Chapter 3 THE REPRESENTATION OF PROCESSING ALGORITHMS

3.1 Concepts

Algorithm is a fundamental concept of computer programming. An algorithm is represented as a set of rules (R_i) which can be applied to the same class of problems (CP_i) in order to obtain the solution (S) by means of some sequential unique operations (OS_i) started, eventually, with some initial conditions (CI_i): $S=R_i(OS_i(CP_i([CI_i])))$.

An algorithm is a prescribed set of well-defined instructions for solving a problem in a finite number of steps. In computing, algorithms are essential because they serve as systematic procedures that computers require [LK.99]. Thus, we can say that an algorithm is a solution of certain kind of problems that are referenced by the term computational or algorithmic problem.

There are many reasons for using algorithms from which at least the following three are very important:

- **Efficiency** – for certain types of problems resources have found over the time efficient ways for solving and implementing them. They exist also as procedures or functions in libraries usable in many programming languages;
- **Abstraction** – an algorithm provides a level of abstraction in solving problems. Generally every complex problem can be decomposed into simpler ones for which known algorithms exists;
- **Reusability** – algorithms are generally reusable for many different situations.

A program is a set of instructions written in a language designed to make a computer perform a series of specific tasks. The instructions tell computers exactly what to do and exactly when to do it. A programming language is a set of grammar rules, characters, symbols and words – the vocabulary – in which those instructions are written. Programming is the designing and writing programs.

A computer program – does no matter if is a single procedure or function, utility tool, an application or the operating system itself – is nothing else than a list of instructions that the microprocessor (or generally processor) can execute. In turn, an instruction is a specific pattern of bits representing a numeric code. The computer sends to the microprocessor the list of instructions that forms the program code, one by one. At receiving of an instruction the microprocessor sees from the instruction code what function is to be done and executes the corresponding actions. All the numeric codes allowed by a microprocessor as instructions form the machine language. The pattern of bits representing instructions is no easy to be manipulated as such by humans. For that reason they represented in a human meaningful form. The representation of instructions in a human meaningful form is called programming language.

Any algorithm must have the following properties:

- **Generality** – the algorithms must solve a class of problems not a particular one;
- **Finality** – the algorithm must find a solution in a finite number of steps;
- **Clarity** – the algorithm must specify all steps that must be realized to obtain a correct solution of the problem.

The problem solving process starts with the problem specification and ends with a concrete (and correct) program. The steps to do in the problem solving process may be: problem definition, problem analysis, algorithm development, coding, program testing and debugging, and documentation.

3.1.1 The Stages of Solving a Problem by Means of Computer

The stages of analysis, design, programming, implementation, and operation of an information system forms the life cycle of the system as described in §1.8. Here we briefly describe the steps in problem solving process by using a programming environment (it can allow the "around" application programming by the possibility of generating programs from general templates, for example) and by considering only a specific process from the whole system. In this context the stages can be:

1st. **Defining/Specifying the problem** [Theme] - by answering to questions as: What the computer program do? What tasks will it perform? What kind of data will it use, and where will get its data from? What will be the output of the program? How will the program interact with the computer user? Specifying the problem requirements forces you to state the problem clearly and unambiguously and to gain a clear understanding of what is required for its solution. Your objective is to eliminate unimportant aspects and to focus on the root problem, and this may not be as easy as it sound.

2nd. **Analyzing the problem** [Analysis] involves identifying the problem (a) inputs, which is the data you have to work with; (b) outputs, the desired results; and (c) any additional requirements or constraints on the solution. At this stage you should also determine the required format in which the results should be displayed (for example, as a table with specific columns, as a card with predefined items etc) and develop a list of problem constants and variables and their relationships. At this level, the relationships between variables can be expressed as formulas.

3rd. **Algorithm development**: find an algorithm for its solution [Design]. Designing the algorithm to solve the problem requires you to write step-by-step procedure – the algorithm – and then verify that the algorithm solves the problem as intended. Writing the algorithm is often the most difficult part of the problem-solving process. Don't attempt to solve every detail of the problem initially; instead, discipline yourself to use a top-down design. In a top-down design (also called divide and conquer), you first list the major steps, or sub-problems, that need to be solved, then solve the original problem by solving each of its sub-problems. For example, most computer algorithms consist of the following sub-problems:

1 - read the data;
2 - perform the computations;
3 - display the results.

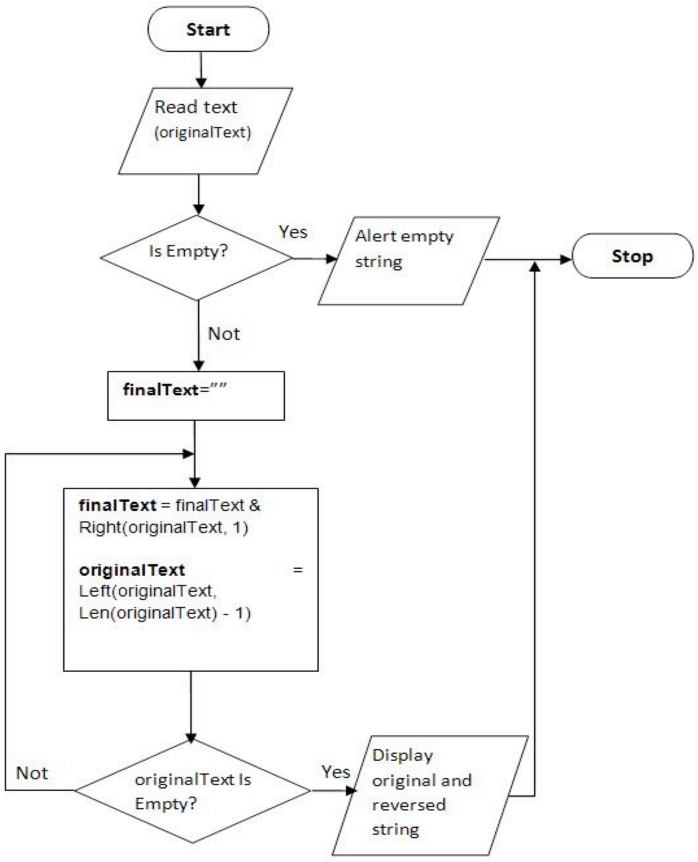
Once you know the sub-problems, you can attack each one individually. For example, the step 2, called "perform the computations", may need to be broken down in more detailed lists of steps (think to salary computation: each perceived tax means other computation formula). This process is called algorithm-refinements.

The development can be expressed as:

- **pseudocode** – a narrative description of the flow and logic of the intended program, written in plain language that expresses each step of the algorithm. For example, the problem is to write a program that will reverse all of the letters in any chunk of text provided to the program. The algorithm to solve that problem can be described in pseudo-code as:

```
01 Obtain the original text (or string) from the user.
02 If the user didn't supply any content, then signal that and quit now.
03 Prepare a destination for the reversed string, empty for now.
04 Repeat the following sequence [steps 5 to 7] until the original string is empty:
05    Copy the last character from the remaining original string.
06    Put that character onto the end of the destination string.
07    Shorten the original string, dropping the last character.
08 [End of repeat section]
09 Show to the user the original and destination string.
```

- **flowchart** - a graphical representation that uses graphic symbols and arrows to express the algorithms. The representation by flowchart for the reverse string solution can be:


Start
Read text (originalText)
Is Empty?
Yes
Alert empty string
Stop
Not
finalText=""
finalText = finalText & Right(originalText, 1)
originalText = Left(originalText, Len(originalText) - 1)
originalText Is Empty?
Not
Yes
Display original and reversed string

After you write the algorithm you must realize step-by-step simulation of the computer execution of the algorithm in a "so called" desk-check process (verifying the algorithm).

4th. **Coding** (or programming): is the process of translating the algorithm into the syntax of a given programming language [Programming]. You must convert each algorithm step into one or more statements in a programming language.

The implementation of the reverse string algorithm in Visual Basic .NET (only the subroutine corresponding) can be:

```
    Private Sub ReverseStringToolStripMenuItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles ReverseStringToolStripMenuItem.Click
0:     Dim originalText As String, finalText As String, oneCharacter As String, preservedText As String
1:     originalText = InputBox("Type text to reverse.")
2:     If (Len(originalText) = 0) Then MsgBox("Empty string supplied. No Action!", vbOKOnly + vbInformation) : End
3:     finalText = ""
       preservedText = originalText
4:     Do While (originalText <> "")
5:         oneCharacter = Microsoft.VisualBasic.Right(originalText, 1)
6:         finalText = finalText & oneCharacter
7:         originalText = Microsoft.VisualBasic.Left(originalText, _
             Len(originalText) - 1)
8:     Loop
9:     MsgBox("The original string is:" & preservedText & Chr(10) & Chr(13) & "The reverse is: " & finalText)
    End Sub
End Class
```

5th. **Testing and debugging**:

- **testing** means running the program, executing all its instructions/functions, and testing the logic by entering sample data to check the output;
- **debugging** is the process of finding and correcting program code mistakes:
  - **syntax errors** – identified, generally, at interpreting/compiling time; a syntax error occurs when your code violates one or more grammar rules of the used programming language. In VB environment the syntax errors signaled in the moment they produced by the VB interpreter.
  - **run-time errors** – are detected and displayed by the computer during the execution of a program. A run-time error occurs when the program directs the computer to perform an invalid operation, such as dividing a number by zero or manipulating undefined or invalid data;
  - **logic errors** (or so called **bugs**) – identified, generally, at run time occurs when a program follows a faulty algorithm. Because logic errors usually do not cause run-time errors and do not display error message they are very difficult to detect.
- **field testing** is realized by users that operate the software with the purpose of locating problems. The main purpose of *field testing* is verifying the program in order to eliminate the logic errors.

6th. **Documenting** the program by:

- **internal documentation** – the instructions and comments within the program itself. In VB programs the comments are placed by using the **Rem** command, as a separate line or preceded by the colon (:) character as a multi command line or ‘ (single apostrophe) as the inline comment;
- **external documentation** - the printed set of instructions (the user’s manual) describing how to operate the program (it can be also on-line documentation).

7th. **Integrate the program in the data process flow** (Implementation) and use the program to solve problems [Exploitation]. In that step the programmer must maintain and update the program, this means to modify the program to remove previously undetected errors and to keep it up to date as government regulations or company policies changes.

Before the computer can execute an assembly or a high-level language program, the programmer must enter the sentences of the program as a so called source program into the computer and the computer must store it in executable form in memory (expressed as machine language sentences). Several system programs, such as editors, compilers or interpreters, linkers etc assist with this task.

The General Steps for Preparing a Program

The algorithm found for a problem is ‘translated’ (coded) into a computer program by using a programming language (usually a high level one). The text of the program is entered into the computer’s memory by typing it in from the keyboard, reading from other storage devices (hard drive, memory stick, magnetic tape, etc.), receiving along the communication channels from other networked devices, etc. This program text, expressed in terms of sentences of a programming language, forms the so called source program (source code). To be executable for the processor, that executes and understands only the machine language, our source program goes through a number of transformations (syntactic and semantic analysis, translations, link editing etc.) that allows be expressed in the machine language.

The general steps for preparing a program (written in a programming language) for execution (figure 3.1) are the following:

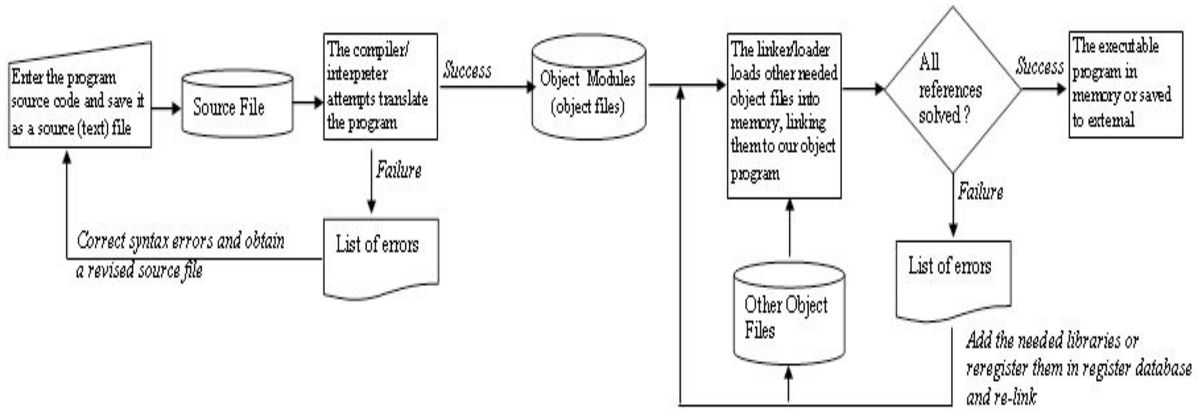

Enter the program source code and save it as a source (text) file
Source File
The compiler/ interpreter attempts translate the program
Success
Failure
List of errors
Correct syntax errors and obtain a revised source file
Object Modules (object files)
The linker/loader loads other needed object files into memory, linking them to our object program
Other Object Files
All references solved ?
Success
Failure
The executable program in memory or saved to external
List of errors
Add the needed libraries or reregister them in register database and re-link

Figure 3.1 Steps for preparing a program for execution

1. Use an editor program to enter each line of the source program into memory and save it to disk as a source file. Most programming environments offer an integrated editor.

> In VB the source code can be stored in modules (files with .bas extension) or in the source file associated to forms. Until you save explicitly the project you work on, all new source code (or the older one corresponding to loaded objects) is stored in temporary files created when loading the project and/or adding components. You can save your changes any time by pressing the *File*, *Save…* option or by pressing the Save button in the toolbar.

2. Use a compiler/interpreter program to translate the source program into machine language. If there are any syntax errors (errors in grammar), the compilers/interpreter displays these errors on the monitor. Use the editor program to correct the errors by editing and resaving the source program.

> In VB environment the syntax errors are signaled as they produced. You can correct them in the same editing session. In that way you can obtain a syntax error free code. [VB6] You can run the program by choosing between *Run*, *Start* (the interpreter only) command and *Run, Start With Full Compile* (the compiler). The compiler is started automatically when you select the *File*, *Make <project name>*.exe option.
> [Visual Studio 2005 (VS-05) and up] The *Build, Publish <project name>* starts the wizard that will allow you to specify the location where the installer is build and

which ”device” the user will use to later on install the application (web site, Universal Naming Convention [UNC] path or file share, CD-ROM/DVD-ROM).

