Free Pascal
Programmers' manual

Programmers' manual for Free Pascal, version 0.99.10
1.5
February 1999

Michaël Van Canneyt

Contents

About this document

This is the programmer's manual for Free Pascal.

It describes some of the peculiarities of the Free Pascal compiler, and provides a glimpse of how the compiler generates its code, and how you can change the generated code. It will not, however, provide you with a detailed account of the inner workings of the compiler, nor will it tell you how to use the compiler (described in the Users' guide). It also will not describe the inner workings of the Run-Time Library (RTL). The best way to learn about the way the RTL is implemented is from the sources themselves.

The things described here are useful if you want to do things which need greater flexibility than the standard Pascal language constructs. (described in the Reference guide)

Since the compiler is continuously under development, this document may get out of date. Wherever possible, the information in this manual will be updated. If you find something which isn't correct, or you think something is missing, feel free to contact me[1].

[1] at `michael@tfdec1.fys.kuleuven.ac.be`

Chapter 1

Compiler directives

Free Pascal supports compiler directives in your source file. They are not the same as Turbo Pascal directives, although some are supported for compatibility. There is a distinction between local and global directives; local directives take effect from the moment they are encountered, global directives have an effect on all of the compiled code.

Many switches have a long form also. If they do, then the name of the long form is given also. For long switches, the + or - character to switch the option on or off, may be replaced by `ON` or `OFF` keywords.

Thus `{$I+}` is equivalent to `{$IOCHECKS ON}` or `{$IOCHECKS +}` and `{$C-}` is equivalent to `{$ASSERTIONS OFF}` or `{$ASSERTIONS -}`

The long forms of the switches are the same as their Delphi counterparts.

1.1 Local directives

Local directives have no command-line counterpart. They influence the compiler's behaviour from the moment they're encountered until the moment another switch annihilates their behaviour, or the end of the unit or program is reached.

`$A` or `$ALIGN`: Align Data

This switch is recognized for Turbo Pascal Compatibility, but is not yet implemented. The alignment of data will be different in any case, since Free Pascal is a 32-bit compiler.

`$ASMMODE` : Assembler mode

The {`$ASMMODE XXX` directive informs the compiler what kind of assembler it can expect in an `asm` block. The `XXX` should be replaced by one of the following:

att Indicates that `asm` blocks contain AT&T syntax assembler.

intel Indicates that `asm` blocks contain Intel syntax assembler.

direct Tells the compiler that asm blocks should be copied directly to the assembler file.

These switches are local, and retain their value to the end of the unit that is compiled, unless they are replaced by another directive of the same type. The command-line switch that corresponds to this switch is `-R`.

$B or $BOOLEVAL: Complete boolean evaluation

This switch is understood by the Free Pascal compiler, but is ignored. The compiler always uses shortcut evaluation, i.e. the evaluation of a boolean expression is stopped once the result of the total exression is known with certainty.

So, in the following example, the function `Bofu`, which has a boolean result, will never get called.

```
If False and Bofu then
  ...
```

$C or $ASSERTIONS : Assertion support

This switch is recognised for Delphi compatibility only. Assertions are not yet supported by the compiler, but will be implemented in the future.

$DEFINE : Define a symbol

The directive

```
{$DEFINE name}
```

defines the symbol `name`. This symbol remains defined until the end of the current module, or until a `$UNDEF name` directive is encountered.

If `name` is already defined, this has no effect. `Name` is case insensitive.

$ELSE : Switch conditional compilation

The {`$ELSE` } switches between compiling and ignoting the source text delimited by the preceding {`$IFxxx`} and following {`$ENDIF`}. Any text after the `ELSE` keyword but before the brace is ignored:

```
{$ELSE some ignored text}
```

is the same as

```
{$ELSE}
```

This is useful for indication what switch is meant.

$ENDIF : End conditional compilation

The {`$ENDIF`} directive ends the conditional compilation initiated by the last {`$IFxxx`} directive. Any text after the `ENDIF` keyword but before the closing brace is ignored:

```
{$ENDIF some ignored text}
```

is the same as

```
{$ENDIF}
```

This is useful for indication what switch is meant to be ended.

$ERROR : Generate error message

The following code

```
{$ERROR This code is erroneous !}
```

will display an error message when the compiler encounters it, and increase the error count of the compiler. The compiler will continue to compile, but no code will be emitted.

$F : Far or near functions

This directive is recognized for compatibility with Turbo Pascal. Under the 32-bit programming model, the concept of near and far calls have no meaning, hence the directive is ignored. A warning is printed to the screen, telling you so.

As an example, : the following piece of code :

```
{$F+}

Procedure TestProc;

begin
 Writeln ('Hello From TestProc');
end;

begin
 testProc
end.
```

Generates the following compiler output:

```
malpertuus: >pp -vw testf
Compiler: ppc386
Units are searched in: /home/michael;/usr/bin/;/usr/lib/ppc/0.9.1/linuxunits
Target OS: Linux
Compiling testf.pp
testf.pp(1) Warning: illegal compiler switch
7739 kB free
Calling assembler...
Assembled...
Calling linker...
12 lines compiled,
 1.00000000000000E+0000
```

You can see that the verbosity level was set to display warnings.

If you declare a function as `Far` (this has the same effect as setting it between {$F+}...{$F-} directives), the compiler also generates a warning :

```
testf.pp(3) Warning: FAR ignored
```

The same story is true for procedures declared as `Near`. The warning displayed in that case is:

```
testf.pp(3) Warning: NEAR ignored
```

$FATAL : Generate fatal error message

The following code

```
{$FATAL This code is erroneous !}
```

will display an error message when the compiler encounters it, and trigger and increase the error count of the compiler. The compiler will immediatly stop the compilation process.

$H or $LONGSTRINGS : Use AnsiStrings

If {`$LONGSTRINGS ON`} is specified, the keyword `String` (no length specifier) will be treated as `AnsiString`, and the compiler will treat the corresponding varible as an ansistring, and will generate corresponding code.

By default, the use of ansistrings is off, corresponding to {`$H-`}.

This feature is still experimental, and should be used with caution for the time being.

$HINT : Generate hint message

If the generation of hints is turned on, through the `-vh` command-line option or the {`$HINTS ON`} directive, then

```
{$Hint This code should be optimized }
```

will display a hint message when the compiler encounters it.

$HINTS : Emit hints

{`$HINTS ON`} switches the generation of hints on. {`$HINTS OFF`} switches the generation of hints off. Contrary to the command-line option `-vh` this is a local switch, this is useful for checking parts of your code.

$IF : Start conditional compilation

The directive {`$IF expr`} will continue the compilation if the boolean expression `expr` evaluates to `true`. If the compilation evaluates to false, then the source are skipped to the first {`$ELSE`} or {`$ENDIF`} directive.

The compiler must be able to evaluate the expression at compile time. This means that you cannot use variables or constants that are defined in the source. Macros and symbols may be used, however.

More information on this can be found in the section about conditionals.

$IFDEF : Start conditional compilation

The {$IFDEF name} will skip the compilation of the text that follows it if the symbol name is not defined. If it is defined, then compilation continues as if the directive wasn't there.

$IFNDEF : Start conditional compilation

The {$IFNDEF name} will skip the compilation of the text that follows it if the symbol name is defined. If it is not defined, then compilation continues as if the directive wasn't there.

$IFOPT : Start conditional compilation

The {$IFOPT switch} will compile the text that follows it if the switch switch is currently in the specified state. If it isn't in the specified state, then compilation continues after the corresponding {$ENDIF} directive.

As an example:

```
{$IFOPT M+}
  Writeln ('Compiled with type information');
{$ENDIF}
```

Will compile the writeln statement if generation of type information is on.

$INFO : Generate info message

If the generation of info is turned on, through the -vi command-line option, then

```
{$INFO This was coded on a rainy day by Bugs Bunny }
```

will display an info message when the compiler encounters it.

$I or $IOCHECK : Input/Output checking

The {$I-} or {$IOCHECK OFF} directive tells the compiler not to generate input/output checking code in your program. By default, the compiler does not generate this code, you must switch it on using the -Ci command-lne switch.

If you compile using the -Ci compiler switch, the Free Pascal compiler inserts input/output checking code after every input/output call in your program. If an error occurred during input or output, then a run-time error will be generated. Use this switch if you wish to avoid this behavior. If you still want to check if something went wrong, you can use the IOResult function to see if everything went without problems.

Conversely, {$I+} will turn error-checking back on, until another directive is encountered which turns it off again.

The most common use for this switch is to check if the opening of a file went without problems, as in the following piece of code:

...

```
assign (f,'file.txt');
{$I-}
rewrite (f);
{$I+}
if IOResult<>0 then
  begin
  Writeln ('Error opening file : "file.txt"');
  exit
  end;
...
```

$I or $INCLUDE : Include file

The {$I filename} or {$INCLUDE filename} directive tells the compiler to read further statements from the file `filename`. The statements read there will be inserted as if they occurred in the current file.

The compiler will append the .pp extension to the file if you don't specify an extension yourself. Do not put the filename between quotes, as they will be regarded as part of the file's name.

You can nest included files, but not infinitely deep. The number of files is restricted to the number of file descriptors available to the Free Pascal compiler.

Contrary to Turbo Pascal, include files can cross blocks. I.e. you can start a block in one file (with a `Begin` keyword) and end it in another (with a `End` keyword). The smallest entity in an include file must be a token, i.e. an identifier, keyword or operator.

The compiler will look for the file to include in the following places:

1. It will look in the path specified in the incude file name.
2. It will look in the directory where the current source file is.
3. it will look in all directories specified in the include file search path.

You can add files to the include file search path with the `-I` command-line option.

$I or $INCLUDE : Include compiler info

In this form:

```
{$INCLUDE %xxx%}
```

where `xxx` is one of `TIME`, `DATE`, `FPCVERSION` or `FPCTARGET`, will generate a macro with the value of these things. If `xxx` is none of the above, then it is assumed to be the value of an environment variable. It's value will be fetched, and inserted in the coe as if it were a srtring.

For example, the following program

```
Program InfoDemo;

Const User = {$I %USER%};
```

```
begin
  Write ('This program was comilped at ',{$I %TIME%});
  Writeln (' on ',{$I %DATE%});
  Writeln ('By ',User);
  Writeln ('Compiler version : ',{$I %FPCVERSION%});
  Writeln ('Target CPU : ',{$I %FPCTARGET%});
end.
```

Creates the following output :

```
This program was comilped at 17:40:18 on 1998/09/09
By michael
Compiler version : 0.99.7
Target CPU : i386
```

$I386_XXX : Specify assembler format

This switch selects the assembler reader. {$I386_XXX} has the same effect as {$ASMMODE XXX}, section 1.1

$L or $LINK : Link object file

The {$L filename} or {$LINK filename} directive tells the compiler that the file filename should be linked to your program.

the compiler will look for this file in the following way:

1. It will look in the path specified in the object file name.
2. It will look in the directory where the current source file is.
3. it will look in all directories specified in the object file search path.

You can add files to the object file search path with the -Fo option.

On LINUX systems, the name is case sensitive, and must be typed exactly as it appears on your system.

Remark : Take care that the object file you're linking is in a format the linker understands. Which format this is, depends on the platform you're on. Typing ld on the command line gives a list of formats ld knows about.

You can pass other files and options to the linker using the -k command-line option. You can specify more than one of these options, and they will be passed to the linker, in the order that you specified them on the command line, just before the names of the object files that must be linked.

$LINKLIB : Link to a library

The {$LINKLIB name} will link to a library name. This has the effect of passing -lname to the linker.

As an example, consider the following unit:

```
unit getlen;
```

```
interface
{$LINKLIB c}

function strlen (P : pchar) : longint;cdecl;

implementation

function strlen (P : pchar) : longint;cdecl;external;

end.
```

If one would issue the command the command

```
ppc386 foo.pp
```

where foo.pp has the above unit in its `uses` clause, then the compiler would link your program to the c library, by passing the linker the `-lc` option.

The same effect could be obtained by removing the linklib directive in the above unit, and specify `-k-lc` on the command-line:

```
ppc386 -k-lc foo.pp
```

`$M` or `$TYPEINFO` : Generate type info

This switch is recognized for Delphi compatibility only since the generation of type information isn't fully implemented yet.

`$MESSAGE` : Generate info message

If the generation of info is turned on, through the `-vi` command-line option, then

```
{$MESSAGE This was coded on a rainy day by Bugs Bunny }
```

will display an info message when the compiler encounters it. The effect is the same as the {`$INFO`} directive.

`$MMX` : Intel MMX support

As of version 0.9.8, Free Pascal supports optimization for the **MMX** Intel processor (see also 7).

