Chapter 2 : Simple sequential algorithm

Dr. Abderrahmane Kefali

Senior Lecturer Class A,
Department of Computer Science,
University of May 8, 1945 - Guelma

kefali.abderrahmane@univ-guelma.dz

This document is prepared for printing two pages per sheet

1) Language and Algorithmic Language

Natural languages are inherently ambiguous, so it is essential to write algorithms in a formal language with precisely defined semantics to avoid any ambiguity. This specialized language is called an ***algorithmic language*** or ***pseudo-language*** or ***pseudocode***.

1.1) What is an Algorithmic Language?

An algorithmic language or pseudocode is a language that is close to natural language and, at the same time, takes into account machine characteristics while being more flexible than a programming language. It is used for describing algorithms.

This language uses a set of keywords and structures to fully and clearly describe the objects manipulated by the algorithm and all the instructions to be executed on these objects to solve a problem.

The pseudocode expresses instructions for solving a given problem independently of the specifics of a particular programming language. Therefore, algorithms written in algorithmic languages have the advantage of being easily translatable into a programming language.

1.2) Elements of an Algorithmic Language

An algorithmic language (like any other programming language) is defined by a set of words constituting its vocabulary, called ***keywords*** or ***reserved words***, in addition to rules of syntax and grammar governing the assembly of these words.

1.2.1) Keywords

Keywords are predefined and reserved words used in algorithms that have a particular meaning (**`ALGORITHM`**, **`BEGIN`**, **`END`**, **`IF`**, **`THEN`**, ... etc.).

2) Parts of an Algorithm

A pseudocode algorithm consists of three essential parts: the algorithm header, the declaration part, and the algorithm's body.

2.1) Algorithm Header

The algorithm header's sole purpose is to identify the algorithm by specifying a name for it. The header starts with the keyword **`ALGORITHM`** followed by the algorithm's name.

Examples:

```
ALGORITHM Sum;
```

```
ALGORITHM Calculation;
```

2.2) Declaration Part

The declaration part includes declarations of all data elements used in the algorithm's body. Declaration involves naming various objects, specifying their type, dimensions, etc.

2.3) Algorithm Body

The algorithm's body includes all instructions and operations to be performed on the data to solve the problem. These instructions involve basic computer operations.

The processing part begins with the keyword **`BEGIN`** and ends with the keyword **`END.`** indicating the end of the algorithm.

All instructions must end with a semicolon "`;`" which serves as a separator between instructions.

3) Data: Variables and Constants

In an algorithm, you will frequently need to temporarily store objects on which all the algorithm's processing will be based. These objects can come from the hard drive or be provided by the user (keyboard input). They can also be results obtained by the algorithm, whether intermediate or final. Whenever you need to store information during an algorithm, you use a variable or a constant. These can be of different types: text, numeric, logical, etc.

3.1) Variables

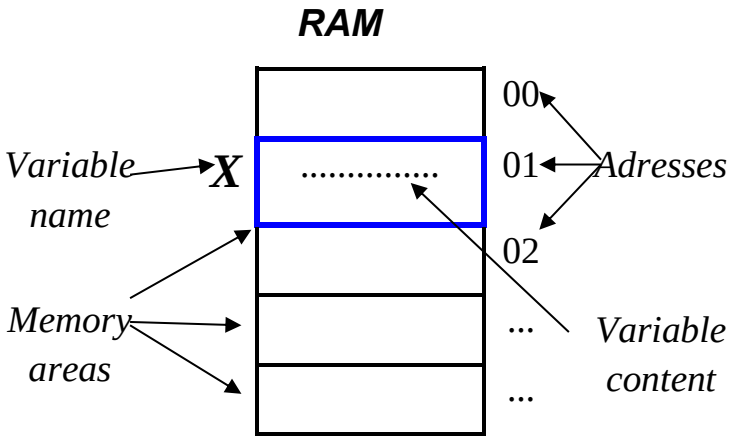
In algorithmics, a variable is a data element with a name and a value that can change during the algorithm's execution.