3. When the source program is error free, the compiler saves its machine-language translation as an object program.

[VS-05] The output of the Build phase is a semi-compiled IL (intermediate language) and includes ready-to-execute versions of the original source code’s types and members (all this content can be “decompiled” using the disassembling tool *ildasm.exe* included in .NET package). The CLR (Common Language Runtime) realizes a final just-in-time (JTI) compile of IL assembly, to prepare it for use on the local platform.

4. The linker/loader program combines your object program with additional object files that may be needed for your program to execute (for example programs for input and output) and stores the final machine language in memory, ready for execution. The linker/loader can also save the final machine language program as an executable file on disk.

In VB environment you can obtain the executable file by activating the *File*, *Make <project name>.exe* option. If you intend to run your program on another computer is possible that the executable do not work. That happened because the program may need some libraries registered in the computer and they are not. To be sure that your program runs anyway use the “Application Setup Wizard” from your VB package to create a setup application program that includes all references needed by the program. The setup will help you to install correctly your program on a new computer system running under a compatible operating system.

Executing a Program

As we seen earlier the computer is a programmable device and the part called CPU is responsible with instruction interpretation and execution. The program must be represented in machine language (patterns of bits). To execute a machine language program, the CPU must examine each program instruction in memory and send out the command signals required to carry out the instruction. Although the instructions normally are executed in sequence, as we will discus later, it is possible to have the CPU skip over some instructions or execute some instructions more than once.

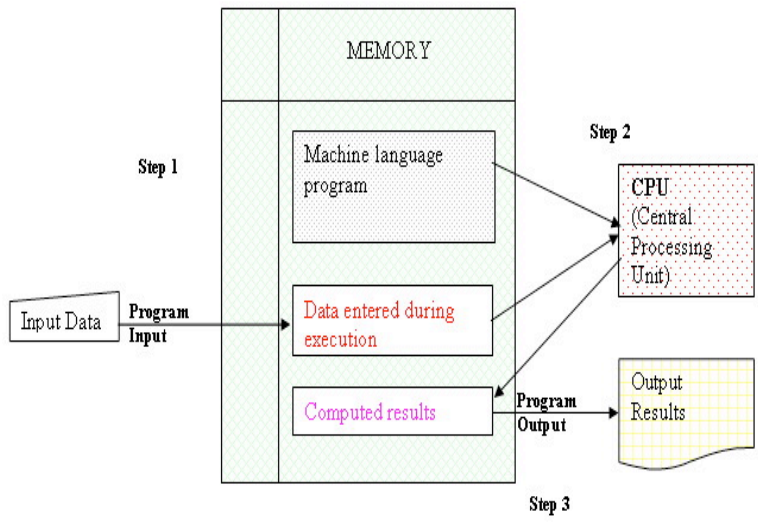


Figure 3.2 Executing a program

During the execution data can be entered into memory and manipulated in some specified way. Special program instructions are used for entering or reading a program's data (called input data) into memory. After the input data have been processed, instructions for displaying or printing values in memory can be executed to display the program result. The lines displayed by a program are called the program output. The flow of information during program execution can be summarized as in figure 3.2

3.1.2 The Description of Algorithms by Means of Logical Diagrams (Flowcharts)

Process, Program and Document Flowcharts

In the general analysis of information system the designer can use three kinds of flowcharts: document, process, and program flowchart and/or narrative descriptions in pseudocode.

Document Flowchart. A document flowchart is a diagram illustrating where documents originate and where copies of them are sent. The most useful kind of document flowchart shows what happens to the copies of a single document from the time they are created until each of them is in a file or in the hands of an outside party. A consolidated flowchart for a single process shows origins and fates of all documents in the process.

Process Flowchart. A process flowchart is a diagram that shows the data and operation for a single process. If a process is complex, several flowcharts may be needed to cover it complexity.

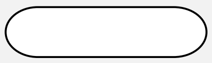
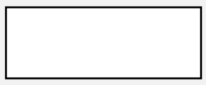
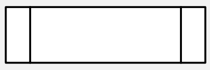
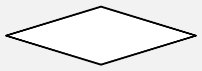
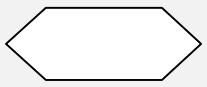
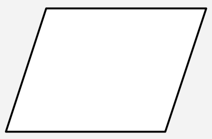
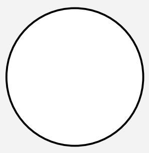
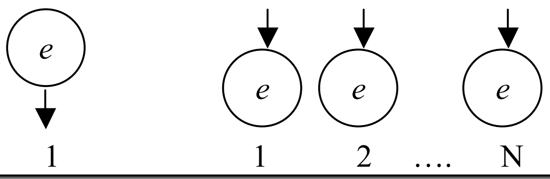
Program Flowchart. A program flowchart is a diagram that shows the data, steps, and logic of a process operation, does show logic and additional processing detail. It also allows you to make certain that formulas and calculations are included in the instructions to the programmer.

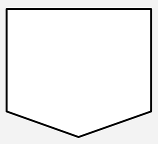
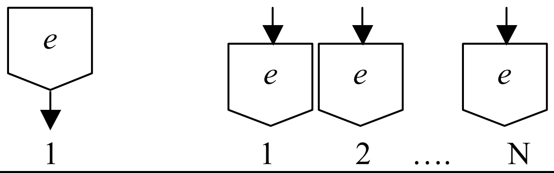
Pseudocode. The statements in the flowchart can be extracted from it and listed in order. They form, in this way, the basis for *pseudocode*. *Pseudocode* is a set of succinct instructions to the programmer using some of the syntax of the language in which the application will be programmed.

The Basic Symbols Used in Drawing Program Flowcharts

The symbols used for building logical diagrams (program flowcharts) are shown in the table 3.1.

Table 3.1 Symbols used in Drawing Program Flowcharts

1. 	**Terminal/Interrupt (Start/Stop)** - It marks the START and the STOP of the logical diagram and/or the beginning and ending of a procedure (function, subroutine). Inside is written, case usage dependent, either the word START/STOP (for main programs) or the call model for the procedure (function or subroutine);
2. 	**Process/Calculation** Block - It is used for representing calculus formula, changes of the index values, assigning of values. Inside are written the expressions of that operations;
3. 	**Procedure** - A call to another process or procedure. Inside is written the name of the procedure followed by the list of parameters in the order as specified in the call model;
4. 	**Decision** Block - It is used for the passing in different points, of the diagram, depending on the true state or false state of a logical condition;
5. 	**Preparation** – used to specify Open/Close operations on files (or computer ports, or connections to host computers or to long distance databases);
6. 	**Input/Output** Block - It is used to represent read and write operations on data. Inside is written the operation sense such as *reading* (expressed by commands as: Read, Input, Accept, Select, Get, In) or *writing* (expressed by commands as: Write, Display, Put, Update, Modify, Print, Out) followed by the logical device name (allowed by the system at opening) and the list of values/variables on which the command acts.
7. 	**Onpage Connector** - Used to link different points of the logical diagram in the same page. Inside is written a label (can be a digit/number or a word – is preferable to be of significance for the reader) that must be defined only once as entry point (the arrow goes from symbol) and how many times needed as exits (or go to). The entry labels must be unique for a particular flowchart; 

8. 	**Offpage Connector** - It links different points of the logical diagram in different pages. It has the same rules as the Onpage Connector. 
9.	**Flow** – Is a connection *from* … *to* that links all the blocks in the diagram and shows the transfer sense of the information.

Fundamental Structures Used in Algorithm Representation

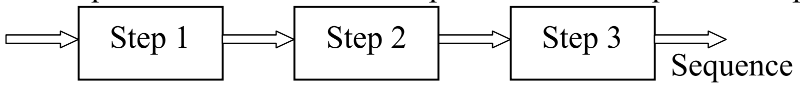
Each algorithm can be represented as a combination of three control structures:
1°. Sequential or process structure;
2°. Alternative or decision structure (If … then … else …);
3°. Loop (repeating, cycle) structure – the condition evaluated first (Do While)

From these basic structures are derived and used by programmers the following three structures:
4°. Case of (Switch case);
5°. Do Until – loop with the condition evaluated after;
6°. Counted loop structure (For … Next).

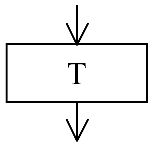
The sequence control of the next step in programs is usually carried out with the aid of various combinations of instructions used to model the control-flow structures or control structures, as the following:
- **Direct sequencing**, of the form ‘do A followed by B’ or ‘do A and then B’ [HF.04];
- **Conditional branching**, of the form ‘if C then do A otherwise do B’ or just ‘if C then do A’, where C is some condition [HF.04];
- **Repeating**, of the form ‘do A while C’ or ‘do A until not C’, where C is some condition, or ‘for Y from 1 to N do A’.

3.2 Sequential Structure

The sequential structure can be represented as a sequence of operations



or as a transformation block:



T from the graphical representation is a data transformation such as assignments and/or computations, data type declarations, input/output operations etc.

Example:

This algorithm realizes an interchange of the content of the variable named *x* with the content of the variable named *y*. The variables *x* and *y* from the name of the procedure *Exchange(x, y)* are called arguments and the real (actual) values for these are passed to the procedure at call time. The call is realized, in almost algorithmic programming languages (as C ++, Visual Basic, Pascal etc.), by writing the name of the procedure followed by the list of the arguments. The call *Exchange(x, y)* means applying the operations included in the procedure body to the content of variables *x* and *y*.

In the computation block from the figure 3.3 the numbered line means:

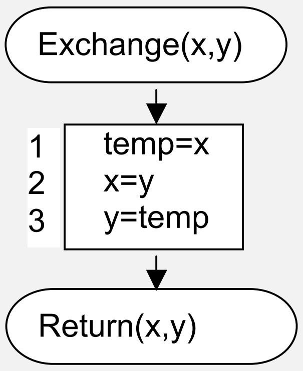


Figure 3.3 The representation of Exchange operation

- (1) the content of variable x is stored in a temporary (working) variable called *temp*;
- (2) the content of variable *x* is overwritten by the content of variable *y* (first is deleted and after deletion the content of *y* is copied);
- (3) the content of *y* is overwritten by those of the variable *temp* (it contains the value of *x* as passed at the call time).

The new values of x and y are returned to the caller.

For example, the call: *Exchange(3, 2)* realizes the operations:

```
temp = 3
x = 2
y = 3 (the temp content)
And returns 2, 3
```

Some programming languages describes, in the syntax of the call model, the return parameters and the usage sense IN, OUT, IN-OUT together with the accepted data type (Integer, Single, Double, String etc.).

3.2.1 Assignments

The *assignments* are represented, in the used programming languages, in one of the next following formats:

x←0 x=0 x:=0
store 0 To x
x = *expression*
variable = *expression*

The interpretation of the last form is: the *variable* before the assignment operator is assigned a value of the *expression* after it, and in the process, the previous value of *variable* is destroyed.

An *expression* can be an expression on character string, a logical expression or arithmetic expression. It can be a variable, a constant, a literal, or a combination of these connected by appropriate operators.

An assignment statement stores a value or a computational result in a variable and is used to perform most arithmetic operation in a program.

1. An *expression on character string* can be built (in VB but not only) using:

- the concatenation operator: & or +

- intrinsic functions for extracting substrings from a string variable or string constant such as:

Right(*string*,*number_of_characters*) - extracting substring from the end

Left(*string*,*number_of_characters*) - extracting substring from the beginning

- functions that manipulate strings:

Cstr(*expression*) – convert the expression in a character string;

Lcase(*string_expression*) – convert the string in lower case;

Ltrim(*string*), Rtrim(*string*), Trim(*string*) – eliminates the spaces (trailing) from left (leading blanks), right and, respectively left-right;

Str(*number*) – converts number in string;

Ucase(*string*) – converts string to uppercase.

In Visual Basic .NET all this functions are available in the namespace Microsoft.VisualBasic and must be prefixed with that construction, such as in the call Microsoft.VisualBasic.Right(”Mihail Eminescu”,8) that returns “Eminescu”.

2. A *logical expression* can be:

• **simple**, with the general syntax:

<*variable*>[<*relation_operator*><*variable*>]

or

<*variable*>[<*relation_operator*><*constant*>]

<*relation_operator*>::=<|<=|>|>=|=|<>

• **complex**, with the general syntax:

e_1 Eqv e_2 - equivalence;

e_1 Imp e_2 - logical implication;

o_1 Is o_2 - equal-to, compare two object reference string like pattern;

o_1 IsNot o_2 - not-equal-to, compare two object reference string like pattern [VS-05];

e_1 Xor e_2 - exclusive or;

e_1 Like e_2 - pattern operand, returns True if the first operand e_1 matches the string pattern defined by the operand e_2[VS-05];

<*logical_expression*$_1$><*logical_operator*><*logical_expression*$_2$>

where the *logical_operator* can be:

And, Or as binary operators (connectives);

Not as unary operator, returns the opposite of a Boolean operand;

AndAlso act just like And operator, but it doesn’t examine or process the second operand if the first one is False [VS-05];

OrElse act just like Or operator, but it doesn’t examine or process the second operand if the first one is True [VS-05].

The precedence of evaluation of logical operators is Not, And, AndAlso, Or, OrElse.

The logical functions works as explained in chapter two. Each one has an associated truth table that take carry of the two states True or False and, in some programming environments, a state called Empty (or Null) to distinguish between False an non value. If you want to see how this really works you must consult the VB programming environment.

3. An *arithmetic expression* uses the syntax:

<operand$_1$><arithmetic_operator><operand$_2$> where:

- *<operand$_1$>,<operand$_2$>* can be numeric variables, literals, constants or arithmetic expressions (or calls to functions that returns numeric values)
- *arithmetic_operator* (binary operators) is one of the following:

Operator	**Significance**	**Example** (for x=3)	**Evaluation Priority (in descending order)**
+	**Add**	**x+1→4**	**1**
-	**Subtract**	**x-1→2**	**1**
*****	**Multiply**	**4*x→12**	**2**
/	**Divide**	**x/2→1,5**	**2**
\ or div	**Integral division**	**x\2→1**	**2**
mod	**Modulus**	**x mod 2→1 remainder 1**	**2**
^	**Exponentiation**	**x^2→9**	**3**

"-" and "+" can be used as sign designator and are referenced as unary operators.

