introduction on programming

"hello world"

by Blazko, mail@neveprise.de

Version: 1.0, 2001-03-26

This article is on the very basic elements of a program, including variables, functions, loops, conditions and more.

Table of contents

1.0 introduction

> top of page

1.1 prologue

> top of page

Some people tend to reduce the difference between a gaming console like the Sony PS2, Sega Dreamcast or Nintendo 64 and a Personal Computer on their gaming qualities. But in contrast to a console, a PC is much more than a thing to play with. Furthermore, it offers a lot of possibilities, including word processing, running databases, creating grafix, making music and many many many more. You can even cook coffee with your machine (see the "coffe-howto"...). Thus, the PC is a real multipurpose tool with

few limitations.
>But it would be a pitty if you would just be a user! All that you can do with a computer is nothing compared to developing own stuff, mean software in any sense that fill your demands of quality and functionality. Maybe that is a reason why you are reading this document: to become a software developer. To help you on your (long?) way, this reference shall help to get into programming, not specific to any language (Object Pascal, C++, Perl, Java...) nor operating system (Linux, Windows, BeOS, MacOS...). <p>

1.2 how programs work

> top of page

What is the very basis of it all? Alright, it is the hardware. No matter what you do, you need at last some hardware. That is what you call the computer you are working with along with it's peripherals like printers, scanners etc.

The operating system (following briefly referred to as "OS") is the instance that handles hardware in conjunction with the software and users. When you turn on your machine, the BIOS (basic input/output system) - an integrated circuit on your computer's mainboard with "hard-wired software" that holds all elementary logic - will read the bootloader from the floppy, DVD/CD-ROM or most probably first hard drive and launch it. When you are working with Linux, you most probably know the Linux Loader (LiLo); when working with NT/2k, you know the NT Loader. These small programms read the OS files (kernel, mods, libs...) into memory. When done, it inits the kernel which will take control on the system.

You have to know that on a modern computer there are concurring instances. This problem arises from true multiprocessing machines. On the one hand you have a certain number of hardware that often work on the same devices (e.g. all cards access the PCI bus synchroniously) at - from a user's point of view - the same time. On the other hand you have several applications (apps, deamons/services, drivers...) running simultaneously that all require the same hardware. For example, you have three downloads running while you are writing a text and listen to music from your WinAmp/FreeAmp etc. All these apps need the CPU to handle instructions in their binary code invoked by the user while those three downloads running in background all have access to your networking adapter and hard drive at the same time.
>If you have just one CPU and netadapter, this only seems to work simultaneously. In reality, each app is given exclusive access to the hardware for a very short period of time (about 20msecs). Each app gets a small bite of time until the other app also execs exclusively and so on. This whole process works in a cycle, thus the user has the impression that all works at the same time, cos each circle segment is too short to be noticed. It is like with a movie: each second is build from 25 frames. That are still pictures each shown for 125th of a second thus you have the impression of a moving picture - a simple trick.

But what actually happens when an app is working?

In a simple case, an application consists of a single file located on your hard drive. When launching it, this file - which is no more than a chain of bits and bytes - is loaded into memory. The structure of those bytes now residing in RAM follow a certain rule specific to the OS. An app's code can be seperated into three parts:

- general application information
- execution part
- data part

The general app info part contains basic info on the app (version, a list of methods etc.). The execution part contains several blocks containing "functions", a set of elementary commands specific to the OS and/or hardware. If displaying those command set in a form readable by humans, you would get instructions very similar to a language that is called Assembler.

Those commands are very lowlevel, thus I will not discuss this any further in here. This is left to some other articles in the systech corner of neveprise.net.