This optimizes certain code parts for the **MMX** Intel processor, thus greatly improving speed. The speed is noticed mostly when moving large amounts of data. Things that change are

- Data with a size that is a multiple of 8 bytes is moved using the `movq` assembler instruction, which moves 8 bytes at a time

Remark that MMX support is NOT emulated on non-MMX systems, i.e. if the processor doesn't have the MMX extensions, you cannot use the MMX optimizations.

When **MMX** support is on, you aren't allowed to do floating point arithmetic. You are allowed to move floating point data, but no arithmetic can be done. If you wish

to do floating point math anyway, you must first switch of **MMX** support and clear the FPU using the `emms` function of the `cpu` unit.

The following example will make this more clear:

```
Program MMXDemo;

uses cpu;

var
   d1 : double;
   a : array[0..10000] of double;
   i : longint;

begin
   d1:=1.0;
{$mmx+}
   { floating point data is used, but we do _no_ arithmetic }
   for i:=0 to 10000 do
     a[i]:=d2;  { this is done with 64 bit moves }
{$mmx-}
   emms;   { clear fpu }
   { now we can do floating point arithmetic }
   ....
end.
```

See, however, the chapter on MMX (7) for more information on this topic.

$NOTE : Generate note message

If the generation of notes is turned on, through the `-vn` command-line option or the {`$NOTES ON`} directive, then

```
{$NOTE Ask Santa Claus to look at this code }
```

will display a note message when the compiler encounters it.

$NOTES : Emit notes

{`$NOTES ON`} switches the generation of notes on. {`$NOTES OFF`} switches the generation of notes off. Contrary to the command-line option `-vn` this is a local switch, this is useful for checking parts of your code.

$OUTPUT_FORMAT : Specify the output format

{`$OUTPUT_FORMAT format`} has the same functionality as the `-A` command-line option : It tells the compiler what kind of object file must be generated. You can specify this switch **only** befor the `Program` or `Unit` clause in your source file. The different kinds of formats are shown in table (1.1) .

Table 1.1: Formats generated by the x86 compiler

Switch value	Generated format
att	AT&T assembler file.
o	Unix object file.
obj	OMF file.
wasm	assembler for the Watcom assembler.

`$P` or `$OPENSTRINGS` : Use open strings

`$PACKENUM` : Minimum enumeration type size

This directive tells the compiler the minimum number of bytes it should use when storing enumerated types. It is of the following form:

```
{$PACKENUM xxx}
{$MINENUMSIZE xxx}
```

Where the form with `$MINENUMSIZE` is for Delphi compatibility. varxxx can be one of `1,2` or `4`, or `NORMAL` or `DEFAULT`, corresponding to the default value of 4.

As an alternative form one can use {`$Z1`}, {`$Z2`} {`$Z4`}. Contrary to Delphi, the default size is 4 bytes ({`$Z4`}).

So the follwoing code

```
{$PACKENUM 1}
Type
  Days = (monday, tuesday, wednesday, thursday, friday,
          saturday, sunday);
```

will use 1 byte to store a variable of type `Days`, wheras it nomally would use 4 bytes. The above code is equivalent to

```
{$Z1}
Type
  Days = (monday, tuesday, wednesday, thursday, friday,
          saturday, sunday);
```

Remark: Sets are always put in 32 bit or 32 bytes, this cannot be changed

`$PACKRECORDS` : Alignment of record elements

This directive controls the byte alignment of the elements in a record, object or class type definition.

It is of the following form:

```
{$PACKRECORDS n}
```

Where `n` is one of 1,2,4,16 or `NORMAL` or `DEFAULT`. This means that the elements of a record that have size greater than `n` will be aligned on `n` byte boundaries. Elements with size less than or equal to `n` will be aligned to a natural boundary, i.e. to a power of two that is equal to or larger than the element's size.

The default alignment (which can be selected with `DEFAULT`) is 2, contrary to Turbo Pascal, where it is 1.

More information on this and an exmple program can be found in the reference guide, in the section about record types.

Remark: Sets are always put in 32 bit or 32 bytes, this cannot be changed

`$Q $OVERFLOWCHECKS`: Overflow checking

The {`$Q+`} or {`$OVERFLOWCHECKS ON`} directive turns on integer overflow checking. This means that the compiler inserts code to check for overflow when doing computations with integers. When an overflow occurs, the run-time library will print a message `Overflow at xxx`, and exit the program with exit code 215.

Remark: Overflow checking behaviour is not the same as in Turbo Pascal since all arithmetic operations are done via 32-bit values. Furthermore, the Inc() and Dec() standard system procedures *are* checked for overflow in Free Pascal, while in Turbo Pascal they are not.

Using the {`$Q-`} switch switches off the overflow checking code generation.

The generation of overflow checking code can also be controlled using the `-Co` command line compiler option (see Users' guide).

`$R` or `$RANGECHECKS` : Range checking

By default, the computer doesn't generate code to check the ranges of array indices, enumeration types, subrange types, etc. Specifying the {`$R+`} switch tells the computer to generate code to check these indices. If, at run-time, an index or enumeration type is specified that is out of the declared range of the compiler, then a run-time error is generated, and the program exits with exit code 201.

The {`$RANGECHECKS OFF`} switch tells the compiler not to generate range checking code. This may result in faulty program behaviour, but no run-time errors will be generated.

Remark: Range checking for sets and enumerations are not yet fully implemented.

`$SATURATION` : Saturation operations

This works only on the intel compiler, and MMX support must be on ({`$MMX +`}) for this to have any effect. See the section on saturation support (section 7.2) for more information on the effect of this directive.

`$STOP` : Generate fatal error message

The following code

```
{$STOP This code is erroneous !}
```

will display an error message when the compiler encounters it. The compiler will immediatly stop the compilation process.

It has the same effect as the {`$FATAL`} directive.

$T or $TYPEDADDRESS : Typed address operator (@)

In the {$T+} or {$TYPEDADDRESS ON} state the @ operator, when applied to a variable, returns a result of type ^T, if the type of the variable is T. In the {$T-} state, the result is always an untyped pointer, which is assignment compatible with all other pointer types.

$UNDEF : Undefine a symbol

The directive

```
{$UNDEF name}
```

un-defines the symbol name if it was previously defined. Name is case insensitive.

$V or $VARSTRINGCHECKS : Var-string checking

When in the + or ON state, the compiler checks that strings passed as parameters are of the same, identical, string type as the declared parameters of the procedure.

$WAIT : Wait for enter key press

If the compiler encounters a

```
{$WAIT }
```

directive, it will resume compiling only after the user has pressed the enter key. If the generation of info messages is turned on, then the compiler will display the follwing message:

```
Press <return> to continue
```

before waiting for a keypress. Careful ! this may interfere with automatic compilation processes. It should be used for debuggig purposes only.

$WARNING : Generate warning message

If the generation of warnings is turned on, through the -vw command-line option or the {$WARNINGS ON} directive, then

```
{$WARNING This is dubious code }
```

will display a warning message when the compiler encounters it.

$WARNINGS : Emit warnings

{$WARNINGS ON} switches the generation of warnings on. {$WARNINGS OFF} switches the generation of warnings off. Contrary to the command-line option -vw this is a local switch, this is useful for checking parts of your code.

$X or $EXTENDEDSYNTAX : Extended syntax

Extended syntax allows you to drop the result of a function. This means that you can use a function call as if it were a procedure. Standard this feature is on. You can switch it off using the {$X-} or {$EXTENDEDSYNTAX OFF}directive.

The following, for instance, will not compile :

```
function Func (var Arg : sometype) : longint;
begin
...          { declaration of Func }
end;

...

{$X-}
Func (A);
```

The reason this construct is supported is that you may wish to call a function for certain side-effects it has, but you don't need the function result. In this case you don't need to assign the function result, saving you an extra variable.

The command-line compiler switch -Sa1 has the same effect as the {$X+} directive.

1.2 Global directives

Global directives affect the whole of the compilation process. That is why they also have a command - line counterpart. The command-line counterpart is given for each of the directives.

$D or $DEBUGINFO: Debugging symbols

When this switch is on ({$DEBUGINFO ON}), the compiler inserts GNU debugging information in the executable. The effect of this switch is the same as the command-line switch -g. By default, insertion of debugging information is off.

$E : Emulation of coprocessor

This directive controls the emulation of the coprocessor. There is no command-line counterpart for this directive.

Intel x86 version

When this switch is enabled, all floating point instructions which are not supported by standard coprocessor emulators will give out a warning.

The compiler itself doesn't do the emulation of the coprocessor.

To use coprocessor emulation under DOS go32v1 there is nothing special required, as it is handled automatically. (As of version 0.99.10, the go32v1 platform will no longer be supported)

To use coprocessor emulation under DOS go32v2 you must use the emu387 unit, which contains correct initialization code for the emulator.

Under LINUX, the kernel takes care of the coprocessor support.

Motorola 680x0 version

When the switch is on, no floating point opcodes are emitted by the code generator. Instead, internal run-time library routines are called to do the necessary calculations. In this case all real types are mapped to the single IEEE floating point type.

Remark : By default, emulation is on. It is possible to intermix emulation code with real floating point opcodes, as long as the only type used is single or real.

`$G` : Generate 80286 code

This option is recognised for Turbo Pascal compatibility, but is ignored,

`$L` or `$LOCALSYMBOLS`: Local symbol information

This switch (not to be confused with the {`$L file`} file linking directive) is recognised for Turbo Pascal compatibility, but is ignored. generation of symbol information is controlled by the `$D` switch.

`$M` or `$MEMORY`: Memory sizes

This switch can be used to set the heap and stacksize. It's format is as follows:

```
{$M StackSize,HeapSize}
```

Wher `StackSize` and `HeapSize` should be two integer values, greater than 1024. The first number sets the size of the stack, and the second the size of the heap. (Stack setting is ignored under LINUX). The two numbers can be set on the command line using the `-Ch` and `-Cs` switches.

`$MODE` : Set compiler compatibility mode

The {`$MODE`} sets the compatibility mode of the compiler. This is equivalent to setting one of the command-line options `-So` or `-Sd` or `-S2`. it has the following arguments:

Default Default mode. This reverts back to the mode that was set on the command-line.

Delphi Delphi compatibility mode. All object-pascal extensions are enabled. This is the same as the command-line option `-Sd`.

TP Turbo pascal compatibility mode. Object pascal extensions are disabled, except ansistrings, which remain valid. This is the same as the command-line option `-So`

FPC FPC mode.

OBJFPC Object pascal mode. This is the same as the `-S2` command-line option.

GPC GNU pascal mode. This is the same as the `-Sp` command-line option.

For an exact description of each of these modes, see appendix D, on page 73

`$N` : Numeric processing

This switch is recognised for Turbo Pascal compatibility, but is otherwise ignored, since the compiler always uses the coprocessor for floating point mathematics.

`$O` : Overlay code generation

This switch is recognised for Turbo Pascal compatibility, but is otherwise ignored.

`$S` : Stack checking

The {`$S+`} directive tells the compiler to generate stack checking code. This generates code to check if a stack overflow occurred, i.e. to see whether the stack has grown beyond its maximally allowed size. If the stack grows beyond the maximum size, then a run-time error is generated, and the program will exit with exit code 202.

Specifying {`$S-`} will turn generation of stack-checking code off.

The command-line compiler switch `-Ct` has the same effect as the {`$S+`} directive.

`$W` or `$STACKFRAMES` : Generate stackframes

The {`$W`} switch directove controls the generation of stackframes. In the on state ({`$STACKFRAMES ON`}), the compiler will generate a stackframe for every procedure or function.

In the off state, the compiler will omit the generation of a stackframe if the following conditions are satisfied:

- The procedure has no parameters.
- The procedure has no local variables.
- If the procedure is not an `assembler` procedure, it must not have a `asm ... end;` block.
- it is not a constuctor or desctructor.

If these conditions are satisfied, the stack frame will be omitted.

`$Y` or `$REFERENCEINFO` : Insert Browser information

This switch controls the generation of browser inforation. It is recognized for compatibility with Turbo Pascal and Delphi only, as Browser information generation is not yet fully supported.

Chapter 2

Using conditionals, Messages and macros

The Free Pascal compiler supports conditionals as in normal Turbo Pascal. It does, however, more than that. It allows you to make macros which can be used in your code, and it allows you to define messages or errors which will be displayed when compiling.

2.1 Conditionals

The rules for using conditional symbols are the same as under Turbo Pascal. Defining a symbol goes as follows:

```
{$Define Symbol }
```

From this point on in your code, the compiler knows the symbol `Symbol`. Symbols are, like the Pascal language, case insensitive.