Or a shift operator [VS-05]:

Operator	**Significance**	**Explanation**	**Evaluation Priority (in descending order)**
<<	Shift Left	Shifts the individual bits in an integer operand to the *left* by the number of bits specified in the second operand (acts as multiplying the left operand by two at the power specified by the second operand). Example: **1024<<2 will produces 4096** (multiplied by 4, 2^2)	2
>>	Shift Right	Shifts the individual bits in an integer operand to the *right* by the number of bits specified in the second operand (acts as dividing the left operand by two at the power specified by the second operand). Example: **1024>>2 will produces 256** (divided by 4, 2^2)	2

The Visual Basic assignment can be combined with the basic arithmetic (with except for &, string concatenation) operators (similarly to C/C++ language) as described in the following table:

Operator	**Based On**	**Examples**	
		The Expression	**Is Equivalent With**
=	**Standard assignment operator**		
+=	**+ (Addition)**	**a+=1**	**a=a+1**
-=	**- (Subtraction)**	**a-=1**	**a=a-1**
***=**	*** (Multiplication)**	**a*=2**	**a=a*2**
/=	**/ (Division)**	**a/=2**	**a=a/2**
\\=	**\\ (Integer Division)**	**a\\=2**	**a=a\\2**
^=	**^ (Exponentiation)**	**a^=2**	**a=a^2**
<<=	**<< (Shift Left)**	**a<<=2**	**a=a<<2**
>>=	**>> (Shift Right)**	**a>>=2**	**a=a>>2**
&=	**& (Concatenation; strings)**	**a&=”this string”**	**a=a & ”this string”**

If the desired order of evaluation differs from the operator evaluation order (3, 2, 1) we can use parenthesis to form sub expressions, as so called embedded or nested expressions. In that case first evaluated is the most nested parenthesis.

For example, the computation of the real root of the polynomial in 2 can be written as: **(-b + sqr(b^2 – 4 * a * c)) / (2*a).**

If an operand in the expression is a function call, generally expressed as *FunctionName(argument list)*, first the function is evaluated and his invocation is replaced with a value having the function return data type and later on, the operand linking the function call to another operand evaluates.

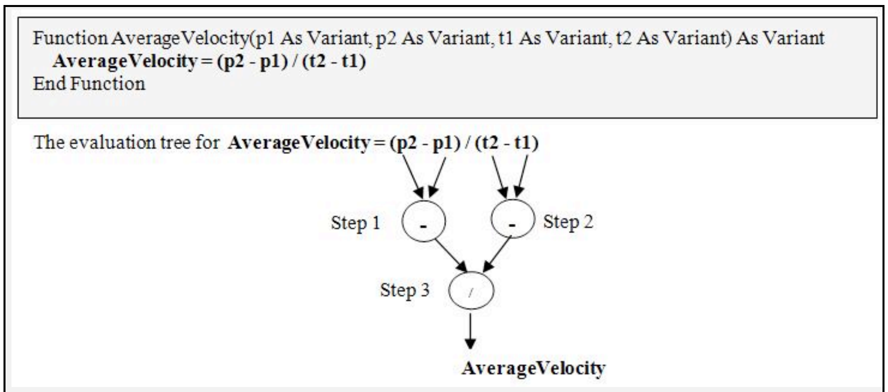
Examples:

Assignment	**Interpretation**
x=3.14	The value 3.14 is assigned to the variable named x
Delta= b^2 – 4*a*c	The evaluation of expression **b^2 – 4*a*c** is assigned to the variable named Delta
i=i+2	The actual value of variable i is given by the old value of variable i to each value 2 added
x=Abs(x)	The actual value of x is the absolute value (Abs) of his older value

If an expression contains more than one operator and/or parentheses the following rules of evaluation applies:

1. **Parentheses rule**: All expression in parentheses must be evaluated separately. Nested parenthesized expressions must be evaluated from inside out, with the innermost expression evaluated first.

Example:
The formula for the average velocity, *v*, of a particle traveling on a line between point p_1 and p_2 in time t_1 to t_2 is $v = \dfrac{p_2 - p_1}{t_2 - t_1}$. This formula can be written and evaluated in VB as shown in figure 3.4.



```
Function AverageVelocity(p1 As Variant, p2 As Variant, t1 As Variant, t2 As Variant) As Variant
   AverageVelocity = (p2 - p1) / (t2 - t1)
End Function
```

The evaluation tree for **AverageVelocity = (p2 - p1) / (t2 - t1)**

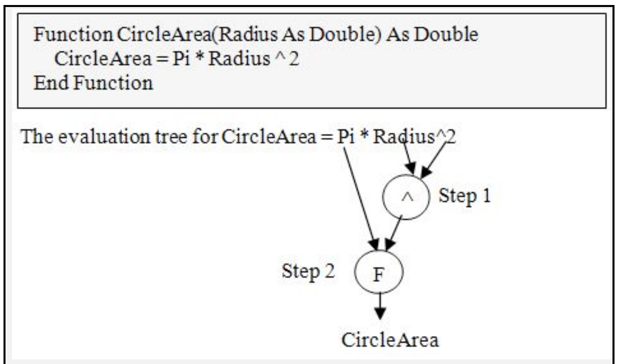
Figure 3.4 Using the Parentheses rule for evaluating expressions

2. **Operator precedence rule**. Operators in the same expression are evaluated from left to right in the following order:

^	first
*, /, mod, \, div, >>, <<	second
+, -	last

Example:

The circle area is given by the formula *Area=πr²* that can be written in VB as *CircleArea=Pi*Radius^2*. The formula can be written and evaluated as shown in figure 3.5.



```
Function CircleArea(Radius As Double) As Double
   CircleArea = Pi * Radius ^ 2
End Function
```

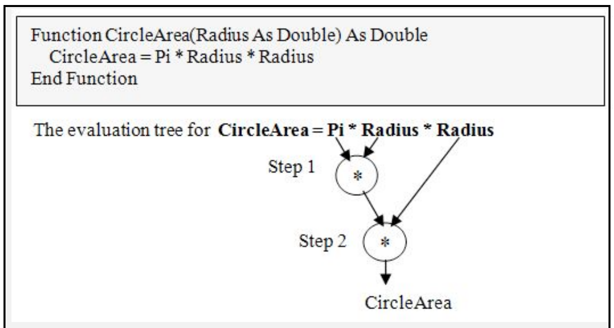
The evaluation tree for CircleArea = Pi * Radius^2

Figure 3.5 Using the operator precedence rule

3. **Left associative rule**. Operators in the same expression and at the same precedence level are evaluated left to right.

Example:

The circle area can be written in VB as *CircleArea=Pi*Radius*Radius* and is evaluated as shown in figure 3.6.



```
Function CircleArea(Radius As Double) As Double
   CircleArea = Pi * Radius * Radius
End Function
```

The evaluation tree for **CircleArea = Pi * Radius * Radius**
Step 1
*
Step 2
*
CircleArea

Figure 3.6 Using the left associative rule

3.2.2 Literals

Some basic data values, such as numbers, date values, strings etc can be included into the source code of a program just as they are. The literals accepted and the rules of defining them are the following:

String Literals. String literals are always surrounded by quote marks (can be up to about two billion characters in length). The character literal is exactly one character in length and is recognized by the *c* trailing after the string, for example “A”c designates the capital “A” character.

Date and time literals. Date and time literals are surrounded by the number sign (pound) and can be specified in any format recognized by Microsoft Window in a specific region, as for example #11/17/2008#.

Numeric literals. There are 11 different kinds of numeric data values, both integers and floating point values, which can be defined by typing the number right in the code, like 27, or 3.1415926535. Visual Basic also lets you specify which of the 11 numeric types to use for a specific number, by appending a special character to the end of the number. For example, normally, 27 is an integer and can be a floating-point "decimal" by appending an @ sign: 27@ floating-point decimal. If you do not specify for a number the trailing character specifying his type the number will be typed, depending of his magnitude and format (real/integer) automatically by the compiler. The type allows to properly align the result of computations, in which those literals involved, to the largest precision and to check if they properly used.

Boolean literals. Boolean values represent the simplest type of computer data: the bit. Boolean values are either true or false, on or off, yes or no, etc. Visual Basic includes the Boolean literals True and False.

The literals supported by Visual Basic and the trailing characters used to specify the type are the following:

Literal Type	Example	Description
Boolean	True	The Boolean data type supports two literal values: True and False.
Char	"Q"c	Single-character literals appear in double quotes with a trailing character c. A literal of type Char is not the same as a single-character literal of type String.
Date	#11/7/2005#	Date or time literals appear between a set of number signs and can include dates, times, or a combination of both. The date or time values can be in any format recognized by Windows, although Visual Studio may reformat your date literal for conformity with its own standards.
Decimal	123.45D 123.45@	Floating point values of type Decimal are followed by a capital D, or the character @.
Double	123.45R 123.45#	Floating point values of type Double are followed by a capital R, or the character #. Also, if you use a numeric literal with a decimal portion, but with no trailing data type character, that literal will be considered as a Double.
Hexadecimal	&HABCD	The hexadecimal literals start with the "&H" character sequence, followed by the hex digits.
Integer	123.45I 123.45%	Integral values of type Integer are followed by a capital I, or the character %.
Long	123.45L 123.45&	Integral values of type Long are followed by a capital L, or the character &.
Octal	&O7654	You can include octal literals in your code by starting the value with the "&O" character sequence, followed by the octal digits.
Short	123.45S	Integral values of type Short are followed by a capital S.
Single	123.45F 123.45!	Floating point values of type Single are followed by a capital F, or the character !.
String	"Simple" "A ""B"" C"	String literals appear within a set of double quotes, with no special character following the closing quote. A quote character within the string literal will be typed twice, as in the second example which will produce the string «A "B" C».

3.2.3 Variables and Constants Declarations

The variable that appears in an assignment statement (and not only) in the left side, or the variables and the constants used in expressions must be declared explicitly. A variable declaration is not an executable sentence. These kinds of sentences are addressed to the program language compiler and/or interpreter that “eat” them (in the order they appear, as a sequence). The result of “eating” is a memory reservation with a naming, maybe.

The naming of constants and variables in VB uses the rules:

1. An identifier must begin with a letter;
2. Can’t be longer than 255 characters;
3. Can’t contain embedded period or embedded type declaration character;
4. Must be unique in same scope (the range from which the variable can be referenced).

Variables: are named storage locations that can contain data that can be modified during program execution (they are reusable). The variables are the memory cells used for storing program’s input data and its computational results. Using variables is a two step process: declaration and assignment. The explicit declaration of variables is realized in VB by using the Dim statement:

Dim *variable*[As *data_type*] [,*variable*[As *data_type*]]…

where:

- *data_type* can be one of Byte, Boolean, Integer, Long, Single, Double, Currency, Decimal, String, Date, [*user_defined*], Variant, Object as described in table 3.1;
- *variable* is a user identifier defined by following the naming rules.

The explicit declaration of variables can be combined with the assignment:

Dim *variable*[As *data_type*][=*value1*][,*variable*[As *data_type*]][=*value2*]…

Table 3.1 Visual Basic Data Types

Data Type		**Required Memory**	**Boundary**
Byte		1 byte	0 to 255
Boolean		2 bytes	**True** or **False**
Char	*	2 bytes	**0 to 65535**
SByte	*	1 byte	-127 to 128
Integer (Int16) **Short**	*	2 bytes	-32,768 to 32,767
Integer (Int32)	*	4 bytes	-2,147,483,648 to 2,147,483,647
Long (long integer)		4 bytes	-2,147,483,648 to 2,147,483,647
Long (Int64)	*	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Single (floating-point)		4 bytes	-3.402823E38 to -1.401298E-45 for negative numbers; 1.401298E-45 to 3.402823E38 for positive numbers
Double (floating-point)		8 bytes	-1.79769313486232E308 to -4.94065645841247E-324 for negative numbers; 4.94065645841247E-324 to 1.79769313486232E308 for positive numbers
Currency (scaled integer)		8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807

Decimal		14 bytes	+/-79,228,162,514,264,337,593,543,950,335 without decimal mark; +/-7.9228162514264337593543950335 with 28 positions for the fraction and the less number <>0 equal to +/-0.0000000000000000000000000001
Date	*	8 bytes	1 January 100 to 31 December 9999 Jan 1, 1 AD to Dec 31, 9999 AD
Object		4 bytes	Any reference to an **Object**
String (variable length)		10 bytes + string length	0 to ≈2 billions (uses UNICODE)
String (fixed length)		string length	1 to ≈65,400
Variant (with numbers)	N	16 bytes	Any number value from the double domain
Variant (with characters)	N	22 bytes + string length	0 to ≈2 billions
UInteger (UInt32)	*	4 bytes	0 to 4,294,967,295 (unsigned integers on 32 bits)
ULong	*	8 bytes	0 to 18,446,744,073,709,551,615
UShort	*	2 bytes	0 to 65535
User-defined[1)] (using **Type**)		Number of required elements	The size of each element is the same with that for the associated data type.

* – Available only in Visual Studio 2005
N – Not available in Visual Studio 2005

[1)]The user defined data type is formed by placing the needed declarative sentences Type block. For example, if we want represent the structure of a row from the Balance_Sheet (the Romanian one) this can be declared by the user as follows:

```
Type Balance_Sheet_Row
        Dim Account_ID As String*22
        Dim Account_Description As String*60
        Dim Db_Jan As Double
        Dim Cr_Jan  As Double
        Dim Db_Prev As Double
        Dim Cr_Prev As Double
        Dim Db_Month As Double
        Dim Cr_ Month As Double
End Type
```

After declaration a user-defined data type can be used in the same way VB data type used. For our example, we can define a memory variable, that we call Current_Row, thus: `Dim Current_Row As Balance_Sheet_Row`

The declaration of references must contain instantiation, such as:

```
Dim aReferenceToAnObject As Object = New Object
```

For strings you don’t have to use new if a literal assigned:

```
Dim aNullString as String = ""
```

A string instance that is initialized with a character repeated a number of times, suppose 20 stars: `Dim starValue As String = New String("*"c,20)`
Such declarations can be split in two parts - declaration and assignment - as in the example:

```
Dim starValue As String
starValue = New String("*"c,20)
```

Constants: can appear as such anywhere as literals, intrinsic constants available in the Visual Basic programming environment or in other Windows applications, or as declarations in the declarative part of the program. Literals can be used only once in the code; if you want use many times you must declare them each time. The constants are like a cross between literals and variables: they have a single never-changeable value just as literals but must be declared and assigned with the value just as variables.