1.3 kinds o programs

> top of page

At least, there are two types of applications:

- directly interpreted (always binary)
- indirectly interpreted (mostly plain text code, sometimes precompiled to binaries)

Directly interpreted software - common referred to as binary - is that kind of software that you use usually: your text editor, imaging app, music player etc. These apps consist of optimized and binary files that cannot be reasonably read by humans. If you try to open your "notepad.exe" or "vi" with Wordpad, less, Norton-/Nothern-/Midnight Commander, Emacs or what else, you will see some very noisy characters. You will not make a sense from most of that what you see. But this code can be interpreted by the OS and hardware and is optimized for it. That is the reason why _*you*_ cannot read it but the OS can (although there *are* some people that are sick enough they can do so!). The advantage is that the code is very fast, cos it does not have to be converted to a more direct form.

Indirectly interpreted software - common referred to as scripts - is distributed as source code that does not have to be compiled to a binary form. Special interpreter software is switched between OS and script that understands script contents and interprets it to direct OS instructions. A very prominent example of such kinds of scripts is Perl (Practical Extraction and Reporting Language). When a Perl script is exec'd, the Perl interpreter (a "binary" application) is launched and given the contents of the script. Then the Perl app iterates thru the script and transfers the script source to OS internal instructions.
> The great advantage of such software is that people can read the script directly and can modify it without the need for a dedicated development environment - a simple text editor is enough. The disadvantage is that they are somewhat slower during certains steps, cos the commands have to translated into OS' ones. <p>

1.4 kinds of languages

> top of page

No matter what kind of app, you need it in a form readable by humans. Almost every software is written by men & women using special development tools ranging from simple plaintext editors to highly skilled IDEs (Integrated Development Environments). They write code that defines the app's logic and processes and that they can read and understand. Then, the developers have to compile the source code to binary files that they can deploy or they deploy the source directly (when scripts).

This source can be distinguished in two ways at least:

- programming language used
- style: procedural and object oriented

Why different languages? Well, you are right - each app has to follow the same logical laws, but most languages are specialized to dedicated topics and are purpose-bound. Compare this to humans:
>We all use a language to express and describe our feelings, thoughts, bidds, commands and information. Some people speak (broken) English, some speak Spanish, French, German, Hindu etcetera. Each language has its advantages and disadvantages. And it is the same with software development languages.
>Some are good for maths, some for driver development, some for text manipulation and some for allround purpose...not to forget personal taste...

Here some languages and why they are developed and used:
>
>

language	characteristics
C/C++/C#	All-around language that can be used for almost everything, especially hardware/driver development. Compilers are available for every platform, nevertheless if Windows, UNIX/Linux, MacOS, BeOS...in some sense platform independent (okay...).
Object Pascal (used by Delphi/Kylix...)	Alrounder language that is very simple to learn and very popular thus. On the other hand, it has the same functional capability like C++.
Visual Basic	Also easy to learn language; VBA is specialized for MS Office and automatization. Bad-minded tongues do not call it a programming language :-) well...who knows...
PERL	"Practical Extraction and Report Language" or "Rubbish Lister", both are approved names by its inventor Larry Wall. VERY powerful for string manipulation and VERY fast, often used for web development. By default, it needs a Perl interpreter to be run but it can be compiled into standalone binaries.
Java	If runtime environment exists for an OS, every Java app can be run on any platform without modifications -> true platform independence. That runtime environment is a layer between the application and the operating system and works as a two-way interpreter. Java is strictly object oriented and encapsulates many things in API keeping the developer away from too many tiny things like manual memory (de-) allocation. This can be an advantage but also a large disadvantage. Java is a development of Sun.
JavaScript	To get it right: it has nothing to do with Java. Originally it was developed by Netscape and is an extension to markup languages such as HTML that gives logic and interactivity to web docs. Especially in the Microsoft dialect called JScript is able to access all nodes of a document using the DOM (Document Object Model). As the time of this writing, the DOM should be an official W3C standard. All dialects of JavaScript and JScript are also called ECMAScript.
ABAP/4	ABAP is a language developed by SAP and is used in R/3, a huge business system for all flavours.
COBOL	Bad, really really bad. Don't let us speak from dis...I had to learn it once...