You can also define a symbol on the command line. the `-dSymbol` option defines the symbol `Symbol`. You can specify as many symbols on the command line as you want.

Undefining an existing symbol is done in a similar way:

```
{$Undef Symbol }
```

If the symbol didn't exist yet, this doesn't do anything. If the symbol existed previously, the symbol will be erased, and will not be recognized any more in the code following the `{$Undef ...}` statement.

You can also undefine symbols from the command line with the `-u` command-line switch..

To compile code conditionally, depending on whether a symbol is defined or not, you can enclose the code in a `{$ifdef Symbol}` .. `{$endif}` pair. For instance the following code will never be compiled :

```
{$Undef MySymbol}
{$ifdef Mysymbol}
  DoSomething;
```

Table 2.1: Symbols defined by the compiler.

Free
VERv
VERv_r
VERv_r_p
OS

```
  ...
{$endif}
```

Similarly, you can enclose your code in a `{$Ifndef Symbol}` .. `{$endif}` pair. Then the code between the pair will only be compiled when the used symbol doesn't exist. For example, in the following example, the call to the `DoSomething` will always be compiled:

```
{$Undef MySymbol}
{$ifndef Mysymbol}
  DoSomething;
  ...
{$endif}
```

You can combine the two alternatives in one structure, namely as follows

```
{$ifdef Mysymbol}
  DoSomething;
{$else}
  DoSomethingElse
{$endif}
```

In this example, if `MySymbol` exists, then the call to `DoSomething` will be compiled. If it doesn't exist, the call to `DoSomethingElse` is compiled.

The Free Pascal compiler defines some symbols before starting to compile your program or unit. You can use these symbols to differentiate between different versions of the compiler, and between different compilers. In table (2.1) , a list of pre-defined symbols is given[1]. In that table, you should change `v` with the version number of the compiler you're using, `r` with the release number and `p` with the patch-number of the compiler. 'OS' needs to be changed by the type of operating system. Currently this can be one of `DOS`, `GO32V2`, `LINUX`, `OS2`, `WIN32`, `MACOS`, `AMIGA` or `ATARI`.

This symbol is undefined if you specify a target that is different from the platform you're compiling on. The `-TSomeOS` option on the command line will define the `SomeOS` symbol, and will undefine the existing platform symbol[2].

As an example : Version 0.9.1 of the compiler, running on a Linux system, defines the following symbols before reading the command line arguments: `FPC`, `VER0`, `VER0_9`, `VER0_9_1` and `LINUX`. Specifying `-TOS2` on the command-line will undefine the `LINUX` symbol, and will define the `OS2` symbol.

Remark: Symbols, even when they're defined in the interface part of a unit, are not available outside that unit.

[1] Remark: The `FPK` symbol is still defined for compatibility with older versions.
[2] In versions prior to 0.9.4, this didn't happen, thus making Cross-compiling impossible.

Except for the Turbo Pascal constructs, from version 0.9.8 and higher, the Free Pascal compiler also supports a stronger conditional compile mechanism: The {`$If` } construct.

The prototype of this construct is as follows :

```
{$If expr}
  CompileTheseLines;
{$else}
  BetterCompileTheseLines;
{$endif}
```

In this directive `expr` is a Pascal expression which is evaluated using strings, unless both parts of a comparision can be evaluated as numbers, in which case they are evaluated using numbers[3]. If the complemete expression evaluates to `'0'`, then it is considered false and rejected. Otherwise it is considered true and accepted. This may have unexpected consequences :

```
{$If 0}
```

Will evaluate to `False` and be rejected, while

```
{$If 00}
```

Will evaluate to `True`.

You can use any Pascal operator to construct your expression : `=`, `<>`, `>`, `<`, `>=`, `<=`, `AND`, `NOT`, `OR` and you can use round brackets to change the precedence of the operators.

The following example shows you many of the possibilities:

```
{$ifdef fpc}

var
   y : longint;
{$else fpc}

var
   z : longint;
{$endif fpc}

var
   x : longint;

begin

{$if (fpc_version=0) and (fpc_release>6) and (fpc_patch>4)}
{$info At least this is version 0.9.5}
{$else}
{$fatalerror Problem with version check}
{$endif}

{$define x:=1234}
```

[3] Otherwise {`$If 8>54` would evaluate to `True`

```
{$if x=1234}
{$info x=1234}
{$else}
{$fatalerror x should be 1234}
{$endif}

{$if 12asdf and 12asdf}
{$info $if 12asdf and 12asdf is ok}
{$else}
{$fatalerror $if 12asdf and 12asdf rejected}
{$endif}

{$if 0 or 1}
{$info $if 0 or 1 is ok}
{$else}
{$fatalerror $if 0 or 1 rejected}
{$endif}

{$if 0}
{$fatalerror $if 0 accepted}
{$else}
{$info $if 0 is ok}
{$endif}

{$if 12=12}
{$info $if 12=12 is ok}
{$else}
{$fatalerror $if 12=12 rejected}
{$endif}

{$if 12<>312}
{$info $if 12<>312 is ok}
{$else}
{$fatalerror $if 12<>312 rejected}
{$endif}


{$if 12<=312}
{$info $if 12<=312 is ok}
{$else}
{$fatalerror $if 12<=312 rejected}
{$endif}

{$if 12<312}
{$info $if 12<312 is ok}
{$else}
{$fatalerror $if 12<312 rejected}
{$endif}

{$if a12=a12}
{$info $if a12=a12 is ok}
{$else}
{$fatalerror $if a12=a12 rejected}
{$endif}
```

```
{$if a12<=z312}
{$info $if a12<=z312 is ok}
{$else}
{$fatalerror $if a12<=z312 rejected}
{$endif}


{$if a12<z312}
{$info $if a12<z312 is ok}
{$else}
{$fatalerror $if a12<z312 rejected}
{$endif}

{$if not(0)}
{$info $if not(0) is OK}
{$else}
{$fatalerror $if not(0) rejected}
{$endif}

{$info ************************************************}
{$info * Now have to follow at least 2 error messages: *}
{$info ************************************************}

{$if not(0}
{$endif}

{$if not(<}
{$endif}

end.
```

As you can see from the example, this construct isn't useful when used with normal symbols, but it is if you use macros, which are explained in section 2.3, they can be very useful. When trying this example, you must switch on macro support, with the `-Sm` command-line switch.

2.2 Messages

Free Pascal lets you define normal, warning and error messages in your code. Messages can be used to display useful information, such as copyright notices, a list of symbols that your code reacts on etc.

Warnings can be used if you think some part of your code is still buggy, or if you think that a certain combination of symbols isn't useful. In general anything which may cause problems when compiling.

Error messages can be useful if you need a certain symbol to be defined to warn that a certain variable isn't defined or so, or when the compiler version isn't suitable for your code.

The compiler treats these messages as if they were generated by the compiler. This means that if you haven't turned on warning messages, the warning will not e displayed. Errors are always displayed, and the compiler stops as if an error had occurred.

For messages, the syntax is as follows :

```
{$Message Message text }
```

Or

```
{$Info Message text }
```

For notes:

```
{$Note Message text }
```

For warnings:

```
{$Warning Warning Message text }
```

For errors :

```
{$Error  Error Message text }
```

Lastly, for fatal errors :

```
{$FatalError  Error Message text }
```

or

```
{$Stop  Error Message text }
```

The difference between `$Error` and `$FatalError` or `$Stop` messages is that when the compiler encounters an error, it still continues to compile. With a fatal error, the compiler stops.

Remark : You cannot use the '}' character in your message, since this will be treated as the closing brace of the message.

As an example, the following piece of code will generate an error when the symbol `RequiredVar` isn't defined:

```
{$ifndef RequiredVar}
{$Error Requiredvar isn't defined !}
{$endif}
```

But the compiler will continue to compile. It will not, however, generate a unit file or a program (since an error occurred).

2.3 Macros

Macros are very much like symbols in their syntax, the difference is that macros have a value whereas a symbol simply is defined or is not defined. If you want macro support, you need to specify the `-Sm` command-line switch, otherwise your macro will be regarded as a symbol.

Defining a macro in your program is done in the same way as defining a symbol; in a {`$define` } preprocessor statement[4]:

[4]In compiler versions older than 0.9.8, the assignment operator for a macros wasn't `:=`, but `=`

Table 2.2: Predefined macros

Symbol	Contains
`FPC_VERSION`	The version number of the compiler.
`FPC_RELEASE`	The release number of the compiler.
`FPC_PATCH`	The patch number of the compiler.

```
{$define ident:=expr}
```

If the compiler encounters `ident` in the rest of the source file, it will be replaced immediately by `expr`. This replacement works recursive, meaning that when the compiler expanded one of your macros, it will look at the resulting expression again to see if another replacement can be made. You need to be careful with this, because an infinite loop can occur in this manner.

Here are two examples which illustrate the use of macros:

```
{$define sum:=a:=a+b;}
...
sum          { will be expanded to 'a:=a+b;'
               remark the absence of the semicolon}
...
{$define b:=100}
sum          { Will be expanded recursively to a:=a+100; }
...
```

The previous example could go wrong :

```
{$define sum:=a:=a+b;}
...
sum          { will be expanded to 'a:=a+b;'
               remark the absence of the semicolon}
...
{$define b=sum} { DON'T do this !!!}
sum          { Will be infinitely recursively expanded... }
...
```

On my system, the last example results in a heap error, causing the compiler to exit with a run-time error 203.

Remark: Macros defined in the interface part of a unit are not available outside that unit ! They can just be used as a notational convenience, or in conditional compiles.

By default, from version 0.9.8 of the compiler on, the compiler predefines three macros, containing the version number, the release number and the patch number. They are listed in table (2.2) .

Remark: Don't forget that macros support isn't on by default. You need to compile with the `-Sm` command-line switch.

Chapter 3

Using Assembly language

Free Pascal supports inserting of assembler instructions in your code. The mechanism for this is the same as under Turbo Pascal. There are, however some substantial differences, as will be explained in the following.

3.1 Intel syntax

As of version 0.9.7, Free Pascal supports Intel syntax for the Intel family of Ix86 processors in it's `asm` blocks.

The Intel syntax in your `asm` block is converted to AT&T syntax by the compiler, after which it is inserted in the compiled source. The supported assembler constructs are a subset of the normal assembly syntax. In what follows we specify what constructs are not supported in Free Pascal, but which exist in Turbo Pascal:

- The `TBYTE` qualifier is not supported.
- The `&` identifier override is not supported.
- The `HIGH` operator is not supported.
- The `LOW` operator is not supported.
- The `OFFSET` and `SEG` operators are not supported. use `LEA` and the various `Lxx` instructions instead.
- Expressions with constant strings are not allowed.
- Access to record fields via parenthesis is not allowed
- Typecasts with normal pascal types are not allowed, only recognized assembler typecasts are allowed.
  Example:

```
mov al, byte ptr MyWord       -- allowed,
mov al, byte(MyWord)          -- allowed,
mov al, shortint(MyWord)      -- not allowed.
```

- Pascal type typecasts on constants are not allowed.
  Example:

```
const s= 10; const t = 32767;
```

in Turbo Pascal:

```
mov al, byte(s)            -- useless typecast.
mov al, byte(t)            -- syntax error!
```

In this parser, either of those cases will give out a syntax error.

- Constant references expressions with constants only are not allowed (in all cases they do not work in protected mode, under linux i386).
  Examples:

  ```
  mov al,byte ptr ['c']      -- not allowed.
  mov  al,byte ptr [100h]    -- not allowed.
  ```

  (This is due to the limitation of Turbo Assembler).