Examples:

Constant	Type
"Welcome to the information century!"	string
$25,000.00	currency
3.14	positive real number
-123	negative integer number
0.123e+3	number written in the scientific notation
“11/12/2009”	date

Constants can be defined as a declaration statement by using the syntax:

Const *constantName*[As *data_type*]=*expression*[,…]

where:

- *data_type* can be one of Byte, Boolean, Integer, Long, Single, Double, Currency, Decimal, String, Date, [*user_defined*], Object, etc as described in table 3.1;
- *constantName* is an user identifier defined by following the naming rules;
- *expression* an expression evaluated to an agreed data type whose evaluation is considered the default value for the constant.

Examples:

```
Const Pi As Single = 3.14159
Const Vat As Single = 0.19, Star As String = “★”
```

A declared constant is a named storage location that contains data that cannot be modified during the program execution. The most used constants are number constants and string constants. A string constant is sequences from 0 to 1024 characters enclosed in quotes.

3.2.4 Input/Output Operations by Using InputBox and MsgBox Functions

VB offers a variety of sentences for input/output operations. Even the flowchart symbol is not the same as the one used to represent the sequence structure the behavior is similar and all this operations are executed as sequences. For instance we introduce now two functions: input - InputBox(…) and output - MsgBox(…) used to activate a standard dialog.

InputBox. Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns a string containing the contents of the text box.

Syntax:
InputBox(*prompt*[, *title*] [, *default*] [, *xpos*] [, *ypos*] [, *helpfile*, *context*])

where:

Argument	**Description**
prompt	Required. Is a string expression that is displayed as the message in the dialog box. Can be up to 1024 characters in length. If ***prompt*** consists of more than one line they must be separated by a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return–linefeed character combination (**Chr(**13**)** & **Chr(**10**)**).
title	Optional. String expression displayed in the title bar of the dialog box. If you omit ***title***, the application name is placed in the title bar.
default	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit ***default***, the text box is displayed empty.
xpos	Optional. A numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If ***xpos*** is omitted, the dialog box is horizontally centered.
ypos	Optional. Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If ***ypos*** is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.
helpfile	Optional. String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If ***helpfile*** is provided, ***context*** must also be provided.
context	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If ***context*** is provided, ***helpfile*** must also be provided.

Example:

The call InputBox("Prompt", "Valoare_implicita", "Titlu") will produces the dialog box from figure 3.7.

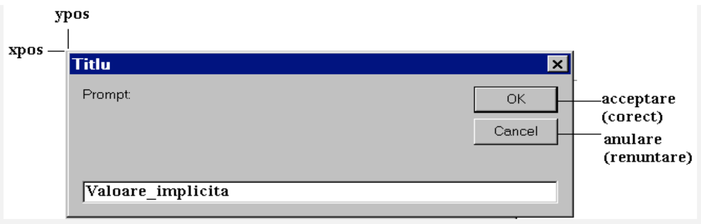


Figure 3.7 Example of using InputBox

MsgBox. Displays a message in a dialog box, waits for the user to click a button, and returns an **Integer** indicating which button the user clicked.

Syntax:

MsgBox(*prompt*[, *buttons*] [, *title*] [, *helpfile*, *context*]**)**
where:

Argument	Description
prompt	Required. Is a string expression that is displayed as the message in the dialog box. The maximum length of ***prompt*** is approximately 1024 characters depending on the width of the characters used. If ***prompt*** consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return – linefeed character combination (**Chr(**13**)** & **Chr(**10**)**) between each line.
buttons	Optional. A numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for ***buttons*** is 0.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit ***title***, the application name is placed in the title bar.
helpfile	Optional. String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If ***helpfile*** is provided, ***context*** must also be provided.
context	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If ***context*** is provided, ***helpfile*** must also be provided.

The value for ***buttons*** argument can be determined as a sum of the following Visual Basic constants:

Constant	Value	Description
vbOKOnly	0	Display **OK** button only.
vbOKCancel	1	Display **OK** and **Cancel** buttons.
vbAbortRetryIgnore	2	Display **Abort**, **Retry**, and **Ignore** buttons.

vbYesNoCancel	3	Display **Yes**, **No**, and **Cancel** buttons.
vbYesNo	4	Display **Yes** and **No** buttons.
vbRetryCancel	5	Display **Retry** and **Cancel** buttons.
vbCritical	16	Display **Critical Message** icon.
vbQuestion	32	Display **Warning Query** icon.
vbExclamation	48	Display **Warning Message** icon.
vbInformation	64	Display **Information Message** icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

For example the call MsgBox("Prompt",vbInformation+vbOkCancel, "Titlu") will produces the dialog shown in figure 3.8. The returned values correspond to the pressed button:

Constant	Value	Description
vbOK	1	**OK**
vbCancel	2	**Cancel**
vbAbort	3	**Abort**
vbRetry	4	**Retry**
vbIgnore	5	**Ignore**
vbYes	6	**Yes**
vbNo	7	**No**

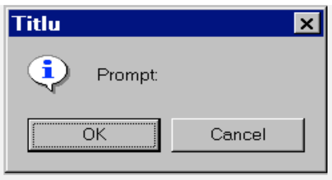


Figure 3.8 Example of an MsgBox dialog

In the following example is illustrated how to write the message on many lines and what means a string expression:

MsgBox "The Student " & Name & **Chr**(13) & **Chr**(10) & " has the average mark: " & media

What is between "…" means literal strings that will be placed as such in the message; *Name* and *media* are named variables whose content will be concatenated by the string concatenation operator (&) with the literals and the function calls **Chr**(13) & **Chr**(10) means display what follows on a separate line.

3.3 Alternative Structure (Decision)

The decision structure (figure 3.9) is used for choosing an alternative (an operation or block of operations) from two possible alternatives. Algorithm steps that select from a choice of actions are called decision steps.

The decision block can be expressed in a natural language as:

- evaluate the expression that defines the logical condition <*condition*>;
- If the result of evaluation is True
  Then execute *operation*$_1$
  Else execute *operation*$_2$;
- continue the execution with the next step in the flow

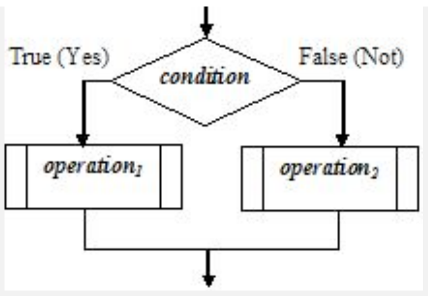


Figure 3.9 The decision block

The logical condition <*condition*> is a logical expression that will be evaluated either to True or either to False. The logical conditions can be simple or complex logical conditions.

A simple logical condition has the general syntax:

<*variable*> [<*relation_operator* ><*variable*>]

or

<*variable*> [<*relation_operator* ><*constant*>]

The *relation_operator* can be one of:

Relation Operator	Interpretation
<	**Less than. Example:** delta < 0
<=	**Less than or equal. Example:** delta <= 0
>	**Greater than. Example:** delta > 0
>=	**Greater than or equal. Example:** delta >= 0
= (in C++ the operator is ==)	**Equal to. Example:** a = 0
<> (in C++ the operator is !=)	**Not equal. Example:** a<>0

If <*variable*> is number or Boolean then is possible to directly compare with 0, respectively True and is not necessary to write the equal relation operator.

The simple logical conditions will be connected by the **AND**, **OR, and NOT** logical operators to form complex conditions. The logical operators are evaluated in the order NOT, AND, and OR. The change of the natural order of evaluation can be done, by using round parenthesis, in the same way as for arithmetic expressions. The precedence of operator evaluation in Boolean expressions (logical expressions) is:

Not
^,*, <<, /, >>, div, mod, and
+, -, or
<, <=, =, <>, >=, >

If … Then … Else. The alternative structure can be expressed in pseudocode as:

Visual Basic (Access, VBA)	C++ (Java)
If *condition* **Then** *operation*$_1$ **Else** *operation*$_2$ **End If** Each condition's Then (true branch) keyword is followed by one or more VB statements that are processed if the condition evaluates to True. Each condition's Else (false branch) keyword, is followed by one or more statements that are processed if the condition evaluates to False.	**if** (*condition*) *operation*$_1$; **else** *operation*$_2$; If one of operations includes a sentences sequence then this sequence will be included in a sentence block: { *operation*$_i$; }
PASCAL	
If *condition* **Then** *operation*$_1$ **Else** *operation*$_2$;	If one of operations includes a sentences sequence then this sequence will be included in a sentence block: **Begin** *operation*$_i$ **End**

Example:

We design the logical flowcharts (figure 3.10) and the corresponding procedures to determine the min value - Min(x, y), and max value - Max(x, y) from two values passed in arguments.

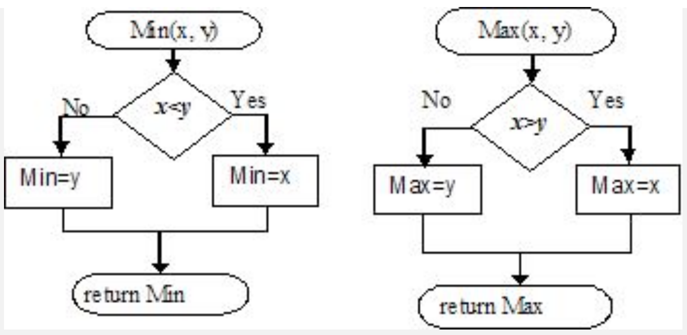


Figure 3. 10 A description of min and max algorithms for any kind of data type values

An implementation of Min and Max	

Function Min(x As Variant, y As Variant) As Variant
    If x > y Then
      Min = y
    Else
      Min = x
  End If
End Function
	Function Max(x As Variant, y As Variant) As Variant
	    If x < y Then
	      Max = y
	    Else
	      Max = x
	  End If
	End Function

It is possible do not have a specific operation on the two branches, that mean situations as depicted in figure 3.11. If the condition is true then the set of sentences placed between If and End If are executed.

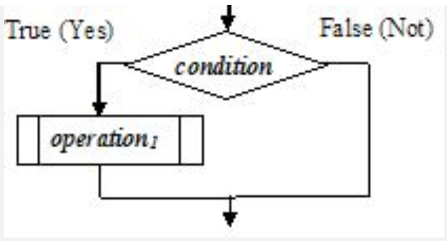


Figure 3. 11 The decision block without the Else branch

The decision block (figure 3.11) is expressed in a natural language as:

- evaluate the condition;

- *if* the result is True *then* execute operation$_1$;

- *else* continue the execution of the program.

The pseudocode can be expressed in one of the formats:

Basic (Access, VBA)	**C++ (Java)**
If *condition* Then *statement* If the condition is True then the statement is executed	**if** (*condition*) *operation*;

If *condition* Then
  Sequence of statements$_1$
End If
If the condition is True Then the group between Then and Else will be executed Else the group of sentences between Else and End If will be executed.
Inline if:
IIf(*condition*, *trueReturn*, *falseReturn*)
If the condition is true then the evaluation of the expression trueReturn will be returned
	if (*condition*) {
	    operations;
	}

Else the evaluation of the expression falseReturn will be returned	
PASCAL	

If *condition* **Then**
    operation;
	If *condition* **Then**
	    Begin
	        operations;
	    End;

Almost programming languages allow nesting *if ... then ... else* sentences (one if statement inside another) to form complex decision structures (decisions with multiple alternatives).

Examples:

a) Another implementation of Min and Max	

Function Min(x As Variant, y As Variant) As Variant
    Min=x
    If x > y Then Min = y
End Function
	Function Max(x As Variant, y As Variant) As Variant
	    Max=x
	    If x < y Then Max = y
	End Function

A special case of the If sentence is the sentence *if...then...elseif...* available in VB (a similar structure is available, for example, in the procedural component PL/SQL of Oracle DBMS). The nested If can be coded as a multiple-alternative decision. The syntax for that nested If is:

```
If condition_1
        Then
                sequence_1
        ElseIf condition_2  Then
                sequence_2
        ⋮
        Else
        ⋮
End If
```

For the first time *condition*$_1$ is tested. If the result is False *condition*$_2$ is tested and so on until a True evaluated condition reached for each the associated sentence block executed. After executing the reached block, the control of processing is passed to the next sentence after End if. If no condition evaluates to True then the sentence block associated to the *Else* branch executes (if Else defined; if not nothing executes).