From the computer's perspective, a variable is just a memory location at an arbitrary address identified by a name and capable of storing data of a type defined in advance. The name is used to locate this memory location so that the computer can access it directly.

The variable can be schematized as a box labeled with a name, having a size, content, and memory address.

Example:

Let's assume we have a variable named ***x***; it can be represented in memory as follows:



To be able to use a variable, we must first declare it in the declaration part of the algorithm in order to allocate memory space for it.

3.2) Constants

Constants are fixed data (values) that do not change during the execution of the algorithm. A constant is identified by a name and has a value that must be set before the algorithm is executed. The value of the constant can be numeric, textual, logical, etc.

Just like a variable, a constant from the computer's perspective is a memory area labeled with a name that stores a value, but this value remains unchanged during the execution of the algorithm.

3.3) Notion of Identifier

An identifier is the name assigned to an object in the algorithm, whether it's the algorithm itself, a variable, a constant, etc. This name allows the computer to distinguish them and humans to understand and refer to them. An identifier is

a sequence of alphanumeric characters that must adhere to the following criteria:

- It must start with a letter or an underscore (_).
- It continues with any number of letters, digits, or underscores (no symbols or spaces).
- It cannot be a keyword.

Note:

In algorithmics, there is no distinction between lowercase and uppercase letters.

Examples:

A, DELTA, X1, VAL, i, K, MM, B_727 are valid identifiers.

END, 12MOT, VAL*2 are invalid identifiers.

4) Data Types

The data manipulated by the algorithm and stored in variables are not all of the same type. This is why it is necessary to assign a data type to each variable to specify what it can contain.

Types serve various purposes, including:

- Defining the set of values that a variable can take. For example, a variable defined as an integer cannot receive the value 7.46.
- Specifying the set of operations, typically called operators, that can be applied to the variable. For instance, you cannot perform multiplication on two string variables.
- Informing the compiler about the amount of memory required to store the variable's value. Thus, an integer and a real number do not have the same size and do not occupy the same space in memory.

In algorithmics, there are several data types, which can be categorized into two classes:

- Elementary types
- Structured or composite types

In this chapter, we focus on elementary types. Structured types will be the subject of other chapters.

4.1) Elementary Types

These are simple types, meaning a variable of these types contains only one value at a time. In elementary types, we distinguish between standard types and non-standard types.

4.1.1) Standard Types

In algorithmics, there are five standard types, also known as elementary data types or primitive data types:

a) Integer

The integer type includes integer numerical values, both positive and negative. It is denoted by the name: **`Integer`**.

b) Real

The real type includes real (floating-point) values, both positive and negative. It is denoted by the name: **`Real`**.

The usual representation for real numbers is the decimal notation "a.b," for example: 3.14, -7.22, ...

c) Character

The character type represents the domain of characters, including lowercase and uppercase alphabetic letters, numerical characters, special characters (., ?, !, <, >, =, , +, ... etc.), and the space character. This type is denoted by the name: **`Character`**.

However, a variable of this type can only contain a single character at a time. Characters are enclosed in single quotes (apostrophes) " ' ".

Examples: 'R', '5', '*', ...

d) String

This type refers to the set of strings that can be formed by composing characters. It is denoted by the name: **`String`**.

Strings of characters are delimited by double quotes.

Examples: "Algo", "123", "True".

e) Boolean (Logical)

The Boolean type is also called the logical type and represents the logical domain, which contains only two values (True and False). It is denoted by the name: **`Boolean`**.

f) Note on Types

Each type has a specific size and representation in computer memory. Different forms of constants should not be confused.

Examples:

- The value 3 (integer type)
- The value 3.0 (real type)

- The value '3' (character type)
- The value "3" (string type).

4.1.2) Non-Standard Types (or User-Defined Types)

Here, we distinguish between enumerated types and interval types.

a) Enumerated Type

The enumerated type is defined by the algorithm's designer and is not known to the compiler. In defining an enumerated type, the designer lists a finite and ordered sequence of values that a variable of this type can take. The defined type must be declared in the declaration part of the algorithm using the keyword **`TYPE`**. The declaration of an enumerated type is done as follows:

```
TYPE name_Type = (val1, val2, ..., valn);
```

Where **`name_Type`** is an identifier representing the name of the type, and **`val1 ... valn`** are the values of this type.