- Brackets within brackets are not allowed

- Expressions with segment overrides fully in brackets are presently not supported, but they can easily be implemented in BuildReference if requested.
  Example:

  ```
  mov al,[ds:bx]     -- not allowed
  ```

  use instead:

  ```
  mov al,ds:[bx]
  ```

- Possible allowed indexing are as follows:
  - `Sreg:[REG+REG*SCALING+/-disp]`
  - `SReg:[REG+/-disp]`
  - `SReg:[REG]`
  - `SReg:[REG+REG+/-disp]`
  - `SReg:[REG+REG*SCALING]`

  Where `Sreg` is optional and specifies the segment override. *Notes:*

  1. The order of terms is important contrary to Turbo Pascal.
  2. The Scaling value must be a value, and not an identifier to a symbol.
     Examples:

     ```
     const myscale = 1;
     ...
     mov al,byte ptr [esi+ebx*myscale] -- not allowed.
     ```

     use:

     ```
     mov al, byte ptr [esi+ebx*1]
     ```

- Possible variable identifier syntax is as follows: (Id = Variable or typed constant identifier.)
  1. `ID`
  2. `[ID]`

3. `[ID+expr]`
4. `ID[expr]`

Possible fields are as follow:

1. `ID.subfield.subfield ...`
2. `[ref].ID.subfield.subfield ...`
3. `[ref].typename.subfield ...`

- Local Labels: Contrary to Turbo Pascal, local labels, must at least contain one character after the local symbol indicator.
  Example:

  ```
  @:                -- not allowed
  ```

  use instead, for example:

  ```
  @1:              -- allowed
  ```

- Contrary to Turbo Pascal local references cannot be used as references, only as displacements.
  example:

  ```
  lds si,@mylabel   -- not allowed
  ```

- Contrary to Turbo Pascal, `SEGCS`, `SEGDS`, `SEGES` and `SEGSS` segment overrides are presently not supported. (This is a planned addition though).
- Contrary to Turbo Pascal where memory sizes specifiers can be practically anywhere, the Free Pascal Intel inline assembler requires memory size specifiers to be outside the brackets.
  example:

  ```
  mov al,[byte ptr myvar]    -- not allowed.
  ```

  use:

  ```
  mov al,byte ptr [myvar]    -- allowed.
  ```

- Base and Index registers must be 32-bit registers. (limitation of the GNU Assembler).
- `XLAT` is equivalent to `XLATB`.
- Only Single and Double FPU opcodes are supported.
- Floating point opcodes are currently not supported (except those which involve only floating point registers).

The Intel inline assembler supports the following macros :

@Result represents the function result return value.

Self represents the object method pointer in methods.

3.2 AT&T Syntax

Free Pascal uses the GNU `as` assembler to generate its object files for the Intel Ix86 processors . Since the GNU assembler uses AT&T assembly syntax, the code you write should use the same syntax. The differences between AT&T and Intel syntax as used in Turbo Pascal are summarized in the following:

- The opcode names include the size of the operand. In general, one can say that the AT&T opcode name is the Intel opcode name, suffixed with a '`l`', '`w`' or '`b`' for, respectively, longint (32 bit), word (16 bit) and byte (8 bit) memory or register references. As an example, the Intel construct '`mov al bl` is equivalent to the AT&T style '`movb %bl,%al`' instruction.
- AT&T immediate operands are designated with '$', while Intel syntax doesn't use a prefix for immediate operands. Thus the Intel construct '`mov ax, 2`' becomes '`movb $2, %al`' in AT&T syntax.
- AT&T register names are preceded by a '`%`' sign. They are undelimited in Intel syntax.
- AT&T indicates absolute jump/call operands with '`*`', Intel syntax doesn't delimit these addresses.
- The order of the source and destination operands are switched. AT&T syntax uses '`Source, Dest`', while Intel syntax features '`Dest, Source`'. Thus the Intel construct '`add eax, 4`' transforms to '`addl $4, %eax`' in the AT&T dialect.
- Immediate long jumps are prefixed with the '`l`' prefix. Thus the Intel '`call/jmp section:offset'` is transformed to '`lcall/ljmp $section,$offset`'. Similarly the far return is '`lret`', instead of the Intel '`ret far`'.
- Memory references are specified differently in AT&T and Intel assembly. The Intel indirect memory reference

  ```
  Section:[Base + Index*Scale + Offs]
  ```

  is written in AT&T syntax as :

  ```
  Section:Offs(Base,Index,Scale)
  ```

  Where `Base` and `Index` are optional 32-bit base and index registers, and `Scale` is used to multiply `Index`. It can take the values 1,2,4 and 8. The `Section` is used to specify an optional section register for the memory operand.

More information about the AT&T syntax can be found in the `as` manual, although the following differences with normal AT&T assembly must be taken into account :

- Only the following directives are presently supported:

  .byte

  .word

  .long

  .ascii

  .asciz

  .globl

- The following directives are recognized but are not supported:

  .align

  .lcomm

  Eventually they will be supported.

- Directives are case sensitive, other identifiers are not case sensitive.
- Contrary to GAS local labels/symbols *must* start with `.L`
- The nor operator `'!'` is not supported.
- String expressions in operands are not supported.
- CBTW,CWTL,CWTD and CLTD are not supported, use the normal intel equivalents instead.
- Constant expressions which represent memory references are not allowed even though constant immediate value expressions are supported.
  examples:

```
const myid = 10;
...
movl $myid,%eax       -- allowed
movl myid(%esi),%eax  -- not allowed.
```

- When the `.globl` directive is found, the symbol following it is made public and is immediately emitted. Therefore label names with this name will be ignored.
- Only Single and Double FPU opcodes are supported.

The AT&T inline assembler supports the following macros :

__RESULT represents the function result return value.

__SELF represents the object method pointer in methods.

__OLDEBP represents the old base pointer in recusrive routines.

3.3 Calling mechanism

Procedures and Functions are called with their parameters on the stack. Contrary to Turbo Pascal, *all* parameters are pushed on the stack, and they are pushed *right* to *left*, instead of left to right for Turbo Pascal. This is especially important if you have some assembly subroutines in Turbo Pascal which you would like to translate to Free Pascal.

Function results are returned in the accumulator, if they fit in the register.

The registers are *not* saved when calling a function or procedure. If you want to call a procedure or function from assembly language, you must save any registers you wish to preserve.

The first thing a procedure does is saving the base pointer, and setting the base pointer equal to the stack pointer. References to the pushed parameters and local variables are constructed using the base pointer.

When the procedure or function exits, it clears the stack.

Table 3.1: Calling mechanisms in Free Pascal

Modifier	Pushing order	Stack cleaned by	Parameters in registers
(none)	Right-to-left	Function	No
cdecl	Right-to-left	Caller	No
export	Right-to-left	Caller	No
stdcall	Right-to-left	Function	No
popstack	Right-to-left	Caller	No

When you want your code to be called by a C library or used in a C program, you will run into trouble because of this calling mechanism. In C, the calling procedure is expected to clear the stack, not the called procedure. In other words, the arguments still are on the stack when the procedure exits. To avoid this problem, Free Pascal supports the `export` modifier. Procedures that are defined using the export modifier, use a C-compatible calling mechanism. This means that they can be called from a C program or library, or that you can use them as a callback function.

This also means that you cannot call this procedure or function from your own program, since your program uses the Pascal calling convention. However, in the exported function, you can of course call other Pascal routines.

As of version 0.9.8, the Free Pascal compiler supports also the `cdecl` and `stdcall` modifiers, as found in Delphi. The `cdecl` modifier does the same as the `export` modifier, and `stdcall` does nothing, since Free Pascal pushes the paramaters from right to left by default. In addition to the Delphi `cdecl` construct, Free Pascal also supports the `popstack` directive; it is nearly the same a the `cdecl` directive, only it still mangles the name, i.e. makes it into a name such as the compiler uses internally.

All this is summarized in table (3.1) . The first column lists the modifier you specify for a procedure declaration. The second one lists the order the paramaters are pushed on the stack. The third column specifies who is responsible for cleaning the stack: the caller or the called function. Finally, the last column specifies if registers are used to pass parameters to the function.

More about this can be found in chapter 4 on linking.

Ix86 calling conventions

Standard entry code for procedures and functions is as follows on the x86 architecture:

```
pushl   %ebp
movl    %esp,%ebp
```

The generated exit sequence for procedure and functions looks as follows:

```
leave
ret  $xx
```

Where `xx` is the total size of the pushed parameters.

To have more information on function return values take a look at the section 3.5 section.

M680x0 calling conventions

Standard entry code for procedures and functions is as follows on the 680x0 architecture:

```
move.l  a6,-(sp)
move.l  sp,a6
```

The generated exit sequence for procedure and functions looks as follows:

```
unlk   a6
move.l (sp)+,a0     ; Get return address
add.l  #xx,sp       ; Remove allocated stack
move.l a0,-(sp)     ; Put back return address on top of the stack
```

Where `xx` is the total size of the pushed parameters.

To have more information on function return values take a look at the section 3.5 section.

3.4 Signalling changed registers

When the compiler uses variables, it sometimes stores them, or the result of some calculations, in the processor registers. If you insert assembler code in your program that modifies the processor registers, then this may interfere with the compiler's idea about the registers. To avoid this problem, Free Pascal allows you to tell the compiler which registers have changed. The compiler will then avoid using these registers. Telling the compiler which registers have changed, is done by specifying a set of register names behind an assembly block, as follows:

```
asm
  ...
end ['R1',...,'Rn'];
```

Here `R1` to `Rn` are the names of the 32-bit registers you modify in your assembly code.

As an example :

```
asm
movl BP,%eax
movl 4(%eax),%eax
movl %eax,__RESULT
end ['EAX'];
```

This example tells the compiler that the `EAX` register was modified.

3.5 Register Conventions

The compiler has different register conventions, depending on the target processor used.

Intel x86 version

When optimizations are on, no register can be freely modified, without first being saved and then restored. Otherwise, EDI is usually used as a scratch register and can be freely used in assembler blocks.

Motorola 680x0 version

Registers which can be freely modified without saving are registers D0, D1, D6, A0, A1, and floating point registers FP2 to FP7. All other registers are to be considered reserved and should be saved and then restored when used in assembler blocks.

Chapter 4

Linking issues

When you only use Pascal code, and Pascal units, then you will not see much of the part that the linker plays in creating your executable. The linker is only called when you compile a program. When compiling units, the linker isn't invoked.

However, there are times that you want to C libraries, or to external object files that are generated using a C compiler (or even another pascal compiler). The Free Pascal compiler can generate calls to a C function, and can generate functions that can be called from C (exported functions). More on these calling conventions can be found in section 3.3.

In general, there are 2 things you must do to use a function that resides in an external library or object file:

1. You must make a pascal declaration of the function or procedure you want to use.
2. You must tell the compiler where the function resides, i.e. in what object file or what library, so the compiler can link the necessary code in.

The same holds for variables. To access a variable that resides in an external object file, you ust declare it, and tell the compiler where to find it. The following sections attempt to explain how to do this.

4.1 Using external functions or procedures

The first step in using external code blocks is declaring the function you want to use. Free Pascal supports Delphi syntax, i.e. you must use the `external` directive. The `external` directive replaces, in effect, the code block of the function. As such, It cannot be used in an interface section of a unit, but must always reside in the implementation section.

There exist four variants of the external direcive :

1. A simple external declaration:

   ```
   Procedure ProcName (Args : TPRocArgs); external;
   ```

   The `external` directive tells the compiler that the function resides in an external block of code. You can use this together with the {`$L` } or {`$LinkLib`

} directives to link to a function or procedure in a library or external object file.

2. You can give the `external` directive a library name as an argument:

   ```
   Procedure ProcName (Args : TPRocArgs); external 'Name';
   ```

   This tells the compiler that the procedure resides in a library with name `'Name'`. This method is equivalent to the following:

   ```
   Procedure ProcName (Args : TPRocArgs);external;
   {$LinkLib 'Name'}
   ```

3. The `external` can also be used with two arguments:

   ```
   Procedure ProcName (Args : TPRocArgs); external 'Name'
                                          name 'OtherProcName';
   ```

   This has the same meaning as the previous declaration, only the compiler will use the name `'OtherProcName'` when linking to the library. This can be used to give different names to procedures and functions in an external library.

   This method is equivalent to the following code:

   ```
   Procedure OtherProcName (Args : TProcArgs); external;
   {$LinkLib 'Name'}

   Procedure ProcName (Args : TPRocArgs);

   begin
     OtherProcName (Args);
   end;
   ```

4. Lastly, onder WINDOWS and OS/2, there is a fourth possibility to specify an external function: In .DLL files, functionas also have a unique number (their index). It is possible to refer to these fuctions using their index:

   ```
   Procedure ProcName (Args : TPRocArgs); external 'Name' Index SomeIndex;
   ```

   This tells the compiler that the procedure `ProcName` resides in a dynamic link library, with index SomeIndex.

   Remark: Note that this is ONLY available under WINDOWS *and* OS/2.

In earlier versions of the Free Pascal compiler, the following construct was also possible :

```
Procedure ProcName (Args : TPRocArgs); [ C ];
```

This method is equivalent to the following statement:

```
Procedure ProcName (Args : TPRocArgs); cdecl; external;
```

However, the `[ C ]` directive is no longer supported as of version 0.99.5 of Free Pascal, therefore you should use the `external` directive, with the `cdecl` directive, if needed.

4.2 Using external variables

Some libaries or code blocks have variables which they export. You can access these variables much in the same way as external functions. To access an external variable, you declare it as follows:

```
Var
  MyVar : MyType; external name 'varname';
```

The effect of this declaration is twofold:

1. No space is allocated for this variable.
2. The name of the variable used in the assebler code is `varname`. This is a case sensitive name, so you must be careful.