Examples:

a) using nested If:

```
Sub Factorial_Call()
 n = InputBox("Type the value for n:", "VB Samples: n!")
 If IsNumeric(n) = True Then

   If Int(n) = n Then

     ‘ Call to the iterative implementation of factorial function
     valReala = Factorial_Iterativ(n)
     Raspuns = MsgBox("The Factorial is: " & valReala, vbInformation + vbOKOnly)

     ‘ Call to the recursive implementation of factorial function
     valReala = Factorial(n)
     Raspuns = MsgBox("The Factorial is: " & valReala, vbInformation + vbOKOnly)

   End If

 End If
End Sub
```

b) We redefine the Min and Max functions by using the procedure Exchange:

```
Function Min(x As Variant, y As Variant) As Variant
   If x > y Then Exchange x, y
   Min = x
End Function
```

```
Function Max(x As Variant, y As Variant) As Variant
   If x < y Then Exchange x, y
   Max = x
End Function
```

c) We define a procedure that take as arguments the values of x and y and arrange them in ascending order called *Asc(x,y)*. We do that in two versions: first realize all the operations and the second call the procedure *Exchange(x,y)*. In these example the values x and y are arranged (sorted) in ascending order, in conformity with the used comparison operator (< in our case). The algorithm can be conformed to any desired ordering by changing accordingly the

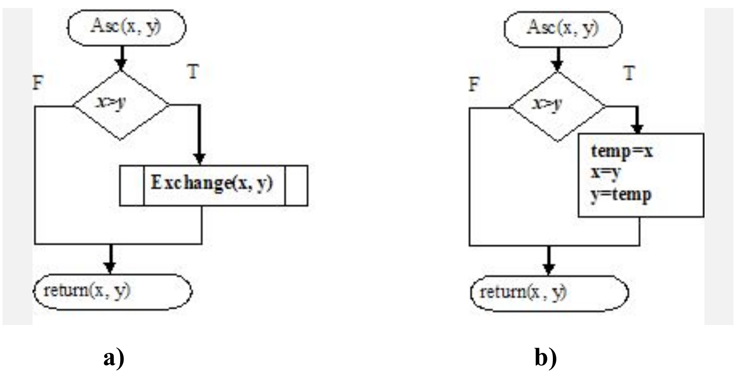


Figure 3.12 The definition of the algorithm for ascending ordering

comparison operator.
The implementations of the algorithms from figure 3.12 are:

```
' By using the function exchange (switch)

Sub Asc(x As Variant, y As Variant)
   If x > y Then Exchange x, y
End Sub
```

```
' By programming all operations

Sub Asc(x As Variant, y As Variant)
   Dim temp As Variant
   If x > y Then
      temp = x
      x = y
      y = temp
   End If
End Sub
```

d) this sequence chooses the case for VAT percent for a VAT calculator:

```
Private Sub PTvaGen()
 If PTva1.Value = True Then
    PrTVA = PrTVA1
    PTva2.Value = False
    Ptvax.Value = False
    EtAltTVA.Visible = False
    AltTVA.Visible = False
    ElseIf PTva2.Value = True Then
           PrTVA = PrTVA2
           PTva1.Value = False
           Ptvax.Value = False
           EtAltTVA.Visible = False
           AltTVA.Visible = False
           ElseIf Ptvax.Value = True Then
                  PrTVA = PrTVAx
                  PTva1.Value = False
                  PTva2.Value = False
                  EtAltTVA.Visible = True
                  AltTVA.Visible = True
                  FrmTva.Refresh
  Else
   MsgBox "Att.! Computation Error!",vbCritical
  End If
End Sub
```

Case of. Executes one of several groups of statements depending on the value of an expression (called selector). The case structure (and statement) can is especially used when selection is based on the value of a single variable or a simple expression (called the case selector): it compares a single value against a set of test cases values. A proposed graphical representation is shown in figure 3.13.

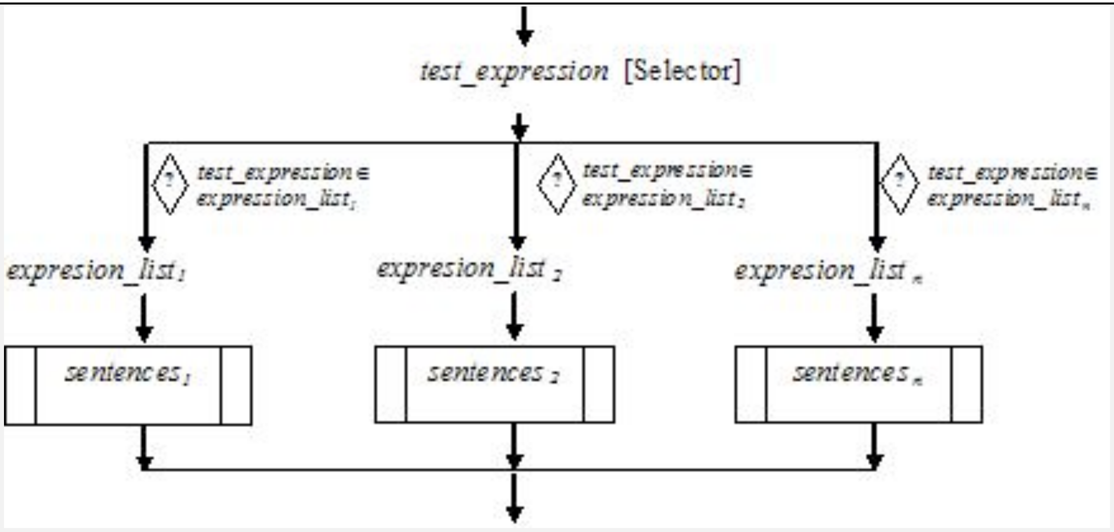

test_expression [Selector]
test_expression ∈ expression_list₁
test_expression ∈ expression_list₂
test_expression ∈ expression_listₙ
expresion_list₁
expresion_list₂
expresion_listₙ
sentences₁
sentences₂
sentencesₙ

Figure 3.13 The Case Of structure

Each *expression_list* is described as one or more values separated by comma.

First syntax:

Basic (Access, VBA)	**C++**

Select Case *test_ expression*
[**Case** *expression_list*$_1$
[*sentences*$_1$]]
[**Case** *expression_list* $_2$
[*sentences* $_2$]]
⋮
[**Case Else**
[*sentences* $_n$]]
End Select
	switch (*expression_int*) {
	case *constant_expression*$_1$:
	operations$_1$
	case *constant_expression*$_2$:
	operations $_2$
	⋮
	default:
	operations $_n$
	}

- each *expression_list$_i$* is represented (or formed) by one or many comma separated values (value list); - in the block **Select Case** the case **Else** can appear only once and only as a last case; - if many cases fit to *test_ expression* then the first founded will be executed; - each sentence block (*sentences$_i$*) can include zero, one or many sentences; - the evaluation of the test expression is realized only once at the beginning of the Select Case structure.	- *expression_int* is an expression that must produced an integral value (int); - *constant_expression$_i$* must be a constant expression; - the label **default:** can by be used only once.
PASCAL	

Case *expression* **of**
    label$_1$:
        operations$_1$;
    label$_2$:
        operations$_2$;
    .
    .
    .
    else or **otherwise**
        operations$_n$;
end
	- expression is also called the selector of instruction Case. The case selector must be an ordinal data type (data types Integer, Boolean and Char are ordinal types, but data type Real is not), or a data whose values may all be listed.
	- *label$_1$, label $_2$, ..., label $_i$,* ... are list of possible values of the selector (they must be of the same data type with the selector);
	- if the value of the selector don't fit to a label the operations specified on branch Else (otherwise) will be executed;
	- the values of constants must be unique for a *label$_i$*.

Second Syntax:

Choose(*index*,*choice_1*[,*choice_2*,…[,*choice_n*]])

The return is Null if *n<index<1*.

Example:

Suppose we want to build a simple calculator (figure 3.14) that deals with formulas of the type:

result = *operator* op_1 or **result** =op_1 *operator* op_2

where the variables op_1 and op_2 are numeric data types.

The choosing of the right computation is realized depending on the property **Caption** of the button. Each button is included into an object collection called *Operator()* and to each button is associated an index.

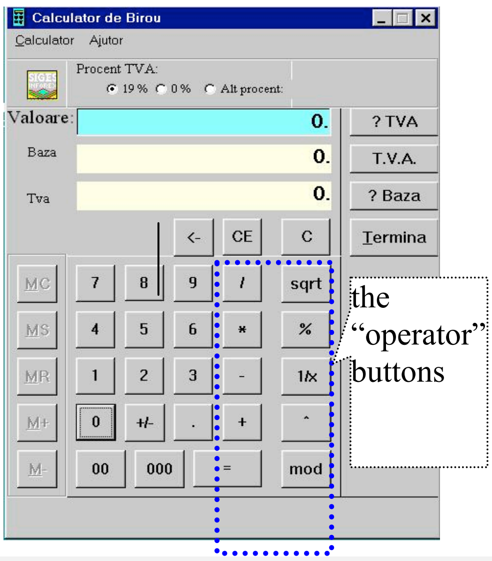


Figure 3.14 An example of a VAT calculator

The selection of the Case (the operation desired=the button pressed) is realized depending on a flag indicator (*OpFlag*) in which is stored the value of the Caption property. The following code sequence represents the operator analyzer of a desk computer allowing the direct VAT elements computation (Romanian rules). The sequence calls user defined functions as AfiseazaBazaTva, for example, to which you must don't think. You can see how the CASE OF pseudocode can be used in practice:

```
Select Case OpFlag ‘ OpFlag contains the pressed Operator key

 Case "1/x"
   AfiseazaBazaTva
   If Op1 = 0 Then
       MsgBox "Nu pot sa impart la 0 [ZERO]", 48, "Calculator"
   Else
       Op1 = 1 / Val(Op1) :Readout = Format(Op1, "###,###,###,###,###.00")
   End If

Case "%"
   AfiseazaBazaTva :Op1 = Val(Op1) / 100#

Case "sqrt"
   AfiseazaBazaTva
   If Op1 < 0 Then
      MsgBox "Nu pot sa Calculez Radical din Numere Negative", 48, "Calculator"
   Else
      Op1 = Sqr(Val(Op1))
   End If
   Readout = Format(Op1, "###,###,###,###,###.00")

Case "T.V.A."
   If Op1 = 0 Then
      MsgBox "De ce sa calculez Baza pentru valoarea TVA 0",48,Calculator"
   Else
     ValBaza = Val(Op1) :ValTva = ValBaza * (PrTVA / 100)
      BazaTVA.Caption = Str(Format(ValBaza, "###,###,###,###,###.00"))
      TvaAferent.Caption = Str(Format(ValTva, "###,###,###,###,###.00"))
      Readout = Str(Format(ValBaza + ValTva, "###,###,###,###,###.00"))
   End If

End Select
```

3.4 Repeating Structure

The repeating structure repeats a block of statements while a condition is True or Until a condition becomes True. The repetition of steps in a program is called a **loop**. The executions of such blocks follow the scenario (while; figure 3.15): the condition is evaluated and if the condition evaluates to:

• **True** then executes the block of statements;

• **False** then end the execution of the cycle (Loop) and continue the execution of the program.

If for the first time the condition is False the sentence block is simply skipped.

Conditional Loop with Condition Evaluated First

Syntax:

Do [{While|Until}*condition*]	→ beginning of cycle
[*statements*] [Exit Do] [*statements*]	→ body of the cycle
Loop	→ the end of sentence block

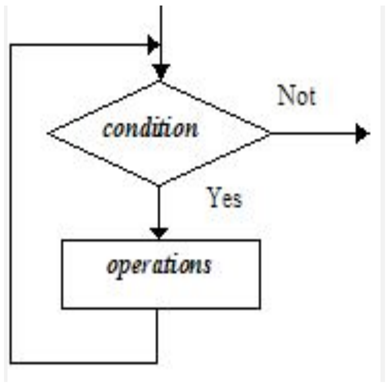


Figure 3. 15 Loop structure

The commands Loop and Exit Do are used to do:

- Loop – an unconditional jump (or branch) to the beginning of the associated cycle (the evaluation of the condition);

- Exit Do – an unconditional ending of the cycle (a jump to the next sentence defined under the loop that ends the body of the cycle).

Do…Until work as:

1) execute statements;
2) evaluate the condition Loop or Exit

The block of commands between Do… and Loop will be executed while/until the conditional expression "condition" evaluates to True.

Visual Basic (Access, VBA)	C++

Do [{**While**|**Until**}] *condition*
operations
[**Exit Do**]
[**Continue Do**]
Loop
 	
A set of commands (*operations*) placed between **Do While** and **Loop** is executed for as longs the logical condition remains True.
Exit Do – Loop statement immediately;
Continue Do – immediately jumps to the end of Do … Loop statement.
	- first syntax:
	
	while (*condition*) *operation*;
	
	- second syntax:
	
	while (*condition*)
	{
	operations;
	[**continue**;]
	[**break**;]
	}
	where:
	- **continue** jump to the condition evaluation;
	- **break** interrupt the cycle and transfer the execution to the sentence that follows to the end block marker }

PASCAL

First syntax:

while *condition* **do**
    operation;

Second syntax:

While *condition* **do**
    Begin
        operations;
        Exit
    End;

- **Exit** means an unconditional exit from loop.

If the repeating structure is placed in the body of a subroutine (function or procedure) we can exit from the cycle by using the returning to caller commands.

Examples:

I. Let be *a*, a positive real number. Define recursively the sequence of x_i of positive numbers as follow:

$x_0 = 1$

$x_{i+1} = (1/2)*(x_i + a/x_i)$ for i=0,1,2,...

Draw a flowchart (figure 3.16) that reads in the value interactively and uses this algorithm to compute the square root of a (it can shown mathematically that $x_i \rightarrow \sqrt{a} \;\; for \;\; i \rightarrow \infty$).

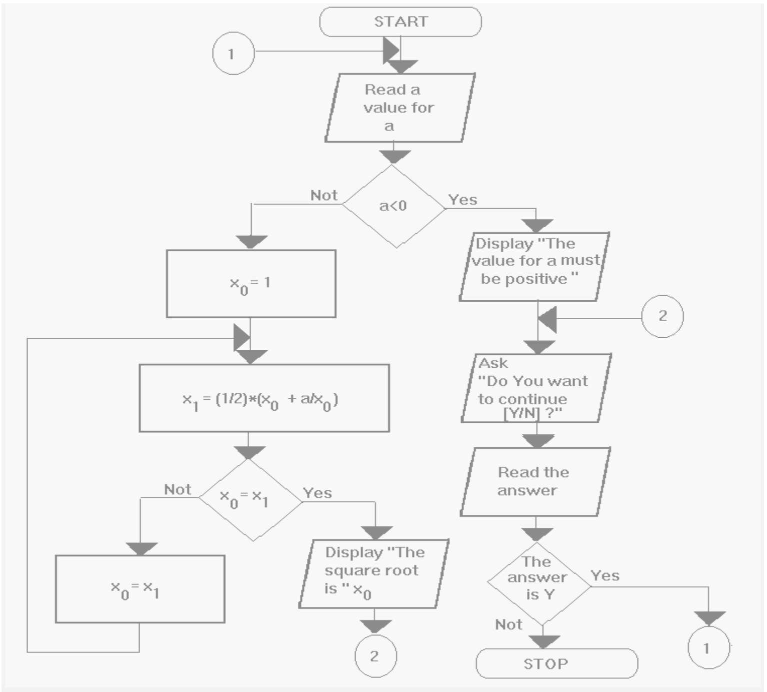


Figure 3. 16 A flowchart for the Newton-Raphson method

The algorithm stops when x_{i-1} equals with x_i-*precision*, where *precision* is a positive fraction low value (the desired precision), that mean to be satisfied the following condition $|x_i - x_{i-1}| \leq precision$. The condition can be described as: the difference from the current value (x_i) and the previous one (x_{i-1}) must be at least the desired precision (for example 0.001 if the precision must be at the third decimal).