Example:

The **`Season`** type can be defined as follows:

```
TYPE Season = (Spring, Winter, Autumn, Summer);
```

b) Interval Type

This type allows us to define a range of values of a scalar type by specifying its lower and upper bounds. The types of the constants that are the bounds of the interval specify the type of the scalar from which the interval is derived. However, the interval can be a range of integer or character values but not real numbers or strings.

Like the enumerated type, an interval type is not known to the compiler and must be declared in the declaration part of the algorithm using the **`TYPE`** keyword as follows:

```
TYPE name_Type = lower_Bound..upper_Bound;
```

Where **`name_Type`** represents the name of the type, and **`lower_Bound`** and **`upper_Bound`** are respectively the lower and upper bounds of the interval.

Examples:

The **`Month`** and **`Alphabet`** types are declared as follows:

```
TYPE Month = 1..12;          // Interval derived from Integer type
TYPE Alphabet = 'a'..'z'; // Interval derived from Character type
```

4.2) Declaration of Variables and Constants

4.2.1) Declaration of Variables

Declaring a variable involves assigning it a name (identifier) and a type.

The declaration of a variable begins with the keyword **VAR**, followed by the variable name, followed by a colon " **:** " and then the variable type. The declaration syntax is as follows:

```
VAR name_Variable: type_Variable;
```

Example:

```
VAR age: Integer;
VAR moy: Real;
VAR prenom: String;
VAR admis: Boolean;
VAR sitFam: Character;
VAR m: Month;
```

Remarks:

- Declaring a variable involves reserving memory space corresponding to the declared variable's type.
- You can declare multiple variables using a single **VAR** keyword, even of different types.
- Variables of the same type can be declared together, separated by commas.
- It is possible to place multiple variable declarations in the same line separated by semicolons.

Example:

```
VAR a : Integer ;          VAR avg : Real ;
VAR b : Integer ;          VAR s : Season ;
VAR c : Integer ;
```

These declarations can be refined as follows:

```
VAR a,b,c : Integer; avg : Real; s: Season;
```

4.2.2) Declaration of Constants

The declaration of a constant begins with the keyword **CONST**, followed by the constant name, followed by the equals sign " = ", and then the constant value.

The declaration syntax is as follows:

```
CONST name_Constant = value_Constant;
```

Example:

```
CONST pi = 3.14;
CONST FirstName = "Mohammed";
CONST Nb = 10;
CONST Point = '.';
```

Remarks:

- A single **CONST** keyword is sufficient to declare multiple constants.
- You can place multiple constant declarations in the same line, separated by semicolons.
- Character constants should be enclosed in single quotes " ' " and string constants in double quotes (quotation marks).
- The declaration of constants comes before the declaration of variables.

Example:

The previous example is equivalent to this one:

```
CONST pi=3.14; FirstName="Mohammed"; Nb=10; Point='.';
```

5) Basic Operations

Let's first discuss the concepts of operators, operands, and expressions.

5.1) Operator and Operand

An operator is a symbol of an operation that allows to perform actions on variables or carry out calculations.

An operand is an entity (variable, constant, or expression) used by an operator.

There are several types of operators.

5.1.1) Arithmetic Operators

These are the usual arithmetic operations.

+ : Addition	**Div** : Integer division
- : Subtraction	**Mod** : Modulo (remainder of integer division)
***** : Multiplication	**^** : Exponentiation (Power)
/ : Division	

Finally, you can use parentheses with the same rules as in mathematics.

The above operators are binary operators, except for "**-**" which can also be unary and signifies a change of sign.

Addition, subtraction, multiplication, and division are applicable to both **`integer`** and **`real`** operands, while **`div`** and **`mod`** are only applicable to **`integer`** operands.

5.1.2) Logical Operators

These operators are used to connect **`logical`** or **`boolean`** operands. These operators are **`NOT`**, **`AND`**, **`OR`**.