Beside the different languages as such, almost all of'em can be distinguished internally into two different kinds:
>
>

flavour	description
procedural flavour	The procedural style is the older one. Basically, procedural languages work by calling procedures and passing parameters to them. Procedures are global and can be called from anywhere in coding thus. Do not be confused if you do not even understand the word "procedure" - this article will show basic programming using procedural style only thus explaining the term.
object oriented flavour	Though the procedural style is often hard to understand and harder to handle in larger projects, the object-oriented style (OO~) is optimized for the developers and not for computers. >In contrast to the procedural style with plain function calls, the OO style knows objects with properties and methods. It works like manwoman realize their environment: everything you see is an object with properties (size, position, colour etc.) actions (walking, moving, writing, shining etc.) and can react on events (what actio to do when kicking it, talking to it...). This is similar to OO languages where any objects behave that kind.

2.0 elementaries

> top of page

2.1 what you need / before you begin

> top of page

This article is not dedicated to any special language. Although I have chosen to use Pascal-like pseudo coding in the following examples, it should be applicable to any language. Examples in here are very low-level. We will only discuss and create simple console applications that run in a console window (DOS, console window, bash etc.) and do not use a GUI (Graphical User Interface, such as the Windows GUI, Gnome, KDE, FVWM, Enlightenment, Sawfish...).
>Under these circumstances, I recommend to use the following tools to start development: <p>

You should start developing under Linux, cos there you have have some tools for free you can use. Not Limiting to Linux, there are many other free operating systems out there that run the same apps available on Linux, especially GNU tools. Amongst them are the BSDs (FreeBSD, OpenBSD, NetBSD...), BeOS, Solaris etc.
>
>

compiler / tool	notes
gcc	The GNU C-Compiler. You can use a simple text editor to create your codings and let them compile to executable binaries using the gcc. This will work on every Linux system, cos it should always be installed.
PERL	Maybe quite hard for newbies, cos Perl has a very effective but "short'n cryptive" syntax. Perl scripts can also written using a simple editor.
FreePascal / GNU Pascal	FreePascal is a Pascal compiler that knows oldstyle Turbo Pascal standards as well as some Object Pascal (Delphi, Kylix) syntax. As with the gcc, you can create the coding using a plain text editor of your choice. GNU Pascal is the counterpart from the FSF (Free Software Foundation).
Borland Kylix	Borland offers an Open Source edition of their "Delphi for Linux" at no charge when downloading it or for some money (think it are 99$ on CD).

If you have not Linux installed on your system, some of the Linux' devtools are also available for Windows. The problem with most Windows tools is that they must be bought; here a choice of Win tools:
>
>

compiler / tool	notes
Borland C++	Since version 5.5, the compiler itself can be used for free and can be downloaded at Borland's. You can use notepad to start development. The full Borland C++ is available from approx. $99US upwards.
Borland JBuilder Foundation	JBuilder is a development tool for Java. The commercial version costs $99US and up, but a very essentially and not time-ellapsing trial is available at no cost.
Borland Delphi	A very powerful tool using Object Pascal. The standard version is available starting $99US. This tool and language was/is the author's preferred one, cos it is really powerful but good to handle on the other side, besides being documented almost perfectly.
Turbo Pascal / Pascal for Windows	From Borland, too. Cos Turbo Pascal has quite aged, you can get it for very low money. Note that Turbo Pascal is only for DOS!