This algorithm is derived from the so known Newton-Raphson method in numerical analysis. Figure 3.17 shows a form representing the interface for the implementation in Visual Studio 2005 of the algorithm depicted in figure 3.16.

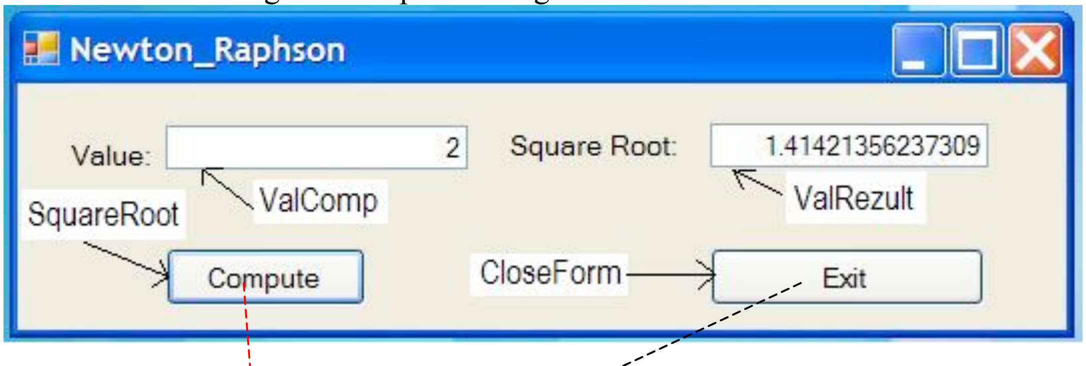


Figure 3.17 The form used as interface for the implementation of the algorithm

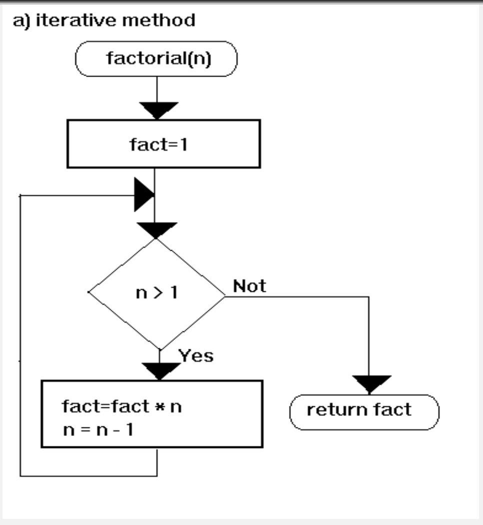
The arrows in the figure 3.17 indicate the name given to the object they point to.

```
Public Class Newton

    Private Sub CloseForm_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles CloseForm.Click
        Me.Close()
    End Sub

    Private Sub SquareRoot_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles SquareRoot.Click
' Memory variables reservation
        Dim x0 As Double, x1 As Double, a As Double
' Check if the value typed by the user in the text box labeled Value (ValComp.Txt) is not numeric and
' sends  a warning message to the user about that and returns to fix the mistake
        If IsNumeric(Trim(Me.ValComp.Text)) = False Then
            MsgBox("Type a valid value and repeat!", vbOKOnly, "Attention!")
        Else
' If the value is a number:
' 1. convert this number from his external representation in a real number
' 2. initialize the variables x0 and x1
           a = Val(Trim(Me.ValComp.Text))
            x0 = 1
            x1 = 0
' 3. starts an “infinite” loop
           Do While True = True
' 4. computes the values by applying the formula
                x1 = (1 / 2) * (x0 + a / x0)
' 5. exits from cycle when two consecutives computed values are equal
                If x0 = x1 Then Exit Do
                x0 = x1
            Loop
' 6. writes the obtained value in the form and exits.
            Me.ValRezult.Text = x0
        End If
        End Sub
End Class
```

II. For a nonnegative integer *n* the factorial of *n*, written *n!*, is defined by:

a) iterative mode	 a) iterative method
	factorial(n)
	fact=1
	n > 1
	Not
	Yes
	fact=fact * n
	n = n - 1
	return fact

0! = 1 *n!=n*(n-1)(n-2)*…*3*2*1* for *n>0*	

Function Factorial_Iterative(n As Integer) As Double
  Dim fact As Double
   fact = 1 ' Initialize to 1 for product
  Do While n>0
    fact = fact * n
    n=n-1 ‘Next cycle
  Loop
  ‘ The returned value
  Factorial_Iterative = fact
End Function	

Conditional Loop with Condition Evaluated After

In this case the operation is executed first and then the condition is evaluated (figure 3.18).

It can be described as:
- the *operations* are executed;
- the *condition* is evaluated;
- **if** the result of the evaluation of the *condition* is False *then loop* to execute again the *operations*;
- **if** the evaluation of the *condition* is True *then* continue the execution of the program (and close the loop).

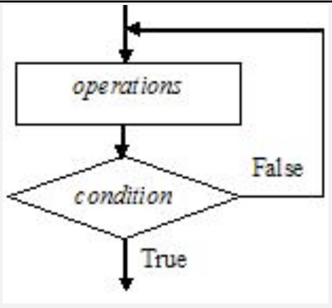


Figure 3. 18 Do … Until

The syntax for the sentences that implements that structure are:

Basic (Access, VBA)	**C++**

Do
    operations
Loop {While|Until} *condition*
	do
	  {
	    operations;
	  } **while** *conditions*

PASCAL	

repeat
  operations;
until *condition*	

Counter Loop. Executes a set of statements (operation$_1$) within a loop a specified number of times (figure 3.19).

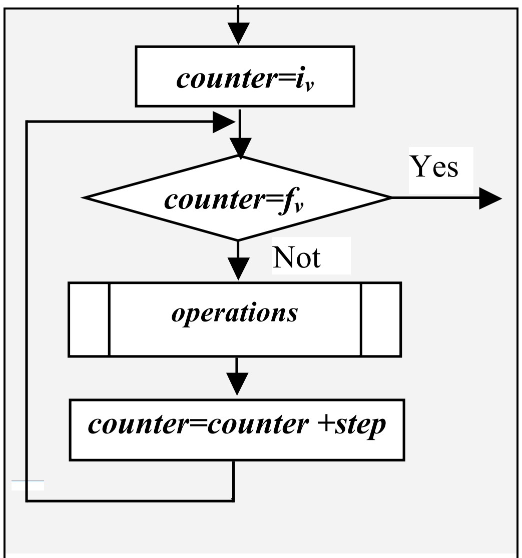


Figure 3. 19 The representation of For …

A variable is used as *counter* to specify how many times the statements inside the loop are executed.

In this diagram:

- i_v - is the initial value (start value - usually 0 or 1);

- f_v - is the end value (the expected value - usually how many times);

- *step* - is the increasing (or decreasing) step for counter.

The pseudocode for this structure is:

Basic (Access, VBA)	C++
For *counter* = v_i **To** f_v [**Step** *s*] *operations* [Exit For] [Continue For] **Next** [*counter*] Where: - the value for *s* can be positive or negative; - if *s* is positive then must have the inequality v_i<f_v (otherwise the cycle never executes); - if *s* is negative then must have the inequality v_i>=f_v (otherwise the cycle never executes); - if a value for Step not specified the default is used (+1); - the cycle can be stopped unconditionally by intermediate of the sentence **Exit For** or can be reiterated by the sentence **Continue For** (goes to the end of For)	**for** (*expression$_1$*; *expression$_2$*; *expression$_3$*) *operation*; If many operations desired in the cycle they must be included as block; *expression$_1$* – is an expression that initialize the counter; *expression$_2$* – contains the definition for ending the loop; *expression$_3$* – is an expression to increment or decrement the value for the counter. The cycle can be unconditionally stopped by using the instruction **break** and can be unconditionally restarted by using the sentence **continue.**

PASCAL

First syntax:
FOR *counter* := v_i **TO** f_v **DO** *operation;* [normal]

Second syntax:
FOR *counter*:=v_i **DOWNTO** f_v **DO** *operation;* [in reverse order]

If many operations desired they must be placed in a **begin ... end** (block).

The execution of For (VB) sentence follows the scenario:

1. The value v_i is assigned to the variable *counter*;
2. The value of variable *counter* is compared with the end value f_v (If the value for step is negative is checked if *counter*<f_v);
3. The operations are executed;
4. The value of variable counter is incremented with the value step (1 if step not specified);
5. Repeat the steps from 2 to 5.

The interpretation of the elements of For…Next sentence is:

what cycle number is ↓ how many times ↓

For *counter*=*start_value* To *end_value* [Step *increment_decrement*]
    [*statements*]
    [Exit For] ← stop the cycle
    [*statements*]
    Next [*counter*]

Examples:

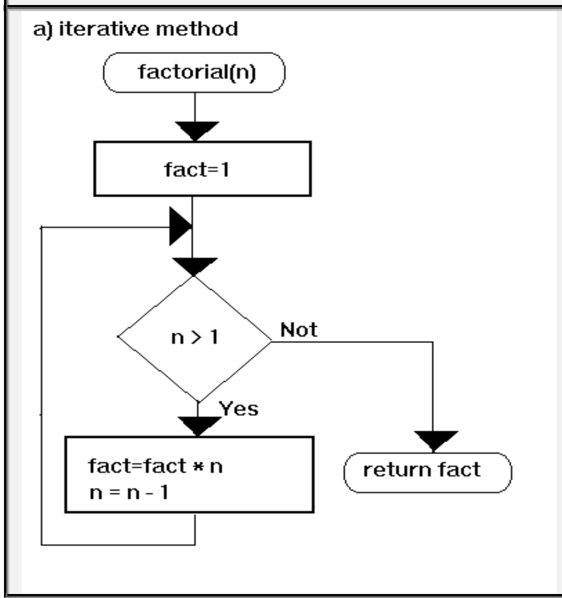
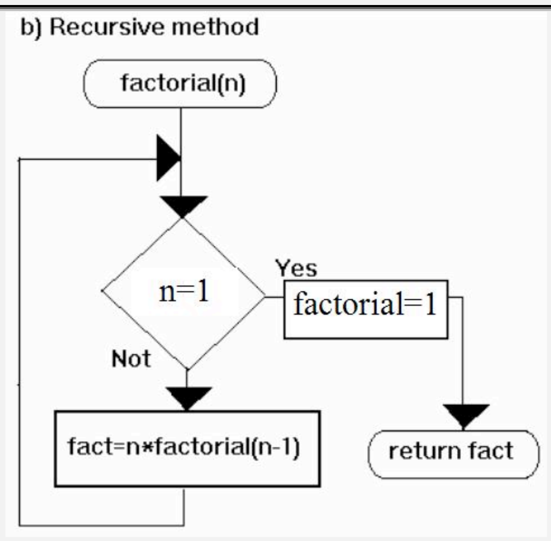
I. Pope pleases the great mathematician Gauss to tell him when Eastern will be in a wanted year. Gauss says that Eastern will be always on: 4 April + **D** days + **E** days where:
D is determined by following the steps:
1 – the year is divided by 19;
2 – the remainder is multiplied by 19;
3 – to the result of step two add fix factor 15;
4 – the sum of values obtained in the steps 1 to 3 is divided to 30 and the remainder is D
E is determined by following the steps:
1 – the year is divided by 4 and the remainder will be multiplied by 2
2 – the year is divided by 4 and the remainder will be multiplied by 4
3 – compute the sum of values obtained to step 1 and 2
4 – to the sum add 6*D and to product add 6
5 – the total sum is divided by 7 and the remainder will be E

A code that implements this algorithm is:

```
Sub Date_Pasti()
   Dim An_Inceput, An_Sfarsit As Variant, WData As Date
   Dim rasp As Byte, i As Integer
   An_Inceput = InputBox("Anul de la care " & Chr(10) & Chr(13) & "calculam data Pastelui:", "Calculul datei Pastelui")
   An_Sfarsit = InputBox("Anul pana la care " & Chr(10) & Chr(13) & "calculam data Pastelui:", "Calculul datei Pastelui")
   For i = An_Inceput To An_Sfarsit + 1
      WData = Data_Paste(i)
      xExemple.Print i, "  ", WData
   Next i
   ras = MsgBox("Pastele cade in anul " & An_Dorit & " la " & Data_Paste, vbOKOnly, "Calcul data Pastelui")
End Sub
Function Data_Paste(An_Dorit As Variant) As Variant
   Dim D As Integer, E As Integer
   D = ((An_Dorit Mod 19) * 19 + 15) Mod 30
   E = (((An_Dorit Mod 4) * 2 + (An_Dorit Mod 7) * 4) + 6 * D + 6) Mod 7
   Data_Paste = DateAdd("d", D + E, CDate("04/04/" & Trim(An_Dorit)))
End Sub
```

II. For a nonnegative integer *n* the factorial of *n*, written *n!*, is defined by:

a) iterative mode	**b) recursive mode**
$0! = 1$ $n! = n*(n-1)(n-2)*\ldots*3*2*1$ for $n>0$	$0! = 1$ $n! = n((n-1)!)$ for $n>0$

 a) iterative method
factorial(n)
fact=1
n > 1
Not
Yes
fact=fact * n
n = n - 1
return fact
	 b) Recursive method
	factorial(n)
	n=1
	Yes
	factorial=1
	Not
	fact=n*factorial(n-1)
	return fact

Function Factorial_Iterative(n As Integer) As Double
  Dim fact As Double
  fact = 1
  For i = 1 To n
    fact = fact * i
  Next i
  Factorial_Iterative = fact
End Function
	Function Factorial(n As Integer) As Double
	  If n = 1 Then
	      Factorial = 1
	   Else
	      Factorial = n * Factorial(n - 1)
	  End If
	End Function

The modern object-oriented programming languages offers a for sentence with the syntax **For Each ... Next** that allow to apply a set of sentences to an object collection or to a multitude (arrays, vectors, multidimensional massive) without specifying the number of cycles (that specification is difficult if the dynamic memory reservation used).

The syntax of that sentence is:

```
For Each element In group
        Sentences
Next element
```

Example:

```
For Each obj In ctl.Tabs
        obj.Caption = LoadResString(CInt(obj.Tag))
        obj.ToolTipText = LoadResString(CInt(obj.ToolTipText))
Next
```

Loops can be nested just as If statements are. Nested loops consist of an outer loop with one or more inner loop. Each time the outer loop is repeated, the inner loops are reentered, their loop-control expressions are reevaluated, and all required iterations are performed.