`NOT` is a unary operator that negates a logical value. **`AND`** and **`OR`** are binary operators that combine two logical operands.

5.1.3) Comparison Operators

Comparison operators allow you to compare two values of the same type and return a boolean result (true or false) based on order relations: natural order for integers and real numbers, and ASCII lexicographic order for characters and strings. These operators are "<", ">", "=", "≠", "≤", "≥".

Comparison operators can be applied to operands of type **`integer`**, **`real`**, **`character`**, or **`string`**.

5.1.4) Alphanumeric Operator

This operator allows you to concatenate, or in other words, join two strings of characters. It is symbolized by the "+" sign.

This sign, when applied to numbers, performs addition, and when applied to strings of characters, performs concatenation.

Example:

"Hello" + " world" gives "Hello world".

5.2) Expression

An expression is a set of operands connected by operators and is equivalent to a single value. The operands can be direct values, constants, variables, or other expressions.

Each expression is associated with a type, which is the type of the value it represents.

Examples:

Let **`a`** and **`b`** be two **`integer`** variables:

- **`12.5 * a + (b/2)`** is an arithmetic expression.
- **`a > b and b ≥ c`** is a logical expression.
- **`"Algo" + "rithm"`** is a string expression.

5.2.1) Validity of an Expression

To ensure the validity of an expression and determine its type, you need to check the syntax of the expression, the compatibility of the operand types it consists of, and the validity of the operators. The type of the operands defines the type of the expression, so it's essential that the operand types of an operator are compatible. For example, adding an **`integer`** and a **`character`** doesn't make sense.

5.2.2) Evaluating an Expression

The evaluation of an expression is based on the priority rules between operators the following order (from highest to lowest precedence):

1. Unary operators: Logical **`NOT`**, Unary **`-`**
2. Power operator: **`^`**
3. Multiplicative operators: **`*`**, **`/`**, **`div`**, **`mod`**, Logical **`AND`**
4. Additive operators: **`+`**, **`-`**, Logical **`OR`**
5. Relational operators: **`<`**, **`≤`**, **`>`**, **`≥`**, **`=`**, **`≠`**

Remarks:

- For operators with the same precedence (priority), the expression is evaluated from left to right.
- If there are parentheses, innermost ones are evaluated first.

Example:

Consider the following declarations:

```
Const i = 3;
Var  j, k: Integer; x, y, z: Real;
     A, B: Boolean; c: Char; ch1, ch2: String;
```

Evaluate and determine the type of the following expressions:

- **`12 * 3 + 5`** is correct and has a value of 41 (**`integer`** type).
- **`12 * (3 + 5.1)`** is correct and has a value of 97.2 (**`real`** type).
- **`-x - k div 3`** is correct and has a **`real`** type.
- **`z mod j + i`** incorrect because **`MOD`** is not valid for **`real`** type.
- **`NOT y AND B>0`** incorrect: **`NOT`** should be applied to a **`boolean`**.
- **`c='c' OR c='t'`** incorrect: **`OR`** is not valid for **`character`** type.
- **`(c='c') OR (c='t')`** is correct and has a **`boolean`** type.

6) Basic Instructions

An instruction is a fundamental action that commands the computer to perform a calculation or communicate with one of its input or output devices.

6.1) Assignment Statement

Assignment is an operation that assigns a value to a variable. It is denoted by the symbol "←". The syntax of this instruction is as follows:

```
variable ← Value;
```

It is read as: **variable** receives **value** or **variable** gets **value**.

The left-hand side of an assignment must be a variable name, while the right-hand side (**value**) can be a direct value, a constant, another variable, or an expression. In the case of an expression, it is evaluated, and its result is stored in the variable.

Examples:

A ← 3; assign the direct value 3 to variable **A**.

B ← A; assign the content of variable **A**, which is 3, to variable **B**.

B ← B - 2; evaluate the expression **B-2** and put the result (equal to 1) into variable **B**.

Course ← "Physics"; assign the string "Physics" to the variable **Course**.

Remarks:

1) Assignment copies the value from the right-hand side to the variable on the left-hand side without modifying the right-hand side.
2) In an assignment, the type of the value (right-hand side) must match the type of the variable (left-hand side).