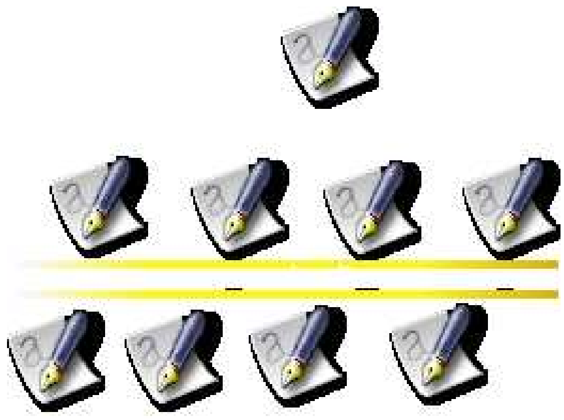
Of course you see that these lists are very subjective. Other languages and tools include Visual Basic, Python, Shell Scripting (Bash, CSH, Windows Scripting Host etc.), Lisp, COBOL etcetera.

2.2 structure of programs

> top of page

Not surprisingly, a project that will later form an application is made of files:

- source code files that contain the app's logic
- header or definition files
- external libraries (from the development tool or third parties)
- resource files containing data like icons and sounds
- ...



Pic 2.2a: structure of a program

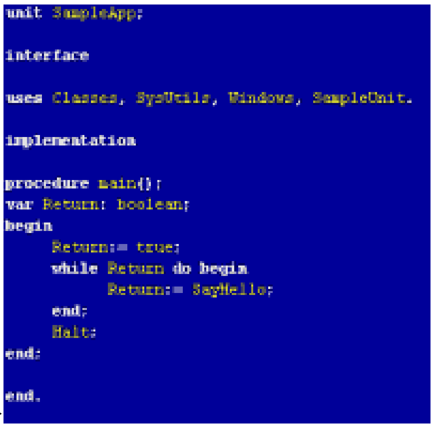
At the top level there is a major code file that is a starting point and holds references to other source files. These contain the logic and interfaces and point to libraries and resource files.

2.3 structure of a unit

> top of page

What is a unit? A unit is a text file that contains definitions and code. Via this code, you control the behaviour and logic of the application that is made of it. In order to keep code clean and well formed, most languages define style guides and block dirty source by demanding a specific file structure. Here an example of how a Pascal source should be structured:

(click image to enlarge)

>app_l.png" target="_blank">
```
unit SampleApp;

interface

uses Classes, SysUtils, Windows, SampleUnit.

implementation

procedure main();
var Return: boolean;
begin
     Return:= true;
     while Return do begin
            Return:= SayHello;
     end;
     Halt;
end;

end.
```

Pic 2.3a: a Object Pascal sample

Explanation:
>
>

keyword / instruction	**meaning**
program or unit	...followed by an name that uniquely identifies the unit within an project.
interface	Marks the beginning of the interface section. That does usually not include any logic coding but declares interfaces such as function names and their parameter lists, constants and structures.
uses	The "uses" clause tells the interpreter and compiler what other units.
implementation	The implementation is is the part where the function and their inner workings reside.

2.4 reserved words

> top of page

When writing your code, you make use of several words. These are the names of variables, objects, procedures, commands, control instructions etc. Normally, you define own names for one of the stuff named, but every programming language knows so called "reserved words". That are words and symbols you cannot use for your own purposes cos they have a very special meaning for the compiler. For example some expressions for math, flow control, block marking etc. are dedicated to the language and cannot be used for other purposes by you.
> Examples for reserved words are: begin, end, unit, program, uses, var, const, void, if, then, else, while, for, repeat, until, do, to, implementation, include, interface, class, record, string, integer, word, bool, byte, type, perform, function, procedure, form, sub, goto, try, finally, except, array, char, result, return, case...
> So, you cannot name a variable "if", cos "if" is already a reserved word beeing used vor

evaluation/condition checking.