The **For** instructions can be nested following the model:

```
For i...
        sentences
        For j ...
                sentences
                ...
                For k ...
                        sentences
                Next k
                …
        Next j
        …
Next i
```

Each **For i** associated with the line containing the **Next** sentence followed by the variable designated as counter.

If the instruction line **Next** don't contains the counter variable name then that associates to the first appearing sentence **For**.

If the line sentence **Next** contains the counter variable then the association is realized in the reverse order of **For's** instructions. In our case, the first cycle that must be enclosed is that defined in the line containing the sentence **For k**, the second instruction **Next** closes the line containing the instruction **For j** and the last closes the line containing **For i**. In the case in which between the sentences Next enclosing For sentences no other sentences needed then is enough to place a single Next sentence referencing, by intermediate of a list, the counter variables of that For's.

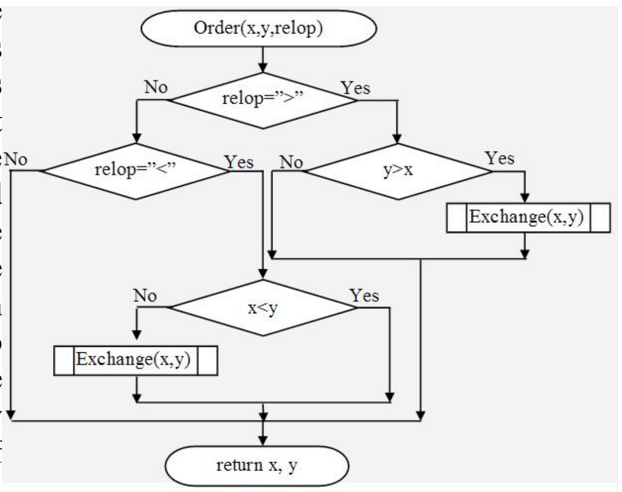


Figure 3.20 An algorithm for ordering two values

Example:

```
For i...
        For j...
                sentences
Next j , i
```

3.5 Commented Samples

I. We realize a generalization of the *Asc* algorithm so that the values (x and y) are passed together with the comparison operator (*relop*) that can be '>', '<' and '='. Depending on the value chosen for the comparison (*relop*) we are the case greater than, less than and otherwise is the equality. The algorithm call the defined algorithm *Exchange(x,y)* to realize the inter exchange (permutation) of the values, assigned at execution time (run time) to the variables x and y, in the sense defined by the used relational operator (comparison). The algorithm returns the values in the specified order (figure 3.20).

```
Sub Order(x As Variant, y As Variant, relop As Variant)
  Select Case relop
        Case "<"
          If x > y Then
            Exchange x, y
          End If
        Case ">"
          If x < y Then
            Exchange x, y
          End If
  End Select
End Sub
```

This code sequence is a possible implementation of the algorithm explained in the flowchart from figure 3.20. The declaration of variables as Variant allows using the subroutine *Order()* to do his action on any kind of data type of the values passed in arguments (text, number, date, etc). The routine can be generalized to receive at input an array, vector or multidimensional massive, or can be called as such from routines that processes such arrays by passing in arguments pairs of values.

II. Given a set of *n* values *v[i]*, i=1,2,3, …, n.

2.1 Build the function that compute the *sum*[*Sum(v,n)*] of this values (a).

(a) The function Sum adds a series *v* of *n* numbers

```
Function Sum(v As Variant, n As Integer) As Double
  Sum = 0 ' The initial value
  For i = 1 To n
    Sum = Sum + v(i)
  Next i
End Function
```

2.2 Build the function that compute the average [A*vg(v,n)*] of these values (b and b').

(b) The function Avg computes the means of a series of numbers

```
Function Avg(v As Variant, n As Integer) As Double
  Sum = 0
  For i = 1 To n
    Sum = Sum + v(i)
  Next i
  Avg=Sum/n
End Function
```

(b') The function Avg computes the mean of a series of numbers and uses the function Sum defined at (a) point

```
Function Avg(v As Variant, n As Integer) As Double
    Avg=Sum(v,n)/n
End Function
```

2.3 Draw a flowchart that exponentiation a number *x* at the power *n* [*Power(x,n)*].

Dissection of the power flowchart (figure 3.21):

- the Start symbol contains the name of algorithm and the call format. In the brackets is indicated the list of arguments to be passed at call time;
- by *i=1* we initialize the counter for the exponent, and we initialize the result to 1 (if *n* is *0* then $x^0=1$);

- by *pow=pow*x* we compute the actual value of the variable *pow* as a multiplication of the previous value of the variable *pow* with the variable *x*;
- in the decision block we check how many times we are realized the repeatedly multiplication. If the multiplication is not realized *n* times we increase the counter and then go to a new multiplication else the algorithm return the result.

A possible implementation of the algorithm is one of the following:

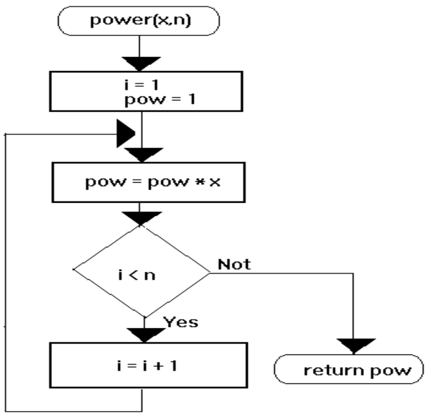


Figure 3. 21 An exponentiation algorithm

(a)

```
' The function Power exponentiation a number x to a power n
Function Power(x As Variant, n As Integer) As Double
  Dim pow As Double
  pow = 1
  For i = 1 To n
        pow = pow * x
  Next i
  Power = pow
End Function
```

(b)

```
' The function Power exponentiation a number x to a power n - compact
Function Power(x As Variant, n As Integer) As Double
  Power = 1
  For i = 1 To n
        Power = Power * x
  Next i
End Function
```

III. We want to solve now a more complex problem: we want to list a Fahrenheit to Celsius correspondence table based on the computation formula:

$CELSIUS^o = (5/9)*(FAHRENHEIT^o - 32)$

The correspondence table will be displayed (figure 3.22) starting with the minimal value (**min**) 0 (zero) and ending with the maximal value (**max**) of 300 degrees and the computation and display will be done from 20 to 20 degrees (**pas**). We use, to solve this problem, assignments instructions, the function MsgBox to display the result and an instruction For that allow us to repeat the execution of a group of sentences until a specified condition satisfied.

The program looks as:

```
0       Sub Coresp_Temp()
1         Dim min, max, pas, fahrenheit#, celsius, tabel
2         ' Computation of the correspondence C°- F°
3         min = 0 ' Starting Value
4         max = 300 ' Ending Value
5         pas = 20 ' From … to … degrees
6         tabel = "Fahrenheit  | Celsius  " & Chr(13) & Chr(10) & _
               String(36, "-") & Chr(13) & Chr(10)
7         For fahrenheit = min To max Step pas
8            celsius = (5 / 9) * (fahrenheit - 32)
9            tabel = tabel & Right(Space(12) & Format(fahrenheit, "#000"), 12) &_
 "    " & Right(Space(12) & Format(celsius, "#000.00"), 12) & Chr(13) & Chr(10)
10        Next fahrenheit
11         MsgBox tabel, , "Conversion Fahrenheit-Celsius"
      End Sub
```

Comments

Line 0 is that in which the type of used procedure and scope declared (public subroutine in our case) together with the name that can be used later on to call the procedure (Coresp_Temp);

> Notes ! The numbers associated to the lines can be typed as such in the source program (they keep from previous versions of Basic). The lines that follows after the line numbered 6 and the line numbered 9 are not numbered because they continuation lines of previous sentence.

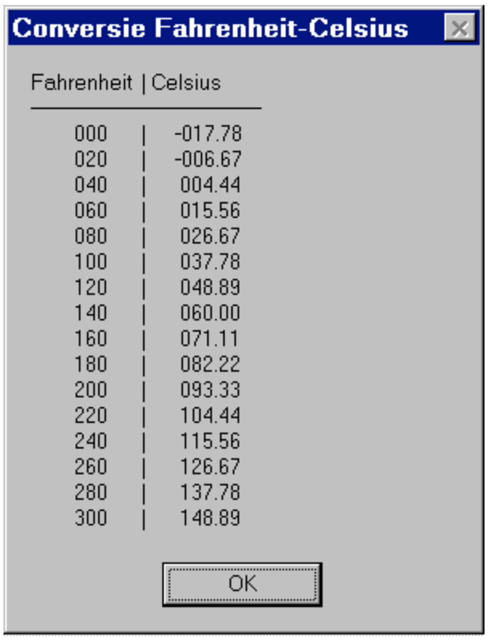
A continuation is specified by placing an _ (underscore) character to the right of the line. The writing of many instructions on the same line is also allowed by placing a : (colon) character before each instruction.

Line 1, Dim *min, max, pas, fahrenheit#, celsius, tabel* declares and reserve the needed memory space. In that line the variables min, max, pas, celsius and tabel, that haven't a data type specification, are of Variant data type (without a specific data type but adapted to those of the first expression in which is used) and *fahrenheit#* is a double precision variable;

Line 2 is a comment line introduced by an ' (single apostrophe); his role is to explain the role played by the routine;

Lines from 3 to 5 are assignment statements. What is in the left of the = (called assignment operator) symbol are variable names (*min, max, pas*). What is in the right represents a constant and/or expression; in that case they are numbers and are called numeric constants. On these lines we define two instructions: one assignment and the second (delimited from the first by ') a comment.

Line 6 is an assignment line that initialize the variable named *tabel* with a character string, formed by two lines, string obtained by evaluating the string expression "Fahrenheit | Celsius " & **Chr**(13) & **Chr**(10) & **String**(36, "-") & **Chr**(13) & **Chr**(10). In that expressions appears string constants (enclosed between “) concatenated (by intermediate of & operator) with the result of evaluation of string (character, text) manipulation functions (as **Chr**, **String**). The character _ (underscore) appearing on line 6 instruct the compiler (or interpreter) that the command line is continued to the line that follows;



```
Conversie Fahrenheit-Celsius

Fahrenheit | Celsius
--------------------
  000   |  -017.78
  020   |  -006.67
  040   |   004.44
  060   |   015.56
  080   |   026.67
  100   |   037.78
  120   |   048.89
  140   |   060.00
  160   |   071.11
  180   |   082.22
  200   |   093.33
  220   |   104.44
  240   |   115.56
  260   |   126.67
  280   |   137.78
  300   |   148.89

           OK
```

Figure 3. 22 The display of Fahrenheit-Celsius correspondence table

Line 7 it introduces a processing cycle that expressed in a natural language as: **For** the variable named *fahrenheit* starting with the value *min* (*fahrenheit=min*) **To** value *max***,** with a **Step** equal to *s* (from step by step) **execute** the instructions from the lines that follows until the program line containing the sentence to repeat (*Next fahrenheit*, line 10). The execution ends when the value of *fahrenheit* is greater than *max*. To each executed cycle, the value of *s* is added to the variable *fahrenheit*.

Line 8 transforms the Fahrenheit degrees in Celsius degrees;

Line 9 transform the computed values in a character string by using string functions and complete a table line.

IV. We want to keep the marks obtained by a student over 5 years of studies in tables of the form:

Student First name & Last name

Studies Year	**Obtained Marks**				
	Discipline 1	**Discipline 2**	**…**	**Discipline 10**
I				
II				
III				
IV				
V				

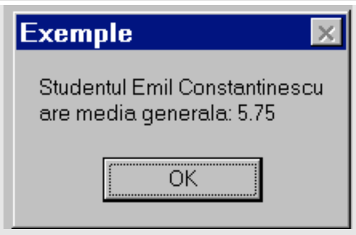
In the table below, on the rows numbered from I to V, the studies years are enumerated, and on columns, we have ten disciplines, representing the disciplines from each year.

By the notation Mark 1,5 we define the mark obtained in the first year to the discipline 5. This table can be represented in computer as a bi-dimensional array or matrix. If we denote by **i** and **j** two variables by which we designate a line (**i**), and respectively a column (**j**) from the table then by Mark i, j we designate the mark obtained in the year **i** to the discipline **j**.
The computation formula (or the algorithm) used to determine the average is:

$$Average = \sum_{i=1}^{5}\sum_{j=1}^{10} Mark_{i,j} / (5*10)$$

The sum of all marks from the table is computed by adding the value *Mark*$_{i,j}$ to the sum obtained by adding the previous values (by following the way: after each adding operation we obtain a subtotal to which the next value is added).
The computation program can be:

```
Sub Average_Marks()
    Dim Mark(5, 11), i As Integer, j As Integer: Rem the definition of
the table in the internal memory
    Dim Average As Double, Name As Variant
    Read_Marks Name, Mark
    Average = 0
    For i = 1 To 5
        For j = 1 To 10
            Average = Average + Average(i, j)
        Next j
    Next i
    Average = Average / (10 * 5)
    MsgBox "Studentul " & Name & Chr(13) & Chr(10) & "are media generala: " & Average
End Sub
```



```
Studentul Emil Constantinescu
are media generala: 5.75
```

An output sample

In the table Mark, we must fill, in each cell (called element) the obtained Mark. We fill the marks by reading them from a file where previously stored or from the keyboard. The table defined by the sentence Dim Mark(5,10) is called usually matrix, bidimensional array or bidimensional massive.