The variable will be accessible with it's declared name, i.e. `MyVar` in this case.

A second possibility is the declaration:

```
Var
  varname : MyType; cvar; external;
```

The effect of this declaration is twofold as in the previous case:

1. The `external` modifier ensures that no space is allocated for this variable.
2. The `cvar` modifier tells the compiler that the name of the variable used in the assebler code is exactly as specified in the declaration. This is a case sensitive name, so you must be careful.

In this case, you access the variable with it's C name, but case insensitive. The first possibility allows you to change the name of the external variable for internal use.

In order to be able to compile such statements, the compiler switch `-Sv` must be used.

As an example, let's look at the following C file (in extvar.c):

```
/*
Declare a variable, allocate storage
*/
int extvar = 12;
```

And the following program (in extdemo.pp):

```
Program ExtDemo;

{$L extvar.o}

Var { Case sensitive declaration !! }
    extvar : longint; cvar;external;
    I : longint; external name 'extvar';
begin
  { Extvar can be used case insensitive !! }
  Writeln ('Variable ''extvar'' has value : ',ExtVar);
  Writeln ('Variable ''I''      has value : ',i);
end.
```

Compiling the C file, and the pascal program:

```
gcc -c -o extvar.o extvar.c
ppc386 -Sv extdemo
```

Will produce a program extdemo which will print

```
Variable 'extvar' has value : 12
Variable 'I'      has value : 12
```

on your screen.

4.3 Linking to an object file

Having declared the external function or variable that resides in an object file, you can use it as if it was defined in your own program or unit. To produce an executable, you must still link the object file in. This can be done with the {`$L file.o`} directive.

This will cause the linker to link in the object file file.o. On LINUX systems, this filename is case sensitive. Under DOS, case isn't important. Note that `file.o` must be in the current directory if you don't specify a path. The linker will not search for file.o if it isn't found.

You cannot specify libraries in this way, it is for object files only.

Here we present an example. Consider that you have some assembly routine that calculates the nth Fibonacci number :

```
.text
.align 4
.globl Fibonacci
.type Fibonacci,@function
Fibonacci:
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
xorl %ecx,%ecx
xorl %eax,%eax
movl $1,%ebx
incl %edx
loop:
decl %edx
je endloop
movl %ecx,%eax
addl %ebx,%eax
movl %ebx,%ecx
movl %eax,%ebx
jmp loop
endloop:
movl %ebp,%esp
        popl %ebp
ret
```

Then you can call this function with the following Pascal Program:

```
Program FibonacciDemo;

var i : longint;

Function Fibonacci (L : longint):longint;cdecl;external;

{$L fib.o}

begin
  For I:=1 to 40 do
    writeln ('Fib(',i,') : ',Fibonacci (i));
end.
```

With just two commands, this can be made into a program :

```
as -o fib.o fib.s
ppc386 fibo.pp
```

This example supposes that you have your assembler routine in fib.s, and your Pascal program in fibo.pp.

4.4 Linking to a library

To link your program to a library, the procedure depends on how you declared the external procedure.

In case you used the follwing syntax to declare your procedure:

```
Procedure ProcName (Args : TPRocArgs); external 'Name';
```

You don't need to take additional steps to link your file in, the compiler will do all that is needed for you. On WINDOWSNT it will link to Name.dll, on LINUX your program will be linked to library libname, which can be a static or dynamic library.

In case you used

```
Procedure ProcName (Args : TPRocArgs); external;
```

You still need to explicity link to the library. This can be done in 2 ways:

1. You can tell the compiler in the source file what library to link to using the {`$LinkLib 'Name'`} directive:

   ```
   {$LinkLib 'gpm'}
   ```

   This will link to the gpm library. On LINUX systems, you needn't specify the extension or 'lib' prefix of the library. The compiler takes care of that. On DOS or WINDOWS systems, you need to specify the full name.

2. You can also tell the compiler on the command-line to link in a library: The `-k` option can be used for that. For example

   ```
   ppc386 -k'-lgpm' myprog.pp
   ```

   Is equivalent to the above method, and tells the linker to link to the gpm library.

As an example; consider the following program :

```
program printlength;

{$linklib c} { Case sensitive }

{ Declaration for the standard C function strlen }
Function strlen (P : pchar) : longint; cdecl;external;

begin
  Writeln (strlen('Programming is easy !'));
end.
```

This program can be compiled with :

```
ppc386  prlen.pp
```

Supposing, of course, that the program source resides in prlen.pp.

You cannot use procedures or functions that have a variable number of arguments in C. Pascal doesn't support this feature of C.

4.5 Making libraries

Free Pascal supports making shared or static libraries in a straightforward and easy manner. If you want to make libraries for other Free Pascal programmers, you just need to provide a command line switch. If you want C programmers to be able to use your code as well, you will need to adapt your code a little. This process is described first.

Exporting functions

When exporting functions from a library, there are 2 things you must take in account:

1. Calling conventions.
2. Naming scheme.

The calling conventions are controlled by the modifiers `cdecl`, `popstack`, `pascal`, `stdcall`. See section 3.3 for more information on the different kinds of calling scheme.

The naming conventions can be controlled by 3 modifiers:

cdecl: A function that has a `cdecl` modifier, will used with C calling conventions, that is, the caller clears the stack. Also the mangled name will be the name *exactly* as in the declaration. `cdecl` is part of the function declaration, and hence must be present both in the interface and implementation section of a unit.

export: A function that has an export modifier, uses also the exact declaration name as its mangled name. Under WINDOWSNT and OS/2, this modifier signals a function that is exported from a DLL. The calling conventions used by a `export` procedure depend on the OS. this keyword can be used only in the implementation section.

Alias: The `alias` modifier can be used to give a supplementary assembler name to your function. This doesn't modify the calling conventions of the function.

If you want to make your procedures and functions available to C programmers, you can do this very easily. All you need to do is declare the functions and procedures that you want to make available as `export`, as follows:

```
Procedure ExportedProcedure; export;
```

Remark : You can only declare a function as exported in the `Implementation` section of a unit. This function may *not* appear in the interface part of a unit. This is logical, since a Pascal routine cannot call an exported function, anyway.

However, the generated object file will not contain the name of the function as you declared it. The Free Pascal compiler "mangles" the name you give your function. It makes the name all-uppercase, and adds the types of all parameters to it. There are cases when you want to provide a mangled name without changing the calling convention. In such cases, you can use the `Alias` modifier.

The `Alias` modifier allows you to specify another name (a nickname) for your function or procedure.

The prototype for an aliased function or procedure is as follows :

```
Procedure AliasedProc; [ Alias : 'AliasName'];
```

The procedure `AliasedProc` will also be known as `AliasName`. Take care, the name you specify is case sensitive (as C is).

Remark: If you use in your unit functions that are in other units, or system functions, then the C program will need to link in the object files from the units too.

Exporting variables

Similarly as when you export functions, you can export variables. when exportig variables, one should only consider the names of the variables. To declare a variable that should be used by a C program, one declares it with the `cvar` modifier:

```
Var MyVar : MyTpe; cvar;
```

This will tell the compiler that the assembler name of the variable (the one which is used by C programs) should be exactly as specified in the declaration, i.e., case sensitive.

It is not allowed to declare multiple variables as `cvar` in one statement, i.e. the following code will produce an error:

```
var Z1,Z2 : longint;cvar;
```

Compiling libraries

Once you have your (adapted) code, with exported and other functions, you can compile your unit, and tell the compiler to make it into a library. The compiler will simply compile your unit, and perform the necessary steps to transform it into a `static` or `shared` (`dynamical`) library.

You can do this as follows, for a dynamical library:

```
ppc386 -CD myunit
```

On LINUX this will leave you with a file libmyunit.so. On WINDOWS and OS/2, this will leave you with myunit.dll.

If you want a static library, you can do

```
ppc386 -CS myunit
```

This will leave you with libmyunit.a and a file myunit.ppu. The myunit.ppu is the unit file needed by the Free Pascal compiler.

The resulting files are then libraries. To make static libraries, you need the ranlib or `ar` program on your system. It is standard on any LINUX system, and is provided with the GCC compiler under DOS. For the dos distribution, a copy of ar is included in the file gnuutils.zip.

BEWARE: This command doesn't include anything but the current unit in the library. Other units are left out, so if you use code from other units, you must deploy them together with your library.

Moving units into a library

You can put multiple units into a library with the `ppumove` command, as follows:

```
ppumove -e ppl -o name unit1 unit2 unit3
```

This will move 3 units in 1 library (called libname.so on linux, name.dll on WINDOWS) and it will create 3 files unit1.ppl, unit2.ppl and file3.ppl, which are unit files, but which tell the compiler to look in library `name` when linking your executable.

The `ppumove` program has options to create statical or dynammical libraries. It is provided with the compiler.

Unit searching strategy

When you compile a program or unit, the compiler will by default always look for .ppl files. If it doesn't find one, it will look for a .ppu file.

To be able to differentiate between units that have been compiled as static or dynamic libraries, there are 2 switches:

-XD: This will define the symbol `FPC_LINK_DYNAMIC`

-XS: This will define the symbol `FPC_LINK_STATIC`

Definition of one symbol will automatically undefine the other.

These two switches can be used in conjunction with the configuration file ppc386.cfg. The existence of one of these symbols can be used to decide which unit search path to set. For example:

```
# Set unit paths

#IFDEF FPC_LINK_STATIC
-Up/usr/lib/fpc/linuxunits/staticunits
```

```
#ENDIF
#IFDEF FPC_LINK_DYNAMIC
-Up/usr/lib/fpc/linuxunits/sharedunits
#ENDIF
```

With such a configuration file, the compiler will look for it's units in different directories, depending on whether `-XD` or `-XS` is used.

4.6 Using smart linking

You can compile your units using smart linking. When you use smartl linking, the compiler creates a series of code blocks that are as small as possible, i.e. a code block will contain only the code for one procedure or function.

When you compile a program that uses a smart-linked unit, the compiler will only link in the code that you actually need, and will leave out all other code. This will result in a smaller binary, which is loaded in memory faster, thus speeding up execution.

To enable smartlinking, one can give the smartlink option on the command line : `-Cx`, or one can put the {`$SMARTLINK ON`} directive in the unit file:

```
Unit Testunit

{SMARTLINK ON}
Interface
...
```

Smartlinking will slow down the compilation process, expecially for large units.

When a unit foo.pp is smartlinked, the name of the codefile is changed to libfoo.a.

Technically speaking, the compiler makes small assembler files for each procedure and function in the unit, as well as for all global defined variables (whether they're in the interface section or not). It then assembles all these small files, and uses ar to collect the resulting object fioles in one archive.

Smartlinking and the creation of shared (or dynamic) libraries are mutually exclusive, that is, if you turn on smartlinking, then the creation of shared libraries is turned of. The creation of static libraries is still possible. The reason for this is that it has little sense in making a smarlinked dynamica library. The whole shared library is loaded into memory anyway by the dynamic linker (or WINDOWSNT), so there would be no gain in size by making it smartinked.

Chapter 5

Objects

In this short chapter we give some technical things about objects. For instructions on how to use and declare objects, see Reference guide.

5.1 Constructor and Destructor calls

When using objects that need virtual methods, the compiler uses two help procedures that are in the run-time library. They are called `Help_Destructor` and `Help_Constructor`, and they are written in assembly language. They are used to allocate the necessary memory if needed, and to insert the Virtual Method Table (VMT) pointer in the newly allocated object.

When the compiler encounters a call to an object's constructor, it sets up the stack frame for the call, and inserts a call to the `Help_Constructor` procedure before issuing the call to the real constructor. The helper procedure allocates the needed memory (if needed) and inserts the VMT pointer in the object. After that, the real constructor is called.

A call to `Help_Destructor` is inserted in every destructor declaration, just before the destructor's exit sequence.

5.2 Memory storage of objects

Objects are stored in memory just as ordinary records with an extra field : a pointer to the Virtual Method Table (VMT). This field is stored first, and all fields in the object are stored in the order they are declared. This field is initialized by the call to the object's `Constructor` method.

If the object you defined has no virtual methods, then a `nil` is stored in the VMT pointer. This ensures that the size of objects is equal, whether they have virtual methods ore not.

The memory allocated looks as in table (5.1) .

5.3 The Virtual Method Table

The Virtual Method Table (VMT) for each object type consists of 2 check fields (containing the size of the data), a pointer to the object's anchestor's VMT (`Nil`

Table 5.1: Object memory layout

Offset	What
+0	Pointer to VMT.
+4	Data. All fields in the order the've been declared.
...	