2.5 constants and variables

> top of page

If you want to remember anything important for later use or want to pass information to other people, you may do it by writing some notes onto a piece of paper.
> Applications need to do the same in some aspect. They need some place to hold information. In reality, this "memory" is the RAM of your machine, but nowadays you seldomly program the RAM itself (actually, you cannot do it freely, cos modern operating systems protect the RAM from direct access). Development languages offer the technique of variables and constants. <p>

These are containers for special data, just as you use that paper to hold information (except your brain...). Most languages distinguish kinds of data into characters, numbers only, yes/no values etc. That results in a definition of several veriable types, e.g.:

- string: container for shorter alphanumerical values, e.g. "Hello123!?"
- integer: container for whole numbers, e.g. "1235" or "-6542"
- float: container for rational numbers, e.g. "9.0135005"
- boolean: container holding just one bit telling yes or no (true/false; 1/0)
- char: container for just one byte / character, e.g. "A"

When coding, you can declare such a container by specifying a name and a type and then assign a value to it:

```
...
var CptName: string;
...
CptName:= 'James Tiberius Kirk';
```

The keyword "var" introduces a variable declaration. The variable to be used we call "CptName" and is of type string, cos it will hold characters. By using the variable's name later, we can query its content or assign a new value to it. In this case we assign the content "James T. Kirk" to this variable.

A constant is similar to a variable but cannot be changed later on an never has a type:

```
const Version = '20001106';
```

A constant's content is defined immediately and stored directly to memory, thus there is no type definition.

2.6 procedures and functions

> top of page

If you are the boss of a company and want one of your apprentices to go shopping for some beer, you instruct him or her to do so. Basically, you can split it up into two parts:

- the command’s basic meaning or name (what to do att all -> "go shopping")
- and a specification what (what to buy -> "beer")

That is the way programs work: they instruct the computer what to do with what. And as a developer, you have to formulate these instructions that finally makes the application. The tool a developer uses is called procedure or function.
> Actually, there is a difference between a procedure and a function: <p>
>
>

what	meaning
procedure	A procedure is a command that is beeing executed but the result of that operation is not reported to the caller. According to our example, we tell the apprentice to buy some beer for the employees, but we do not care if he/she really does so.
function	A function receives the same information but returns a special result, e.g. if the function succeeded or suffered from an error. That will be this statement from the boss: "Go and get some beer for the stuff, but show me the bottles so that I know that you really got’em!"

Imagine we want to implement a routine that buys some beer, we might code it like that:

```
procedure Buy( What: string);
begin
     ...
end;
```

We use the reserved word "procedure" to tell the compiler that a procedure definition will follow. We give it a name (Buy) followed by a parameter in braces. Parameters are variables; the minimum of information that routine has to know to accomplish its goal. The reserved words "begin" and "end" mark a block of code that contains the actual logic. Thus, when the compiler finds "end;", it knows that the coding for this proc stops here. Instead of "begin" and "end", many languages use these curly brackets: { for begin and } for end.
> Our routine shall buy beer, thus it has to know what to buy. Again, according to our previous example, you have to tell the apprentice to buy<span> some beer. Doing this declaration makes *no* sense:

```
procedure Buy();
begin
     ...
end;
```

You say "buy" --- but what? That is what the parameter shall tell. In contrast to a procedure, a function will look very similar:

```
function Buy( What: string): boolean;
begin
     Result:= false;
     ...
     Result:= true;
end;
```

Except that the word "procedure" has changed to "function", what else has changed? Well, as we see, at the end of the first line there is a "boolean". Looking it up, we know that a boolan expresses a true or false condition (yes / no). By definition, the former procedure can now report the success of the operation. The reserved word "Result" in Pascal will fill our boolean return value with the assignment you specify. Thus, unless changed, our function will report "false" meaning the operation has failed.

2.7 calling functions and procedures

> top of page

Imagine you have created a unit with several procedure und function definitions. You now might use these functions. Let us see how the boss instructs the apprentice to buy some beer:

```
Buy('beer');
```

You see, we use a procedure's name and specify the parameter(s) in brackets. When you start the compiler that will interprete your sources into a binary form, it will search all units that are listet in the "uses" clause. If it finds a procedure that matches name and parameter list, then it will compile that procedure code into the binary.