6.2) Input/Output Instructions

6.2.1) Reading (input)

Reading is a basic action that allows to enter a value from the keyboard and assign it to a variable. The syntax of this instruction is as follows:

```
READ(variable);
```

This instruction means to place the value entered via the keyboard by the user into the memory location reserved for the **variable**.

Example:

```
Var x, y: integer; a: real;

READ(x);
```

READ(y);

READ(a);

Remarks:

- The value entered via the keyboard must be compatible with the receiving variable.
- Several reading instructions can be grouped together into a single instruction by separating the variables to be read with commas.
  READ(v1); READ(v2); …; READ(vn);
  is equivalent to: **READ(v1,v2,…,vn);**

Example:

The reading instructions in the previous example can be combined into a single instruction: **READ(x,y,z);**

6.2.2) Writing

The writing is an instruction allowing to display a value on the screen. This value can be a direct value, a constant, the content of a variable, a message, the result of an expression, etc.

The syntax of this instruction is as follows:

WRITE(Value);

Example:

WRITE("The mark: "); displays the message "The mark: ".

WRITE(mark); displays the content of the variable **mark**.

WRITE(6*2+5); evaluates the expression **6*2+5** and displays its result (**17**).

Remarks:

- Multiple writing instructions can be grouped into a single instruction by separating the values to be displayed with commas.
- Messages to be displayed must be enclosed in double quotation marks.

Example:

WRITE("The mark: ", mark, "/20");

This instruction displays the message "**The mark:**" followed by the content of the variable **mark**, followed by the string "**/20**".

7) Building a Simple Algorithm

To construct an algorithm, you must combine all the concepts presented above. Thus, an algorithm is composed of:

- **Header**: Indicates the name of the algorithm.
- **Declaration Part**: Where we describe the objects we will use in the algorithm (variables, constants, types, etc.).
- **Algorithm Body**: Encompasses all the instructions of the algorithm placed between **BEGIN** and **END**. These instructions are typically presented in the following order:
  - ***Data Input:*** we should first retrieve the necessary data through reading.
  - ***Data Processing:*** we perform the necessary operations to solve the problem using assignment instructions.
  - ***Result Output:*** Finally, we display the results obtained using the writing instruction.

Recall that the structure of an algorithm takes the following form:

```
ALGORITHM <name of the algorithm>;        } Header
Const <list of constants>;                }
Type <list of types >;                    } declaration part
Var <list of variables>;                  }
BEGIN                                     }
<Sequence of actions>                     } algorithm body
END.                                      }
```

Example:

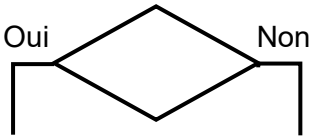
The algorithm for calculating the sum of two integer numbers is as follow:

```
ALGORITHM  sum;
VAR x,y,s:Integer;
Begin
WRITE("Enter 2 numbers: ");
READ(x,y);
s ← x + y;
WRITE("The sum of the 2 numbers is ",s);
END.
```

8) Representation of an Algorithm Using a Flowchart

A flowchart is a graphical representation of a problem's solution, which has the advantage of being easily understandable but comes with the disadvantage of consuming significant space.