Table 5.2: Virtual Method Table memory layout

Offset	What
+0	Size of object type data
+4	Minus the size of object type data. Enables determining of valid VMT pointers.
+8	Pointer to ancestor VMT, `Nil` if no ancestor available.
+12	Pointers to the virtual methods.
...	

if there is no anchestor), and then the pointers to all virtual methods. The VMT layout is illustrated in table (5.2) .

The VMT is constructed by the compiler. Every instance of an object receives a pointer to its VMT.

Chapter 6

Generated code

The Free Pascal compiler relies on the assembler to make object files. It generates just the assembly language file. In the following two sections, we discuss what is generated when you compile a unit or a program.

6.1 Units

When you compile a unit, the Free Pascal compiler generates 2 files :

1. A unit description file (with extension .ppu, or .ppw on WINDOWSNT).
2. An assembly language file (with extension .s).

The assembly language file contains the actual source code for the statements in your unit, and the necessary memory allocations for any variables you use in your unit. This file is converted by the assembler to an object file (with extension .o) which can then be linked to other units and your program, to form an executable.

By default (compiler version 0.9.4 and up), the assembly file is removed after it has been compiled. Only in the case of the -s command-line option, the assembly file must be left on disk, so the assembler can be called later. You can disable the erasing of the assembler file with the -a switch.

The unit file contains all the information the compiler needs to use the unit:

1. Other used units, both in interface and implementation.
2. Types and variables from the interface section of the unit.
3. Function declarations from the interface section of the unit.
4. Some debugging information, when compiled with debugging.
5. A date and time stamp.

Macros, symbols and compiler directives are *not* saved to the unit description file. Aliases for functions are also not written to this file, which is logical, since they cannot appear in the interface section of a unit.

The detailed contents and structure of this file are described in the first appendix. You can examine a unit description file using the dumpppu program, which shows the contents of the file.

If you want to distribute a unit without source code, you must provide both the unit description file and the object file.

You can also provide a C header file to go with the object file. In that case, your unit can be used by someone who wishes to write his programs in C. However, you must make this header file yourself since the Free Pascal compiler doesn't make one for you.

6.2 Programs

When you compile a program, the compiler produces again 2 files :

1. An assembly language file containing the statements of your program, and memory allocations for all used variables.
2. A linker response file. This file contains a list of object files the linker must link together.

The link response file is, by default, removed from the disk. Only when you specify the `-s` command-line option or when linking fails, then the ile is left on the disk. It is named link.res.

The assembly language file is converted to an object file by the assembler, and then linked together with the rest of the units and a program header, to form your final program.

The program header file is a small assembly program which provides the entry point for the program. This is where the execution of your program starts, so it depends on the operating system, because operating systems pass parameters to executables in wildly different ways.

It's name is prt0.o, and the source file resides in prt0.s or some variant of this name. It usually resided where the system unit source for your system resides. It's main function is to save the environment and command-line arguments, set up the stack. Then it calls the main program.

Chapter 7

Intel MMX support

7.1 What is it about ?

Free Pascal supports the new MMX (Multi-Media extensions) instructions of Intel processors. The idea of MMX is to process multiple data with one instruction, for example the processor can add simultaneously 4 words. To implement this efficiently, the Pascal language needs to be extended. So Free Pascal allows to add for example two `array[0..3] of word`, if MMX support is switched on. The operation is done by the `MMX` unit and allows people without assembler knowledge to take advantage of the MMX extensions.

Here is an example:

```
uses
   MMX;   { include some predefined data types }

const
   { tmmxword = array[0..3] of word;, declared by unit MMX }
   w1 : tmmxword = (111,123,432,4356);
   w2 : tmmxword = (4213,63456,756,4);

var
   w3 : tmmxword;
   l : longint;

begin
   if is_mmx_cpu then  { is_mmx_cpu is exported from unit mmx }
     begin
{$mmx+}   { turn mmx on }
        w3:=w1+w2;
{$mmx-}
     end
   else
     begin
        for i:=0 to 3 do
          w3[i]:=w1[i]+w2[i];
     end;
end.
```

7.2 Saturation support

One important point of MMX is the support of saturated operations. If a operation would cause an overflow, the value stays at the highest or lowest possible value for the data type: If you use byte values you get normally 250+12=6. This is very annoying when doing color manipulations or changing audio samples, when you have to do a word add and check if the value is greater than 255. The solution is saturation: 250+12 gives 255. Saturated operations are supported by the `MMX` unit. If you want to use them, you have simple turn the switch saturation on: `$saturation+`

Here is an example:

```
Program SaturationDemo;
{
  example for saturation, scales data (for example audio)
  with 1.5 with rounding to negative infinity
}

var
   audio1 : tmmxword;

const
   helpdata1 : tmmxword = ($c000,$c000,$c000,$c000);
   helpdata2 : tmmxword = ($8000,$8000,$8000,$8000);

begin
   { audio1 contains four 16 bit audio samples }
{$mmx+}
   { convert it to $8000 is defined as zero, multiply data with 0.75 }
   audio1:=tmmxfixed16(audio1+helpdata2)*tmmxfixed(helpdata1);
{$saturation+}
   { avoid overflows (all values>$7fff becomes $ffff) }
   audio1:=(audio1+helpdata2)-helpdata2;
{$saturation-}
   { now mupltily with 2 and change to integer }
   audio1:=(audio1 shl 1)-helpdata2;
{$mmx-}
end.
```

7.3 Restrictions of MMX support

In the beginning of 1997 the MMX instructions were introduced in the Pentium processors, so multitasking systems wouldn't save the newly introduced MMX registers. To work around that problem, Intel mapped the MMX registers to the FPU register.

The consequence is that you can't mix MMX and floating point operations. After using MMX operations and before using floating point operations, you have to call the routine `EMMS` of the `MMX` unit. This routine restores the FPU registers.

careful: The compiler doesn't warn if you mix floating point and MMX operations, so be careful.

The MMX instructions are optimized for multi media (what else?). So it isn't possible to perform each operation, some opertions give a type mismatch, see section 7.4 for the supported MMX operations

An important restriction is that MMX operations aren't range or overflow checked, even when you turn range and overflow checking on. This is due to the nature of MMX operations.

The `MMX` unit must be always used when doing MMX operations because the exit code of this unit clears the MMX unit. If it wouldn't do that, other program will crash. A consequence of this is that you can't use MMX operations in the exit code of your units or programs, since they would interfere with the exit code of the `MMX` unit. The compiler can't check this, so you are responsible for this !

7.4 Supported MMX operations

7.5 Optimizing MMX support

Here are some helpful hints to get optimal performance:

- The `EMMS` call takes a lot of time, so try to seperate floating point and MMX operations.
- Use MMX only in low level routines because the compiler saves all used MMX registers when calling a subroutine.
- The NOT-operator isn't supported natively by MMX, so the compiler has to generate a workaround and this operation is inefficient.
- Simple assignements of floating point numbers don't access floating point registers, so you need no call to the `EMMS` procedure. Only when doing arithmetic, you need to call the `EMMS` procedure.

Chapter 8

Memory issues

8.1 The 32-bit model.

The Free Pascal Pascal compiler issues 32-bit code. This has several consequences:

- You need a 386 processor to run the generated code. The compiler functions on a 286 when you compile it using Turbo Pascal, but the generated programs cannot be assembled or executed.
- You don't need to bother with segment selectors. Memory can be addressed using a single 32-bit pointer. The amount of memory is limited only by the available amount of (virtual) memory on your machine.
- The structures you define are unlimited in size. Arrays can be as long as you want. You can request memory blocks from any size.

The fact that 32-bit code is used, means that some of the older Turbo Pascal constructs and functions are obsolete. The following is a list of functions which shouldn't be used anymore:

Seg() : Returned the segment of a memory address. Since segments have no more meaning, zero is returned in the Free Pascal run-time library implementation of `Seg`.

Ofs() : Returned the offset of a memory address. Since segments have no more meaning, the complete address is returned in the Free Pascal implementation of this function. This has as a consequence that the return type is `Longint` instead of `Word`.

Cseg(), Dseg() : Returned, respectively, the code and data segments of your program. This returns zero in the Free Pascal implementation of the system unit, since both code and data are in the same memory space.

Ptr accepted a segment and offset from an address, and would return a pointer to this address. This has been changed in the run-time library. Standard it returns now simply the offset. If you want to retain the old functionality, you can recompile the run-time library with the `DoMapping` symbol defined. This will restore the Turbo Pascal behaviour.

memw and mem these arrays gave access to the DOS memory. Free Pascal supports them, they are mapped into DOS memory space. You need the `GO32` unit for this.

Table 8.1: Stack frame when calling a procedure

Offset	What is stored	Optional ?
+x	parameters	Yes
+12	function result	Yes
+8	self	Yes
+4	Frame pointer of parent procedure	Yes
+0	Return address	No

You shouldn't use these functions, since they are very non-portable, they're specific to DOS and the ix86 processor. The Free Pascal compiler is designed to be portable to other platforms, so you should keep your code as portable as possible, and not system specific. That is, unless you're writing some driver units, of course.

8.2 The stack

The stack is used to pass parameters to procedures or functions, to store local variables, and, in some cases, to return function results.

When a function or procedure is called, then the following is done by the compiler :

1. If there are any parameters to be passed to the procedure, they are pushed from right to left on the stack.
2. If a function is called that returns a variable of type `String`, `Set`, `Record`, `Object` or `Array`, then an address to store the function result in, is pushed on the stack.
3. If the called procedure or function is an object method, then the pointer to `self` is pushed on the stack.
4. If the procedure or function is nested in another function or procedure, then the frame pointer of the parent procedure is pushed on the stack.
5. The return address is pushed on the stack (This is done automatically by the instruction which calls the subroutine).

The resulting stack frame upon entering looks as in table (8.1) .

Intel x86 version

The stack is cleared with the `ret` I386 instruction, meaning that the size of all pushed parameters is limited to 64K.

DOS

Under the DOS targets , the default stack is set to 256Kb. This value cannot be modified for the GO32V1 target. But this can be modified with the GO32V2 target using a special DJGPP utility `stubedit`. It is to note that the stack size may be changed with some compiler switches, this stack size, if *greater* then the default stack size will be used instead, otherwise the default stack size is used.

Linux

Under Linux, stack size is only limited by the available memory by the system.

OS/2

Under OS/2, stack size is determined by one of the runtime environment variables set for EMX. Therefore, the stack size is user defined.

Motorola 680x0 version

All depending on the processor target, the stack can be cleared in two manners, if the target processor is a MC68020 or higher, the stack will be cleared with a simple `rtd` instruction, meaning that the size of all pushed parameters is limited to 32K.

Otherwise on MC68000/68010 processors, the stack clearing mechanism is sligthly more complicated, the exit code will look like this:

```
{
  move.l  (sp)+,a0
  add.l   paramsize,a0
  move.l  a0,-(sp)
  rts
}
```

Amiga

Under AmigaOS, stack size is determined by the user, which sets this value using the stack program. Typical sizes range from 4K to 40K.

Atari

Under Atari TOS, stack size is currently limited to 8K, and it cannot be modified. This may change in a future release of the compiler.

8.3 The heap

The heap is used to store all dynamic variables, and to store class instances. The interface to the heap is the same as in Turbo Pascal, although the effects are maybe not the same. On top of that, the Free Pascal run-time library has some extra possibilities, not available in Turbo Pascal. These extra possibilities are explained in the next subsections.

The heap grows

Free Pascal supports the `HeapEerror` procedural variable. If this variable is non-nil, then it is called in case you try to allocate memory, and the heap is full. By default, `HeapError` points to the `GrowHeap` function, which tries to increase the heap.

The growheap function issues a system call to try to increase the size of the memory available to your program. It first tries to increase memory in a 1 Mb. chunk. If this fails, it tries to increase the heap by the amount you requested from the heap.

If the call to `GrowHeap` has failed, then a run-time error is generated, or nil is returned, depending on the `GrowHeap` result.

If the call to `GrowHeap` was successful, then the needed memory will be allocated.

Using Blocks

If you need to allocate a lot of small block for a small period, then you may want to recompile the run-time library with the `USEBLOCKS` symbol defined. If it is recompiled, then the heap management is done in a different way.

The run-time library keeps a linked list of allocated blocks with size up to 256 bytes[1]. By default, it keeps 32 of these lists[2].

When a piece of memory in a block is deallocated, the heap manager doesn't really deallocate the occupied memory. The block is simply put in the linked list corresponding to its size.

When you then again request a block of memory, the manager checks in the list if there is a non-allocated block which fits the size you need (rounded to 8 bytes). If so, the block is used to allocate the memory you requested.