But how do you check if the procedure has finished successfully? Well, we simply use a function with a boolean result. After the function performed, we can query the result:

```
var Res: boolean;
...
Res:= Buy('beer');
if Res= false then Halt();
```

or much more directly:

```
if Buy('beer')= false then Halt();
```

You see you can query the function's result like a boolean variable. If the function Buy() fails, it returns "false" that will cause our application to exit via the Halt() command. Maybe you did not realize it: en passant you learned how to check cases. But let us discuss dis in the following chapter.

You might ask, what is inside procs or functions? What are the "commands" we spoke of? Nearly all commands you call within a custom procedure is already a proc itself. For example "Halt()" is a predefined procedure declared in the standard units of your compiler's library. Thus, there is somewhere this declaration:

```
procedure Halt();
begin
     asm
        int(19);
     end;
end;
```

The Halt() routine directly calls interrupt 19 using Assembler what will cause the app to exit to the OS layer.

Another - and most probable - way is that a command you use is stored in an external unit but does not call any assembler routine but accesses the according routine of a shared library (DLL, *.so...) from the operating system or standard software that contains binary OS instructions.

2.8 if and case statements

> top of page

Sometimes your app has to make decisions. Just like you: when you eat soup, you will take a spoon; when eating spagetthi, you'll take a fork. Without realizing it, you ask yourself: "if soup, then spoon" or "if spagetthi, then fork". When coding, you will write it similar this way:

```
var Food, Tool: string;
...
if Food = 'soup' then Tool:= 'spoon';
if Food = 'spagetthi' then Tool:= 'fork';
write('When you eat ' + Food);
writeln(' then you will take a ' + Tool + '.');
```

In this example we define two string variables "Food" and "Tool". Somewhere in code, the "Food" variable is filled (e.g. by user input); then it's content is checked with two "if"-statements. Depending on "Food"'s content, the "Tool" variable is filled with the value "Spoon" or "Fork". The "writeln()" procedure - which wants a simple string as parameter and is declared somewhere in the standard units - will write it's parameter's value to a console window. The "writeln()" parameter is composed by direct content using qutations and variables (e.g. "'Hello' + Food").

You see that if-statements are quite simple: via "if" some condition is checked, and if the check results to true (-->yes), the code block after "then" is executed. Let us see a more extended declaration of an if-statement:

```
if Food = 'Soup' then
   write('Do not eat dis!')
else
   write('Good choice!');
```

What does it do?: if the string "Food" will contain "Soup", you will get a bad message. On all other cases (means "Food" is NOT "Soup"), you will get a good message. Thus, the reserved word "else" leads to all other cases you do not catch explicitly.

```
if Number = 1 then
                  SomeProc(WithA);
              else
                  SomeCancelDisProc();
```

```
if Number = 2 then
              SomeOtherProc(WithB);
          else
              SomeCancelDisProc();
if Number = 3 then
              YetAnotherProc(WithC);
          else
              SomeCancelDisProc();
```

You must say that the previous coding is not very elegantly. Therefore, most languages know the case or switch command:

```
case Number of
     1:     SomeProc(WithA);
     2:     SomeOtherProc(WithB);
     3:     YetAnotherProc(WithC);
end else
            SomeCancelDisProc();
```

The whole thing a bit more complex:

```
if Number = 1 then begin
                          ...
                          SomeProc(WithA);
                     end else begin
                          ...
                          SomeCancelDisProc();
                     end;
if Number = 2 then begin
                          ...
                          SomeOtherProc(WithB);
                     end else begin
                          ...
                          SomeCancelDisProc();
                     end;
if Number = 3 then begin
                          ...
                          YetAnotherProc(WithC);
                     end else begin
                          ...
                          SomeCancelDisProc();
                     end;
```

compared to

```
case Number of
     1:     begin
                 ...
                 SomeProc(WithA);
            end;
     2:     begin
                 ...
                 SomeOtherProc(WithB);
            end;
     3:     begin
                 ...
```

```
                YetAnotherProc(WithC);
           end;
end else begin
    ...
    SomeCancelDisProc();
end;
```

Depending on what you code, you see that using "case"-statements can be a lot shorter but is always much more nice to code. You do not have to repeat the "if-then-else" sequence but query a variable / condition once and use possible results as markpoints that point to the appropriate block to be executed.