In this example the procedure Read_Marks is defined as follows:

```
Sub Read_Marks(Name As Variant, Mark As Variant)
   Name = InputBox("First Name & Last Name:", "Example of using VB")
   For i = 1 To 5
      For j = 1 To 10
         Mark(i, j) = InputBox("Name:" & Name & Chr(13) & Chr(10) & _
          "Mark: (" & i & " , " & j & " ) ", "Example of using VB: Marks Input")
      Next j
   Next i
End Sub
```

In this example we suppose that the user types numbers between 0 (zero) and 10 (ten) as integer values. Because the function InputBox() reads text values the mistakes (any other characters than digits and/or spaces between digits) will produces a computation error message. To avoid that is necessary to test whether the typed value is number or not.

```
' Read_Number reads a value from keyboard and verifies
' if number or not (a process called validation). If the value is not a number
' signals that to the user who can choose between cancel or resume the operation from typing
```

```
Function Read_Number(xNr As Variant, denNr As Variant) As Variant

 Dim Answer As String
 Do While True = True ' an infinite cycle
   xNr = InputBox("Type the value for " & denNr & ":", "Example")
   If Not (IsNumeric(Trim(xNr))) Then
     Answer = MsgBox("The Value for " & denNr & " must be Numerical !", vbOKCancel,  _
                     "Example") ' This is a continuation line
      If Answer = 2 Then ' Cancel Button pressed
         Read_Number = "*Cancel" ' The returned value to the caller is *Cancel
         Exit Do ` Exit from the infinite cycle and return to the caller
      End If
    Else
      Read_Number = Trim(xNr) ' The returned value will be the number
                              ' without extra spaces (to the left or right)
      Exit Do ' Exit from the infinite cycle and return to the caller
   End If
 Loop ' Restart the cycle
End Function
```

V. The following example illustrates how the arithmetic operators and intrinsic functions used:

```
1    Function Data_Paste(An_Dorit As Variant) As Date
2       Dim D As Integer, E As Integer
3       D = ((An_Dorit Mod 19) * 19 + 15) Mod 30
4       E = (((An_Dorit Mod 4) * 2 + (An_Dorit Mod 7) * 4) + 6 * D + 6) Mod 7
5       Data_Paste = DateAdd("d", D + E, CDate("04/04/" & Trim(An_Dorit)))
6    End Sub
```

Line 1. Declares the name of the function (Data_paste – Easter date) and the input argument required (An_Dorit As Variant – Wanted year) and returns a Datetime type value;

Line 2. Working variables D and E declarations as integers (represent years);

Line 3 and 4. Compute the value for D and, respectively E, as described in the algorithm introduced earlier. In both formulas are exploited the parenthesis evaluation rule and operator precedence rule. In the formula D=((An_Dorit Mod 19) * 19 + 15) Mod 30, first is computed the modulus 19 of An_Dorit and second this value is multiplied by 19; in the third step to the previous value is added the constant 15; in the fourth step the modulus 30 is obtained from the last value and then assigned to D;

Line 5. In this line a string date value is obtained as a concatenation of strings An_Dorit (Variant value) with 4 April ("04/04/") from which a date value is obtained as a conversion (CDate), and to that obtained date is added the number of days ("d") in terms of date (DateAdd – this function allow to add days, month, years to a date and to obtain a correct date; it take account of the number of days in which month and if a year is leap year or not). The function requires three parameters separated by , (comma) character, with the general format DateAdd(*string$_1$,integer,string$_2$*). The first parameter of that function (*string$_1$*)is the unit of the added value ("d" still for days) and is given as a string literal value (a value enclosed in quotation marks). The second parameter (*integer*) gives after how may time units from the initial date (*string$_2$*) the new date. This parameter is given as an expression (D+E), where both D and E are integers as defined in line 2 and as computed in lines 3 and 4, respectively. The result of adding these two integers will be an integer too, as required by the function call model. The last required parameter (*string$_2$*) represents the beginning date (April 4 of wanted year – An_Dorit) and is given as a string concatenation expression formed by the literal value "04/04/" to which is concatenated (by the concatenation operator &) the desired year Trim(An_Dorit). The An_Dorit variable is of Variant data type and the Trim functions eliminates the trailing blanks may be to the left or to the right of the value. By using this function we prevent the case the user type a year in the text box of the dialog box preceded or followed by space characters (but we don't prevent the spaces between digits; if you want do that use the function ReadNumber from example IV). The value obtained is assigned to the name of the function (Data_Paste=…) as the value to be returned to the caller;

Line 6. End the function execution and returns the processing control to the caller.

VI. The following procedures implement the division conversion method for decimal numbers to any base that is a power of 2.

```
‘ A cell rezervation and initialization with the the string of hexadecimal system

  Dim myDigits As String = "0123456789abcdef"

‘ Ask the user to type the values for number and base in a dialog box. Each value typed by user is
‘ checked to be numeric. If not numbers a message is given to the user and execution stops. The
‘ function dec2base(decimal-number,new-base)

  Sub Dec2IntAnyBase()
    Dim b As String
    Dim d As Integer, bb As Integer
    d = InputBox("Type an integer number.")
    If IsNumeric(Trim(d)) = False Then
      MsgBox("Type an integer value!", vbCritical + vbOKOnly)
      Exit Sub
    End If
    b = InputBox("Type an integer number for base.")
    If IsNumeric(Trim(b)) = False Then
      MsgBox("Type an integer value!", vbCritical + vbOKOnly)
      Exit Sub
    End If
    bb = Int(Val(Trim(b)))
    b = dec2base(d, bb)
    If b <> "*Err" Then
      MsgBox(d.ToString & " in base " & bb.ToString & " is " & b)
    End If
  End Sub
```

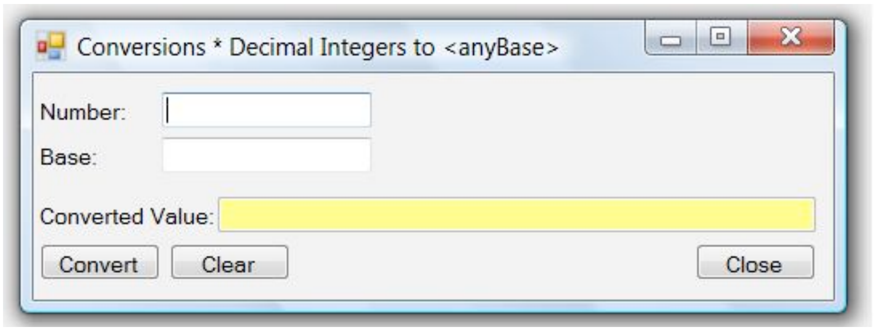
```
‘ Receives at call time a copy of the values for number and base and determines if the wanted base is
‘ a power of 2 (two) and which one. If not a power of two or the base minus one is zero signals that
“ to the user and stops processing.
‘ Extracts the digit in the new base from the string myDigits from the position determined by
‘ anding the number with the base-1
‘ Starts a loop as long as the number not yet less than the most significant digit in the new base
‘ Divides the number by the new base reprezented as 2 at the power determined previously here
‘ Extracts the digit in the new base from the string myDigits from the position determined by
‘ anding the number with the base-1
‘ Restarts the loop
‘ Returns the value to the caller

  Function dec2base(ByVal d, ByVal b) As String
    Dim newD As String, bb As Integer, b1 As Integer
    bb = whichPowerOf2(b)
    b1 = b - 1
    If bb <= 0 Or b1 = 0 Then
      dec2base = "*Err"
      Exit Function
    End If
    newD = Mid(myDigits, (d And b1) + 1, 1)
    Do While (d > b1)
      d >>= bb
      newD = Mid(myDigits, (d And b1) + 1, 1) & newD
    Loop
    dec2base = newD
  End Function

‘ This function determines if a number is a power of two or not.
‘ If the number is a power of two determines that power and returns to caller that value

  Function whichPowerOf2(ByVal d) As Integer
    Dim nr As Integer = 0, dd As Integer
    dd = d
    If d < 2 Then
      Return 0
    End If
    If (d - 2 * (d >> 1)) <> 0 Then
      MsgBox("The base is not a power of two!", vbCritical + vbOKOnly)
      whichPowerOf2 = 0
      Exit Function
    End If
    Do While d > 1
      d >>= 1
      nr += 1
    Loop
    If dd <> 2 ^ nr Then
      MsgBox("The base is not a power of two!", vbCritical + vbOKOnly)
      whichPowerOf2 = 0
      Exit Function
    End If
    Return nr
  End Function
```

VII. The conversion algorithm implemented as a Class. The algorithm for conversion implemented with basic arithmetic operators (+,-,*,/). The clarity of the code have a greater priority than the optimality and shortest of the code, in this example (and also in most of examples introduced here).



When the button Convert is pressed the event procedure convertButton_Click is called that realizes the following operations:

1. Rezervation of memory cells used to store the number and new base:

```
Dim Ni As Long, b As Long
```

2. Check if what typed by user in the text box Number is a number. If not a number signals that by a message and returns in the form to retype the number or end processing:

```
If Trim(Me.decNumber.Text) = "" Or IsNumeric(Trim(Me.decNumber.Text)) = False Then
    MsgBox("Err. The value you want convert is not a number !", vbCritical + vbOKOnly)
    Me.decNumber.Focus()
    Exit Sub
End If
```

3. Check if what typed by user in the text box Base is a number. If not a number signals that by a message and returns in the form to retype the number or end processing:

```
If Trim(Me.newBase.Text) = "" Or IsNumeric(Trim(Me.newBase.Text)) = False Then
    MsgBox("Err. The value for new base is not a number !", vbCritical + vbOKOnly)
    Me.newBase.Focus()
    Exit Sub
End If
```

4. Converts the strings of digits for number and base in integer values:

```
Ni = Trim(Me.decNumber.Text)
b = Int(Val(Trim(Me.newBase.Text)))
```

   The Val() function convert a string in a real number. If the string is empty or space the converted value is 0.

   Int() convert the base in an integer number and ensure that the base is integer. All these operations must be realized because the user types in a text box that accepts any combination of characters.

5. If the new base is 0 then an error message displayed and the control is passed to the user to fill in the form a valid value or end:

```
    If b = 0 Then
       MsgBox("Err. Actually according to all algebric theory divison by 0 (zero) not supported !" & _
       Chr(10) & Chr(13) & "A fractionary number is converted to integer!", vbCritical + vbOKOnly)
       Me.newBase.Focus()
       Exit Sub
    End If
```

6. Calls the function to convert the number in the new base IntegerToBinary(Ni, b), and stores the value in form's box Converted Value:

```
Me.convertedValue.Text = IntegerToBinary(Ni, b)
```

```
Public Class Convesion
   Private Sub convertButton_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
   Handles convertButton.Click
      Dim Ni As Long, b As Long
      If Trim(Me.decNumber.Text) = "" Or IsNumeric(Trim(Me.decNumber.Text)) = False Then
         MsgBox("Err. The value you want convert is not a number !", vbCritical + vbOKOnly)
         Me.decNumber.Focus()
         Exit Sub
      End If
      If Trim(Me.newBase.Text) = "" Or IsNumeric(Trim(Me.newBase.Text)) = False Then
         MsgBox("Err. The value for new base is not a number !", vbCritical + vbOKOnly)
         Me.newBase.Focus()
         Exit Sub
      End If
      Ni = Trim(Me.decNumber.Text)
      b = Int(Val(Trim(Me.newBase.Text)))
      If b = 0 Then
         MsgBox("Err. Actually according to all algebric theory divison by 0 (zero) not supported !" & _
         Chr(10) & Chr(13) & "A fractionary number is converted to integer!", vbCritical + vbOKOnly)
         Me.newBase.Focus()
         Exit Sub
      End If
      Me.convertedValue.Text = IntegerToBinary(Ni, b)
   End Sub

' This is the code associated to the Close button
   Private Sub closeButton_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
   Handles closeButton.Click
      Me.Close()
   End Sub

'This is the code associated to the Clear button (emptyies the text boxes)
   Private Sub cleanBoxes_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
   Handles cleanBoxes.Click
      Me.decNumber.Text = ""
      Me.newBase.Text = ""
      Me.convertedValue.Text = ""
   End Sub
```

The function *IntegerToBinary()* receives as arguments a copy of the number to be converted and the new base and returns a string containing the number in the new base. If the base is less than or equal to 16 it uses the hexadecimal digits to express the new number othewise the digits in the new base are written by using the decimal digits and separated by dots. The function realizes the operations:

1. Reservation of memory cells required by the algorithm:
```
Dim newNi As String, oldNi As Integer
```

2. The memory cell tht will store the number in the new base is erased:
```
newNi = ""
```

3. A cycle is started while the number to be converted not 0 (Ni>=1)

4. The value of Ni preserved in the variable oldNi and a new value for the number is computed as the integral part of the division of the number Ni to the new base b:
```
Ni = Int(Ni / b)
```

5. Check if the base is grather than 16 and if it is add the digit in the new base by concatenating the string of those digits obtained in the previous steps to "." and the fraction part of the previous step division (in reverse order); if the base less than or equal to 16 concatenates the digit supplied by the function isTheDigit():
```
If (b) > 16 Then
   newNi = (oldNi - (Ni * b)) & IIf(newNi <> "", ".", "") & newNi
Else
   newNi = isTheDigit((oldNi - (Ni * b))) & newNi
End If
```

6. Go to a new cycle (Loop - step 3).

7. If the convesion ends then the value for the number in the new base is assigned to the function name and returned to the caller of that function:
```
IntegerToBinary = newNi
```

```
Function IntegerToBinary(ByVal Ni, ByVal b) As String
   Dim newNi As String, oldNi As Integer
   newNi = ""
   Do While (Ni >= 1)
      oldNi = Ni
      Ni = Int(Ni / b)
      If (b) > 16 Then
         newNi = (oldNi - (Ni * b)) & IIf(newNi <> "", ".", "") & newNi
      Else
         newNi = isTheDigit((oldNi - (Ni * b))) & newNi
      End If
   Loop
   IntegerToBinary = newNi
End Function
```

This function is very simple: it defines a vector of 16 characters and initialize each one with the corresponding hexadecimal digits represented as characters. The function receives a value computed by following the conversion algorithm steps and returns a character string containing the digit in the new base (for example for 10 returns “a”). An optimized (a little strange!), method can be defined by using the bitwise operators (you can see an implementation in VB in example VI and in the site http://www.avrams.ro/conversions.htm).

```
    Function isTheDigit(ByVal i) As String
        Dim dg(0 To 15) As String
        dg(0) = "0"
        dg(1) = "1"
        dg(2) = "2"
        dg(3) = "3"
        dg(4) = "4"
        dg(5) = "5"
        dg(6) = "6"
        dg(7) = "7"
        dg(8) = "8"
        dg(9) = "9"
        dg(10) = "a"
        dg(11) = "b"
        dg(12) = "c"
        dg(13) = "d"
        dg(14) = "e"
        dg(15) = "f"
        isTheDigit = dg(i)
    End Function
End Class
```