This method of allocating works faster if the heap is very fragmented, and you allocate a lot of small memory chunks.

Since it is invisible to the program, this provides an easy way of improving the performance of the heap manager.

Using the split heap

Remark : The split heap is still somewhat buggy. Use at your own risk for the moment.

The split heap can be used to quickly release a lot of blocks you alloated previously.

Suppose that in a part of your program, you allocate a lot of memory chunks on the heap. Suppose that you know that you'll release all this memory when this particular part of you program is finished.

In Turbo Pascal, you could foresee this, and mark the position of the heap (using the `Mark` function) when entering this particular part of your program, and release the occupied memory in one call with the `Release` call.

For most purposes, this works very good. But sometimes, you may need to allocate something on the heap that you *don't* want deallocated when you release the allocated memory. That is where the split heap comes in.

When you split the heap, the heap manager keeps 2 heaps: the base heap (the normal heap), and the temporary heap. After the call to split the heap, memory is allocated from the temporary heap. When you're finished using all this memory, you unsplit the heap. This clears all the memory on the split heap with one call. After that, memory will be allocated from the base heap again.

So far, nothing special, nothing that can't be done with calls to `mark` and `release`. Suppose now that you have split the heap, and that you've come to a point where you need to allocate memory that is to stay allocated after you unsplit the heap again. At this point, mark and release are of no use. But when using the split heap, you can tell the heap manager to –temporarily– use the base heap again to allocate

[1] The size can be set using the `max_size` constant in the `heap.inc` source file.
[2] The actual size is `max_size div 8`.

memory. When you've allocated the needed memory, you can tell the heap manager that it should start using the temporary heap again. When you're finished using the temporary heap, you release it, and the memory you allocated on the base heap will still be allocated.

To use the split-heap, you must recompile the run-time library with the `TempHeap` symbol defined. This means that the following functions are available :

```
procedure Split_Heap;
procedure Switch_To_Base_Heap;
procedure Switch_To_Temp_Heap;
procedure Switch_Heap;
procedure ReleaseTempHeap;
procedure GetempMem(var p : pointer;size : longint);
```

`split_heap` is used to split the heap. It cannot be called two times in a row, without a call to `releasetempheap`. `Releasetempheap` completely releases the memory used by the temporary heap. Switching temporarily back to the base heap can be done using the `switch_to_base_heap` call, and returning to the temporary heap is done using the `switch_to_temp_heap` call. Switching from one to the other without knowing on which one your are right now, can be done using the `switch_heap` call, which will split the heap first if needed.

A call to `GetTempMem` will allocate a memory block on the temporary heap, whatever the current heap is. The current heap after this call will be the temporary heap.

Typically, what will appear in your code is the following sequence :

```
Split_Heap
...
{ Memory allocation }
...
{ !! non-volatile memory needed !!}
Switch_To_Base_Heap;
getmem (P,size);
Switch_To_Temp_Heap;
...
{Memory allocation}
...
ReleaseTempHeap;
{All allocated memory is now freed, except for the memory pointed to by 'P' }
...
```

8.4 using dos memory under the Go32 extender

Because Free Pascal is a 32 bit compiler, and uses a DOS extender, accessing DOS memory isn't trivial. What follows is an attempt to an explanation of how to access and use DOS or real mode memory[3].

In *Proteced Mode*, memory is accessed through *Selectors* and *Offsets*. You can think of Selectors as the protected mode equivalents of segments.

In Free Pascal, a pointer is an offset into the `DS` selector, which points to the Data of your program.

[3]Thanks to an explanation of Thomas schatzl (E-mail:`tom_at_work@geocities.com`).

To access the (real mode) DOS memory, somehow you need a selector that points to the DOS memory. The GO32 unit provides you with such a selector: The `DosMemSelector` variable, as it is conveniently called.

You can also allocate memory in DOS's memory space, using the `global_dos_alloc` function of the GO32 unit. This function will allocate memory in a place where DOS sees it.

As an example, here is a function that returns memory in real mode DOS and returns a selector:offset pair for it.

```
procedure dosalloc(var selector : word;
                   var segment : word;
                   size : longint);

var result : longint;

begin
     result := global_dos_alloc(size);
     selector := word(result);
     segment := word(result shr 16);
end;
```

(you need to free this memory using the `global_dos_free` function.)

You can access any place in memory using a selector. You can get a selector using the `allocate_ldt_descriptor` function, and then let this selector point to the physical memory you want using the `set_segment_base_address` function, and set its length using `set_segment_limit` function. You can manipulate the memory pointed to by the selector using the functions of the GO32 unit. For instance with the `seg_fillchar` function. After using the selector, you must free it again using the `free_ldt_selector` function.

More information on all this can be found in the Unit reference, the chapter on the GO32 unit.

Chapter 9

Optimizations

9.1 Non processor specific

The following sections describe the general optimizations done by the compiler, they are non processor specific. Some of these require some compiler switch override while others are done automatically (those which require a switch will be noted as such).

Constant folding

In Free Pascal, if the operand(s) of an operator are constants, they will be evaluated at compile time.

Example

```
   x:=1+2+3+6+5;
will generate the same code as
   x:=17;
```

Furthermore, if an array index is a constant, the offset will be evaluated at compile time. This means that accessing MyData[5] is as efficient as accessing a normal variable.

Finally, calling `Chr`, `Hi`, `Lo`, `Ord`, `Pred`, or `Succ` functions with constant parameters generates no run-time library calls, instead, the values are evaluated at compile time.

Constant merging

Using the same constant string two or more times generates only one copy of the string constant.

Short cut evaluation

Evaluation of boolean expression stops as soon as the result is known, which makes code execute faster then if all boolean operands were evaluated.

Constant set inlining

Using the `in` operator is always more efficient then using the equivalent `<>`, `=`, `<=`, `>=`, `<` and `>` operators. This is because range comparisons can be done more easily with `in` then with normal comparison operators.

Small sets

Sets which contain less then 33 elements can be directly encoded using a 32-bit value, therefore no run-time library calls to evaluate operands on these sets are required; they are directly encoded by the code generator.

Range checking

Assignments of constants to variables are range checked at compile time, which removes the need the generation of runtime range checking code.

Remark: This feature was not implemented before version 0.99.5 of Free Pascal.

Shifts instead of multiply or divide

When one of the operands in a multiplication is a power of two, they are encoded using arithmetic shifts instructions, which generates more efficient code.

Similarly, if the divisor in a `div` operation is a power of two, it is encoded using arithmetic shifts instructions.

The same is true when accessing array indexes which are powers of two, the address is calculated using arithmetic shifts instead of the multiply instruction.

Automatic alignment

By default all variables larger then a byte are guaranteed to be aligned at least on a word boundary.

Furthermore all pointers allocated using the standard runtime library (`New` and `GetMem` among others) are guaranteed to return pointers aligned on a quadword boundary (64-bit alignment).

Alignment of variables on the stack depends on the target processor.

Remark: Quadword alignment of pointers is not guaranteed on systems which don't use an internal heap, such as for the Win32 target.

Remark: Alignment is also done *between* fields in records, objects and classes, this is *not* the same as in Turbo Pascal and may cause problems when using disk I/O with these types. To get no alignment between fields use the `packed` directive or the {`$PackRecords n`} switch. For further information, take a look at the reference manual under the `record` heading.

Smart linking

This feature removes all unreferenced code in the final executable file, making the executable file much smaller.

Smart linking is switched on with the `-Cx` command-line switch, or using the {`$SMARTLINK ON`} global directive.

Remark: Smart linking was implemented starting with version 0.99.6 of Free Pascal.

Inline routines

The following runtime library routines are coded directly into the final executable : `Lo`, `Hi`, `High`, `Sizeof`, `TypeOf`, `Length`, `Pred`, `Succ`, `Inc`, `Dec` and `Assigned`.

Remark: Inline `Inc` and `Dec` were not completely implemented until version 0.99.6 of Free Pascal.

Case optimization

When using the `-O1` (or higher) switch, case statements will be generated using a jump table if appropriate, to make them execute faster.

Stack frame omission

Under specific conditions, the stack frame (entry and exit code for the routine, see section 3.3) will be omitted, and the variable will directly be accessed via the stack pointer.

Conditions for omission of the stack frame :

- The function has no parameters nor local variables.
- Routine does not call other routines.
- Routine does not contain assembler statements. However, a `assembler` routine may omit it's stack frame.
- Routine is not declared using the `Interrupt` directive.
- Routine is not a constructor or destructor.

Register variables

When using the `-Or` switch, local variables or parameters which are used very often will be moved to registers for faster access.

Remark: Register variable allocation is currently an experimental feature, and should be used with caution.

Intel x86 specific

Here follows a listing of the opimizing techniques used in the compiler:

1. When optimizing for a specific Processor (`-Op1, -Op2, -Op3`, the following is done:
   - In `case` statements, a check is done whether a jump table or a sequence of conditional jumps should be used for optimal performance.

- Determines a number of strategies when doing peephole optimization, e.g.: `movzbl (%ebp), %eax` will be changed into `xorl %eax,%eax; movb (%ebp),%al` for Pentium and PentiumMMX.

2. When optimizing for speed (`-OG`, the default) or size (`-Og`), a choice is made between using shorter instructions (for size) such as `enter $4`, or longer instructions `subl $4,%esp` for speed. When smaller size is requested, things aren't aligned on 4-byte boundaries. When speed is requested, things are aligned on 4-byte boundaries as much as possible.

3. Fast optimizations (`-O1`): activate the peephole optimizer

4. Slower optimizations (`-O2`): also activate the common subexpression elimination (formerly called the "reloading optimizer")

5. Uncertain optimizations (`-Ou`): With this switch, the common subexpression elimination algorithm can be forced into making uncertain optimizations.

   Although you can enable uncertain optimizations in most cases, for people who do not understand the following technical explanation, it might be the safest to leave them off.

   > *If uncertain optimizations are enabled, the CSE algortihm assumes that*
   >
   > - *If something is written to a local/global register or a procedure/function parameter, this value doesn't overwrite the value to which a pointer points.*
   > - *If something is written to memory pointed to by a pointer variable, this value doesn't overwrite the value of a local/global variable or a procedure/function parameter.*

   The practical upshot of this is that you cannot use the uncertain optimizations if you both write and read local or global variables directly and through pointers (this includes `Var` parameters, as those are pointers too).

   The following example will produce bad code when you switch on uncertain optimizations:

```
Var temp: Longint;

Procedure Foo(Var Bar: Longint);
Begin
  If (Bar = temp)
    Then
      Begin
        Inc(Bar);
        If (Bar <> temp) then Writeln('bug!')
      End
End;

Begin
  Foo(Temp);
End.
```

   The reason it produces bad code is because you access the global variable `Temp` both through its name `Temp` and through a pointer, in this case using the `Bar` variable parameter, which is nothing but a pointer to `Temp` in the above code.

On the other hand, you can use the uncertain optimizations if you access global/local variables or parameters through pointers, and *only* access them through this pointer[1].

For example:

```
Type TMyRec = Record
                a, b: Longint;
              End;
     PMyRec = ^TMyRec;


     TMyRecArray = Array [1..100000] of TMyRec;
     PMyRecArray = ^TMyRecArray;

Var MyRecArrayPtr: PMyRecArray;
    MyRecPtr: PMyRec;
    Counter: Longint;

Begin
  New(MyRecArrayPtr);
  For Counter := 1 to 100000 Do
    Begin
       MyRecPtr := @MyRecArrayPtr^[Counter];
       MyRecPtr^.a := Counter;
       MyRecPtr^.b := Counter div 2;
    End;
End.
```

Will produce correct code, because the global variable `MyRecArrayPtr` is not accessed directly, but only through a pointer (`MyRecPtr` in this case).

In conclusion, one could say that you can use uncertain optimizations *only* when you know what you're doing.

Motorola 680x0 specific

Using the `-O2` switch does several optimizations in the code produced, the most notable being:

- Sign extension from byte to long will use `EXTB`
- Returning of functions will use `RTD`
- Range checking will generate no run-time calls
- Multiplication will use the long `MULS` instruction, no runtime library call will be generated
- Division will use the long `DIVS` instruction, no runtime library call will be generated

[1] You can use multiple pointers to point to the same variable as well, that doesn't matter.

9.2 Optimization switches

This is where the various optimizing switches and their actions are described, grouped per switch.

-On: with n = 1..3: these switches activate the optimizer. A higher level automatically includes all lower levels.