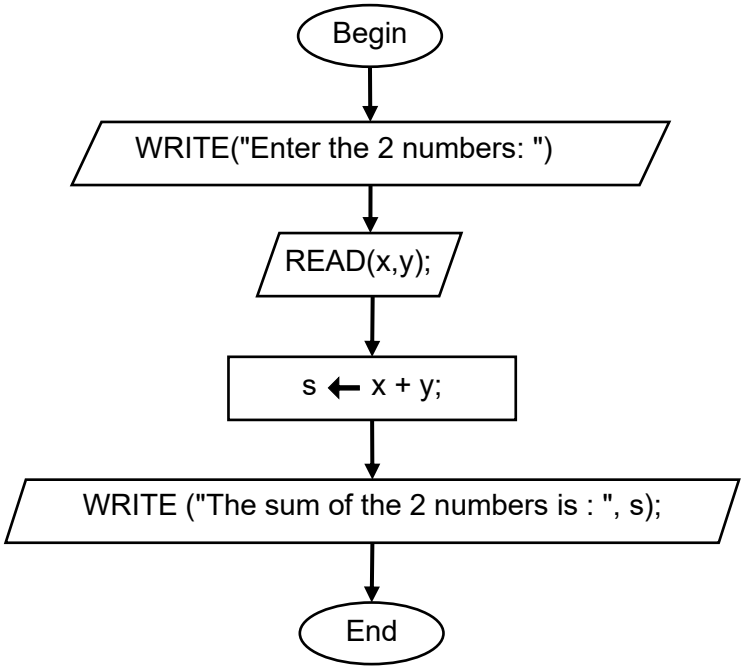
Operations within a flowchart are represented by symbols connected to each other by arrowed lines that indicate the flow path. The main symbols used are as follows:

Symbol	Role
	Used to mark the beginning and end of a flowchart.
	Used to mark read and write operations.
	It is used for assignment operations (actions).
	Used to represent tests or conditional branching.
	Symbol of connection between various symbols. It also indicates the sequencing of operations.

The transition from an algorithm to a flowchart is achieved by representing each of its instructions using the corresponding graphical form and connecting them with arrows.

Example:

The algorithm for calculating the sum of two numbers mentioned earlier can be represented by the following flowchart:



9) Translation into C Language

9.1) The C Language: Presentation

The C language is a general-purpose programming language invented in 1972 by Dennis Ritchie and Ken Thompson with the aim of developing the famous UNIX operating system. Today, C has become one of the most widely used programming languages, especially for system programming. As a result, the kernels of major operating systems like Windows and Linux are developed primarily in the C language.

9.2) Basic Elements of the C Language

9.2.1) Structure of a C Language Program

The simplest structure of a C program is as follows:

```
<Library Declarations>
main()
{
    <Constant and Variable Declarations>
    <Instructions>
}
```

The first part includes the declaration of the libraries of functions to be used in the program. Among these libraries, we mention:

- **`stdio.h`** : This is the library of standard input and output functions. Including the **`stdio.h`** library is done using the preprocessor directive: **`#include <stdio.h>`**
- **`math.h`** : This is the library of basic mathematical functions. Including the **`math.h`** library is done using the preprocessor directive: **`#include <math.h>`**

It is important to pay attention to the following elements:

- **`main`** is a predefined name of the main function that must exist in a C language program. It should be in lowercase.
- The parentheses after the **`main`** function are mandatory.
- The curly braces (**`{`** and **`}`**) mark the beginning and end of a block of instructions or a function. They replace **`BEGIN`** and **`END`** in algorithmic notation.

9.2.2) The Declaration Section

a) Identifiers

Identifiers in the C language have the same characteristics as in algorithmics. Additionally, the C language is case-sensitive, meaning it distinguishes between uppercase and lowercase letters.

Example:

The identifiers: **`NOM`**, **`Nom`**, and **`nom`** are three different identifiers in C.

b) Predefined Types in the C Language

The various data types recognized in the C language are summarized in the following table:

Data type	Signification	Size (Bytes)	Range of acceptable values
`char`	Character	1	-128 to 127
`unsigned char`	Unsigned Character	1	0 to 255
`short`	Short Integer	2	-32768 to 32767
`unsigned short`	Unsigned Short Integer	2	0 to 65535
`int`	Integer	4	-2147483648 to 2147483 647
`unsigned int`	Unsigned Integer	4	0 to 4294967295
`long`	Long Integer	4	-2147483648 to 2147483647
`unsigned long`	Unsigned long Integer	4	0 to 4294967295
`float`	Flottant (real)	4	3.4×10^{-38} to 3.4×10^{38}
`double`	Double Flottant	8	1.7×10^{-308} to 1.7×10^{308}
`long double`	Long Double Flottant	10	3.4×10^{-4932} to 3.4×10^{4932}

Remarks:

- The C language does not have **`Boolean`** and **`String`** types.
- The C language does not differentiate between a character itself (e.g., 'A') and its ASCII code. Therefore, you can represent the **`char`** type as an **`integer`** encoded on 1 byte.

c) Variable Declaration

Variable declaration can be done in two possible ways:

```
type_Variable name_Variable;
type_Variable name_Variable = initial_value;
```

`type_Variable` is the data type contained in the variable (one of the types in the table above), **`name_Variable`** is the variable name, and **`initial_value`** is the initial value of the variable.