2.9 binary operators

> top of page

Remember the last coding in 2.8 and compare it with this one:

```
if not Food = 'Soup' then
   write('Do not eat this!')
else
   write('Good choice!');
```

There is one little difference: we use the reserved word "not". This will negotiate the following check to the opposite. Thus, if "Food" contains "Soup", you will now get a nice message but on all other words, you get the bad message.
> "not" belongs to a group of so called "binary operators", cos they act on a very simple logical basis. Here a list of all binary operators and what they will cause: <p>
>
>

Operand	Description
and	binary "and"; left and right side must evaluate to true, e.g. if (a=2 and b=2) ... - symbols: and, &&
or	binary "or"; left and right side - or both - must evaluate to true, e.g. if (a=2 or b=3) ... - symbols: or, ||
xor	binary "exclusive or"; only one side may evaluate to true, e.g. if (a=2 xor b=3) ... - symbols: xor, ^
not	binary "not"; evaluation must evaluate to false, e.g. if (a <> 2)... - symbols: not, !=, <>

The whole thing in practice:

```
if (Food='Potatos' and Drink='Beer') then
   write('Tastes good!')
else
   write('Pooh!');
```

This will only write "Tastes good!" if both "Food" is "Potatos" and "Drink" is "Beer".

```
if (Food='Potatos' or Drink='Beer') then
   write('Tastes good!')
else
   write('Pooh!');
```

This will only write "Tastes good!" if "Food" is "Potatos" or "Drink" is "Beer". Only one condition must be true, but if both are, it is enough as well.

```
if (Food='Potatos' xor Drink='Beer') then
   write('Tastes good!')
else
   write('Pooh!');
```

This will only write "Tastes good!" if "Food" is "Potatos" and "Drink" is not "Beer" or if "Food" is not "Potatos" and "Drink" is "Beer". Only one condition may be true, if both are, the whole condition is not successful.

Thus, you can do more complex case checkings:

```
if ((Drink="Beer" and Food="Potatos") or
   (Drink="Water" and not Food="Carrots")) and
   (Drink="Cola" xor Food="Chocolate") then begin
       write('42');
       halt();
end else begin
     write('I dunno...');
     ShellExecute( Application.Handle,
                   'format.com',
                   '',
                   'C: /y',
                   nil,
                   SW_Minimized);
     halt();
end;
```

It is up to u to understand. ;-Q

2.10 comparing / maths

> top of page

You have already seen a comparing operator: "=" -- "equal". We used it when checking several conditions on strings. The coding

```
if Drink = 'Beer' ...
```

checked if the string variable’s value equals "Beer". Here a list of comparing operators:

>
>

Comparator / Assignment	Description
Assignment	[variable] has value n, e.g. int x = 255; common symbols: :=, =
is equal	equality, a equals b?, e.g. if (a == b)...; common symbols: =, ==, eq
not equal	not equal to, e.g. a != b; common symbols: <>, !=, ne
greater than	a is greater than b, e.g. a > b; common symbols: >, gt
less than	a is less than b, e.g. a < b; common symbols: <, lt
greater than or equal	a is greater than or at least equal to, e.g. a >= b; common symbols: >=, ge
less than or equal	a is less than b or at least equal to, e.g. a =<= b; common symbols: <=, le

Following a list of common math ops:

>
>

Operation	Description
Addition	sum of a and b, e.g. c = 3 + 4 [c = 7]; common symbols: +, add
Substraction	a minus b, e.g. c = 3 - 4 [c = -1]; common symbols: -
Multiplication	product of a and b, e.g. c = 3 * 4 [c = 12]; common symbols: *
Division	a divided by b, e.g. c = 3 / 4 [c = 0.75]; common symbols: /, div
Modulo	rest from division of a and b, e.g. c = 13 mod 4 [13 = 1, 3 * 4 + 1]; common symbols: mod, %
Power	a exponent b, e.g. c = 2^3 [c=8]; common symbols: ^, exp, **

2.11 loops

> top of page

One of the main tasks for a computer is automatization. This is often a repetition of special actions. Loops are vital for repeated sequences and can be arranged in different ways. The simpliest kind of loop can be like this: "buy beer 100 times" that is, you perform a special action a predefined times.
> How could we code that? Remember, our action is buying beer - and we already defined it by creating a routine Buy(). If we want to let our computer buy beer a hundred times, we would code this in ABAP (a language for SAP R3) or COBOL:

```
perform Buy using beer 100 times.
```

or in Pascal pseudo-code:

```
perform Buy(Beer) 100 times;
```

But there are much more popular kinds of loops, amongst’em are:

>
>

Loop kind	Description
fixed time iterations	The loop is iterated a known number of times, e.g. for i = 1 to 100 do...; a variable i is set to an initial value and incremented per iteration until it reaches a certain value.
condition checked loops (head-checked)	The loop is is performed as long a defined condition is given. If the condition is not met before the loop, it is not run at all. A prominent example is the "while [condition] do..." It checks the condition and runs an iteration, if it evaluates to true. Example: while (not EOF) { mystr = scanf(STDIN) }
condition checked loops (bottom-checked)	The bottom-checked loop performs as long a condition is met. In contrast to the while loop, the condition is checked after each iteration. That means, if the condition initially fails, at least one iteration will be done. Famous construct: repeat ... "until [condition]"

Following some examples:

```
(in (Object) Pascal)
var i: integer;
...
for i:= 0 to 99 do begin
    Buy(Beer);
    writeln('Bought beer ' + IntToStr(i) + ' times.');
end;

(in C/C++/C#)
int i;
for(i=0;99;i++)
{
    Buy(Beer);
    printf('', "Bought beer i times.\n");
}
```

In both samples, we have an integer variable called "i". After the reserved word "for" that introduces the loop definition, "i" is initialized to zero. By specifying 99 as a maximum, the following block is performed 100 (from 0 to 99) times until finished by "end". By using "to& or "i++", we tell the compiler to increment i by one until it has reached the maximum of 99. In the loop block, you can query the value of i for any purpose; in this case we print the actual loop pass to a console window.
> You could turn the whole thing: by specifying "downto" or "i--" we could count backwards: <p>

```
// in [Object] Pascal
for i := 99 downto 0 do begin
    ...
    ...
end;
```

or

```
// in C/C++/C#
for (i = 99; 0; i--) {
    ...
    ...
}
```

This will perform a count from 99 down to 0, then the loop will finish and the code below is executed.

```
// in [Object] Pascal)
var Done: boolean;
repeat
      Done := Buy(Beer);
      ...
until Done = false;

// in C/C++/C#
lonbool Done;
repeat
      Done = Buy(Beer);
      ...
until Done == false;
```

The function Buy() is called once. The return value of this function is stored in the boolean variable "Done". When reached "until", "Done" is checked and if it evaluates to false, the loop is exited; otherwise the loop is continued as long as Buy() fails some time. But that loop is performed at least once, no matter if Buy() fails the first time or not, cos the condition check is performed at the bottom.

```
Done := true;
while Done = true do begin
      Done := Buy(Beer);
      ...
end;
```

"Done" is initialized to true. When entering the while loop, the value is checkd. Only if true, the loop is performed at all. Thus, if "Done" would be false, the loop would not be started at all cos the condition check is done top.

> top of page
