode size

Invalid character: <

Invalid character: >

Unsupported opcode

Invalid suffix for intel assembler

Extended not supported in this mode

Comp not supported in this mode

Invalid Operand:

Override operator not supported

C.13.3 m68k specific errors.

Increment and Decrement mode not allowed together You are trying to use dec/inc mode together.

Invalid Register list in movem/fmovem The register list is invalid. Normally a range of registers should be separated by - and individual registers should be separated by a slash.

Invalid Register list for opcode

68020+ mode required to assemble

Appendix D

Run-time errors

Applications generated by Free Pascal might generate run-time errors when certain abnormal conditions are detected in the application. This appendix lists the possible run-time errors and gives information on why they might be produced.

1 Invalid function number An invalid operating system call was attempted.

2 File not found Reported when trying to erase, rename or open a non-existent file.

3 Path not found Reported by the directory handling routines when a path does not exist or is invalid. Also reported when trying to access a non-existent file.

4 Too many open files The maximum number of files currently opened by your process has been reached. Certain operating systems limit the number of files which can be opened concurrently, and this error can occur when this limit has been reached.

5 File access denied Permission to access the file is denied. This error might be caused by one of several reasons:

- Trying to open for writing a file which is read-only, or which is actually a directory.
- File is currently locked or used by another process.
- Trying to create a new file, or directory while a file or directory of the same name already exists.
- Trying to read from a file which was opened in write-only mode.
- Trying to write from a file which was opened in read-only mode.
- Trying to remove a directory or file while it is not possible.
- No permission to access the file or directory.

6 Invalid file handle If this happens, the file variable you are using is trashed; it indicates that your memory is corrupted.

12 Invalid file access code Reported when a reset or rewrite is called with an invalid `FileMode` value.

15 Invalid drive number The number given to the `Getdir` or `ChDir` function specifies a non-existent disk.

16 Cannot remove current directory Reported when trying to remove the currently active directory.

17 Cannot rename across drives You cannot rename a file such that it would end up on another disk or partition.

100 Disk read error An error occurred when reading from disk. Typically happens when you try to read past the end of a file.

101 Disk write error Reported when the disk is full, and you're trying to write to it.

102 File not assigned This is reported by `Reset`, `Rewrite`, `Append`, `Rename` and `Erase`, if you call them with an unassigned file as a parameter.

103 File not open Reported by the following functions : `Close, Read, Write, Seek, EOf, FilePos, FileSize, Flush, BlockRead,` and `BlockWrite` if the file is not open.

104 File not open for input Reported by `Read, BlockRead, Eof, Eoln, SeekEof` or `SeekEoln` if the file is not opened with `Reset`.

105 File not open for output Reported by write if a text file isn't opened with `Rewrite`.

106 Invalid numeric format Reported when a non-numeric value is read from a text file, and a numeric value was expected.

150 Disk is write-protected (Critical error)

151 Bad drive request struct length (Critical error)

152 Drive not ready (Critical error)

154 CRC error in data (Critical error)

156 Disk seek error (Critical error)

157 Unknown media type (Critical error)

158 Sector Not Found (Critical error)

159 Printer out of paper (Critical error)

160 Device write fault (Critical error)

161 Device read fault (Critical error)

162 Hardware failure (Critical error)

200 Division by zero The application attempted to divide a number by zero.

201 Range check error If you compiled your program with range checking on, then you can get this error in the following cases:

1. An array was accessed with an index outside its declared range.
2. Trying to assign a value to a variable outside its range (for instance an enumerated type).

202 Stack overflow error The stack has grown beyond its maximum size (in which case the size of local variables should be reduced to avoid this error), or the stack has become corrupt. This error is only reported when stack checking is enabled.

203 Heap overflow error The heap has grown beyond its boundaries. This is caused when trying to allocate memory explicitly with `New`, `GetMem` or `ReallocMem`, or when a class or object instance is created and no memory is left. Please note that, by default, Free Pascal provides a growing heap, i.e. the heap will try to allocate more memory if needed. However, if the heap has reached the maximum size allowed by the operating system or hardware, then you will get this error.

204 Invalid pointer operation You will get this if you call `Dispose` or `Freemem` with an invalid pointer (notably, `Nil`).

205 Floating point overflow You are trying to use or produce real numbers that are too large.

206 Floating point underflow You are trying to use or produce real numbers that are too small.

207 Invalid floating point operation Can occur if you try to calculate the square root or logarithm of a negative number.

210 Object not initialized When compiled with range checking on, a program will report this error if you call a virtual method without having called its object's constructor.

211 Call to abstract method Your program tried to execute an abstract virtual method. Abstract methods should be overridden, and the overriding method should be called.

212 Stream registration error This occurs when an invalid type is registered in the objects unit.

213 Collection index out of range You are trying to access a collection item with an invalid index (`objects` unit).

214 Collection overflow error The collection has reached its maximal size, and you are trying to add another element (`objects` unit).

215 Arithmetic overflow error This error is reported when the result of an arithmetic operation is outside of its supported range. Contrary to Turbo Pascal, this error is only reported for 32-bit or 64-bit arithmetic overflows. This is due to the fact that everything is converted to 32-bit or 64-bit before doing the actual arithmetic operation.

216 General Protection fault The application tried to access invalid memory space. This can be caused by several problems:

1. Dereferencing a `nil` pointer.
2. Trying to access memory which is out of bounds (for example, calling `move` with an invalid length).

217 Unhandled exception occurred An exception occurred, and there was no exception handler present. The `sysutils` unit installs a default exception handler which catches all exceptions and exits gracefully.

219 Invalid typecast Thrown when an invalid typecast is attempted on a class using the `as` operator. This error is also thrown when an object or class is typecast to an invalid class or object and a virtual method of that class or object is called. This last error is only detected if the `-CR` compiler option is used.

222 Variant dispatch error No dispatch method to call from variant.

223 Variant array create The variant array creation failed. Usually when there is not enough memory.

224 Variant is not an array This error occurs when a variant array operation is attempted on a variant which is not an array.

225 Var Array Bounds check error This error occurs when a variant array index is out of bounds.

227 Assertion failed error An assertion failed, and no `AssertErrorProc` procedural variable was installed.

229 Safecall error check This error occurs is a safecall check fails, and no handler routine is available.

231 Exception stack corrupted This error occurs when the exception object is retrieved and none is available.

232 Threads not supported Thread management relies on a separate driver on some operating systems (notably, Unixes). The unit with this driver needs to be specified on the uses clause of the program, preferably as the first unit (cthreads on unix).