- Level 1 (`-O1`) activates the peephole optimizer (common instruction sequences are replaced by faster equivalents).
- Level 2 (`-O2`) enables the assembler data flow analyzer, which allows the common subexpression elimination procedure to remove unnecessary reloads of registers with values they already contain.
- Level 3 (`-O3`) enables uncertain optimizations. For more info, see -Ou.

-OG: This causes the code generator (and optimizer, IF activated), to favor faster, but code-wise larger, instruction sequences (such as "`subl $4,%esp`") instead of slower, smaller instructions ("`enter $4`"). This is the default setting.

-Og: This one is exactly the reverse of -OG, and as such these switches are mutually exclusive: enabling one will disable the other.

-Or: this setting (once it's fixed) causes the code generator to check which variables are used most, so it can keep those in a register.

-Opn: with n = 1..3: setting the target processor does NOT activate the optimizer. It merely influences the code generator and, if activated, the optimizer:

- During the code generation process, this setting is used to decide whether a jump table or a sequence of successive jumps provides the best performance in a case statement.
- The peephole optimizer takes a number of decisions based on this setting, for example it translates certain complex instructions, such as

  ```
  movzbl (mem), %eax|
  ```

  to a combination of simpler instructions

  ```
  xorl %eax, %eax
  movb (mem), %al
  ```

  for the Pentium.

-Ou: This enables uncertain optimizations. You cannot use these always, however. The previous section explains when they can be used, and when they cannot be used.

9.3 Tips to get faster code

Here some general tips for getting better code are presented. They mainly concern coding style.

- Find a better algorithm. No matter how much you and the compiler tweak the code, a quicksort will (almost) always outperform a bubble sort, for example.

- Use variables of the native size of the processor you're writing for. For the 80x86 and compatibles, this is 32 bit, so you're best of using longint and cardinal variables.
- Turn on the optimizer.
- Write your if/then/else statements so that the code in the "then"-part gets executed most of the time (improves the rate of successful jump prediction).
- If you are allocating and disposing a lot of small memory blocks, check out the heapblocks variable (heapblocks are on by default from release 0.99.8 and later)
- Profile your code (see the -pg switch) to find out where the bottlenecks are. If you want, you can rewrite those parts in assembler. You can take the code generated by the compiler as a starting point. When given the `-a` command-line switch, the compiler will not erase the assembler file at the end of the assembly process, so you can study the assembler file.

  Note: Code blocks which contain an assembler block, are not processed at all by the optimizer at this time. Update: as of versino 0.99.11, the Pascal code surrounding the assembler blocks is optimized.

9.4 Floating point

This is where can be found processor specific information on Floating point code generated by the compiler.

Intel x86 specific

All normal floating point types map to their real type, including `comp` and `extended`.

Motorola 680x0 specific

Early generations of the Motorola 680x0 processors did not have integrated floating point units, so to circumvent this fact, all floating point operations are emulated (when the `$E+` switch ,which is the default) using the IEEE `Single` floating point type. In other words when emulation is on, Real, Single, Double and Extended all map to the `single` floating point type.

When the `$E` switch is turned off, normal 68882/68881/68040 floating point opcodes are emitted. The Real type still maps to `Single` but the other types map to their true floating point types. Only basic FPU opcodes are used, which means that it can work on 68040 processors correctly.

Remark: `Double` and `Extended` types in true floating point mode have not been extensively tested as of version 0.99.5.

Remark: The `comp` data type is currently not supported.

Appendix A

Anatomy of a unit file

A.1 Basics

The best and most updated documentation about the ppu files can be found in ppu.pas and ppudump.pp which can be found in rtl/utils/.

To read or write the ppufile, you can use the ppu unit ppu.pas which has an object called tppufile which holds all routines that deal with ppufile handling. Describing the layout of a ppufile, the methods which can be used for it are described.

A unit file consists of basically five or six parts:

1. A unit header.
2. A file interface part.
3. A definition part. Contains all type and procedure definitions.
4. A symbol part. Contains all symbol names and references to their definitions.
5. A browser part. Contains all references from this unit to other units and inside this unit. Only available when the `uf_has_browser` flag is set in the unit flags
6. A file implementation part (currently unused). implementation part.

A.2 reading ppufiles

We will first create an object ppufile which will be used below. We are opening unit test.ppu as an example.

```
var
  ppufile : pppufile;
begin
{ Initialize object }
  ppufile:=new(pppufile,init('test.ppu');
{ open the unit and read the header, returns false when it fails }
  if not ppufile.open then
    error('error opening unit test.ppu');

{ here we can read the unit }
```

```
{ close unit }
  ppufile.close;
{ release object }
  dispose(ppufile,done);
end;
```

Note: When a function fails (for example not enough bytes left in an entry) it sets the `ppufile.error` variable.

A.3 The Header

The header consists of a record containing 24 bytes:

```
tppuheader=packed record
    id       : array[1..3] of char; { = 'PPU' }
    ver      : array[1..3] of char;
    compiler : word;
    cpu      : word;
    target   : word;
    flags    : longint;
    size     : longint; { size of the ppufile without header }
    checksum : longint; { checksum for this ppufile }
  end;
```

The header is already read by the `ppufile.open` command. You can access all fields using `ppufile.header` which holds the current header record.

field	description
`id`	this is allways 'PPU', can be checked with `function ppufile.CheckPPUId:boolean;`
`ver`	ppu version, currently '015', can be checked with `function ppufile.GetPPUVersion:longint;` (returns 15)
`compiler`	compiler version used to create the unit. Doesn't contain the patchlevel. Currently 0.99 where 0 is the high byte and 99 the low byte
`cpu`	cpu for which this unit is created. 0 = i386 1 = m68k
`target`	target for which this unit is created, this depends also on the cpu! For i386: 0 Go32v1 1 Go32V2 2 Linux-i386 3 OS/2 4 Win32 For m68k: 0 Amiga 1 Mac68k 2 Atari 3 Linux-m68k
`flag`	the unit flags, contains a combination of the uf_ constants which are definied in ppu.pas
`size`	size of this unit without this header
`checksum`	checksum of the interface parts of this unit, which determine if a unit is changed or not, so other units can see if they need to be recompiled

A.4 The sections

After this header follow the sections. All sections work the same! A section contains of entries and is ended with also an entry, but containing the specific ibend constant (see ppu.pas for a list).

Each entry starts with an entryheader.

```
tppuentry=packed record
  id   : byte;
  nr   : byte;
  size : longint;
end;
```

field	Description
id	this is 1 or 2 and can be check if it the entry is correctly found. 1 means its a main entry, which says that it is part of the basic layout as explained before. 2 toggles that it it a sub entry of a record or object
nr	contains the ib constant number which determines what kind of entry it is
size	size of this entry without the header, can be used to skip entries very easily.

To read an entry you can simply call `ppufile.readentry:byte`, it returns the `tppuentry.nr` field, which holds the type of the entry. A common way how this works is (example is for the symbols):

```
repeat
  b:=ppufile.readentry;
  case b of
 ib<etc> : begin
           end;
ibendsyms : break;
  end;
until false;
```

Then you can parse each entry type yourself. `ppufile.readentry` will take care of skipping unread bytes in the entry an read the next entry correctly! A special function is `skipuntilentry(untilb:byte):boolean;` which will read the ppufile until it finds entry `untilb` in the main entries.

Parsing an entry can be done with `ppufile.getxxx` functions. The available functions are:

```
procedure ppufile.getdata(var b;len:longint);
function  getbyte:byte;
function  getword:word;
function  getlongint:longint;
function  getreal:ppureal;
function  getstring:string;
```

To check if you're at the end of an entry you can use the following function:

```
function  EndOfEntry:boolean;
```

notes:

1. `ppureal` is the best real that exists for the cpu where the unit is created for. Currently it is `extended` for i386 and `single` for m68k.
2. the `ibobjectdef` and `ibrecorddef` have stored a definition and symbol section for themselves. So you'll need a recursive call. See ppudump.pp for a correct implementation.

A complete list of entries and what their fields contain can be found in ppudump.pp.

A.5 Creating ppufiles

Creating a new ppufile works almost the same as writing. First you need to init the object and call create:

```
ppufile:=new(pppufile,'output.ppu');
ppufile.create;
```

After that you can simply write all needed entries. You'll have to take care that you write at least the basic entries for the sections:

```
ibendinterface
ibenddefs
ibendsyms
ibendbrowser (only when you've set uf_has_browser!)
ibendimplementation
ibend
```

Writing an entry is a little different than reading it. You need to first put everything in the entry with ppufile.putxxx:

```
procedure putdata(var b;len:longint);
procedure putbyte(b:byte);
procedure putword(w:word);
procedure putlongint(l:longint);
procedure putreal(d:ppureal);
procedure putstring(s:string);
```

After putting all the things in the entry you need to call `ppufile.writeentry(ibnr:byte)` where `ibnr` is the entry number you're writing.

At the end of the file you need to call `ppufile.writeheader` to write the new header to the file. This takes automatically care of the new size of the ppufile. When that is also done you can call `ppufile.close` and dispose the object.

Extra functions/variables available for writing are:

```
ppufile.NewHeader;
ppufile.NewEntry;
```

This will give you a clean header or entry. Normally called automatically in `ppufile.writeentry`, so you can't forget it.

```
ppufile.flush;
```

to flush the current buffers to the disk

```
ppufile.do_crc:boolean;
```

set to false if you don't want that the crc is updated, this is necessary if you write for example the browser data.

Appendix B

Compiler and RTL source tree structure

B.1 The compiler source tree

All compiler source files are in one directory, normally in source/compiler. For more informations about the structure of the compiler have a look at the Compiler Manual which contains also some informations about compiler internals.

Appendix C

Compiler limits

Although many of the restrictions imposed by the MS-DOS system are removed by use of an extender, or use of another operating system, there still are some limitations to the compiler:

1. Procedure or Function definitions can be nested to a level of 32.
2. Maximally 255 units can be used in a program when using the real-mode compiler (i.e. a binary that was compiled by Borland Pascal). When using the 32-bit compiler, the limit is set to 1024. You can change this by redefining the `maxunits` constant in the files.pas compiler source file.

Appendix D

Compiler modes

Here we list the exact effect of the different compiler modes. They can be set with the `$Mode` switch, or by command line switches.

D.1 FPC mode

This mode is selected by the `$MODE FPC` switch. On the command-line, this means that you use none of the other compatibility mode switches. It is the default mode of the compiler. This means essentially:

1. You must use the address operator to assign procedural variables.
2. A forward declaration must be repeated exactly the same by the implementation of a function/procedure. In particular, you can not omit the parameters when implementing the function or procedure.
3. Overloading of functions is allowed.
4. Nested comments are allowed.
5. The Objpas unit is NOT loaded.
6. You can use the cvar type.
7. PChars are converted to strings automatically.

D.2 TP mode

This mode is selected by the `$MODE TP` switch. On the command-line, this mode is selected by the `-So` switch.

1. You cannot use the address operator to assign procedural variables.
2. A forward declaration must not be repeated exactly the same by the implementation of a function/procedure. In particular, you can omit the parameters when implementing the function or procedure.
3. Overloading of functions is not allowed.
4. Nested comments are not allowed.
5. You can not use the cvar type.

D.3 Delphi mode

This mode is selected by the `$MODE DELPHI` switch. On the command-line, this mode is selected by the `-Sd` switch.

1. You can not use the address operator to assign procedural variables.
2. A forward declaration must not be repeated exactly the same by the implementation of a function/procedure. In particular, you can not omit the parameters when implementing the function or procedure.
3. Overloading of functions is not allowed.
4. Nested comments are not allowed.
5. The Objpas unit is loaded right after the system unit. One of the consequences of this is that the type `Integer` is redefined as `Longint`.

D.4 GPC mode

This mode is selected by the `$MODE GPC` switch. On the command-line, this mode is selected by the `-Sp` switch.

1. You cmust use the address operator to assign procedural variables.
2. A forward declaration must not be repeated exactly the same by the implementation of a function/procedure. In particular, you can omit the parameters when implementing the function or procedure.
3. Overloading of functions is not allowed.
4. Nested comments are not allowed.
5. You can not use the cvar type.

D.5 OBJFPC mode

This mode is selected by the `$MODE OBJFPC` switch. On the command-line, this mode is selected by the `-S2` switch.

1. You must use the address operator to assign procedural variables.
2. A forward declaration must be repeated exactly the same by the implementation of a function/procedure. In particular, you can not omit the parameters when implementing the function or procedure.
3. Overloading of functions is allowed.
4. Nested comments are allowed.
5. The Objpas unit is loaded right after the system unit. One of the consequences of this is that the type `Integer` is redefined as `Longint`.
6. You can use the cvar type.
7. PChars are converted to strings automatically.