Examples:

```
int x, y;
float z;
char a;
```

d) Constant Declaration

The declaration of constants in the C language is in the following form:

```
#define name_Constant value_Constant
```

Remarks:

- Each constant must be declared in a separate line, and that line does not end with a semicolon.
- Character constants must be enclosed in single quotes.
- String constants must be enclosed in double quotes.

Examples:

```
#define nb 2              // Integer constant named nb with a value of 2
#define pi 3.14           // Constant named pi with a value of 3.14
#define b 'v'             // Character constant named b with a value of 'v'
#define a "Hello"  // Constant named a with a value of "Bonjour"
```

9.2.3) Processing Section

a) Operators

a.1) Arithmetic Operators

The classical arithmetic operators include the unary operator "**-**" (sign change) and the binary operators:

+ : addition **-** : subtraction ***** : multiplication

/ : division (integer and real) **%** : remainder of the division

Note:

In C, "**/**" is used for both integer and floating-point division. If both operands are integers, the "**/**" operator will perform integer division (quotient). However, it will yield a floating-point value as soon as one of the operands is a floating-point number. For example, **a=9/6**; will return **1**, while **a=9.0/6**; will return **1.5** because one of the operands is a real number (**9.0**).

a.2) Logical Operators

! : Negation **&&** : Logical AND **||** :Logical OR

Since the boolean type does not exist in C, the value returned by logical operators is an integer, which is **1** if the condition is true and 0 otherwise.

a.3) Relational Operators

> : strictly greater	>= : greater than or equal	== : equal
< : strictly less	<= : less than or equal	!= : not equal

The value returned is of type **int**: 1 if the condition is true and 0 otherwise.

b) Assignment

Assignment in the C language is symbolized by the "**=**" sign. Its syntax is:

```
name_Variable = Value ;
```

Examples:

```
A = 3 ;          // Assign the direct value 3 to variable A
B = A-1 ;        // Store the result of the expression A-1 in variable B
```

c) Reading (input)

Input in C is done using the **scanf** function from the **stdio.h** library. It allows you to enter data from the keyboard and store it at the addresses specified by the function's parameters. The syntax of this function is:

```
scanf("control string", &variable1, &variable2,…)
```

The function's parameters consist of a **control string** and the address (indicated by the "**&**" sign) of the variables where the input data should be stored.

The **control string** specifies the format in which the input data is to be converted. Thus, for each variable, a format specifier is specified. Format specifiers are indicated by a character preceded by the "**%**" sign. The format code and the variable type must match. The input formats for the **scanf** function are summarized in the following table:

Format	Data type	Data representation
%d	Int	Signed decimal
%hd	short int	Signed decimal
%ld	long int	Signed decimal
%u	unsigned int	Unsigned decimal
%hu	unsigned short int	Unsigned decimal
%lu	unsigned long int	Unsigned decimal
%f	Float	Floating-point, fixed decimal

%lf	Double	Floating-point, fixed decimal
%Lf	long double	Floating-point, fixed decimal
%c	Char	Character
%s		String of characters

Example :

```
int a; float b,c;
Scanf("%d",&a) ;
Scanf("%f%f",&b,&c) ;
```

d) Writing (output)

Writing is done using the **printf** function from the **stdio.h** library.

The **printf** function allows to display data specified by the function's parameters on the screen. The syntax of this function is:

```
printf("control string",expression1,expression2,…);
```

The **control string** contains the text to be displayed and the format specifiers corresponding to each expression in the parameter list. The format specifiers are the same as those presented in the table above.

Example:

```
float a=3.14;
printf(”The value of P is %f”, a);
```

In this example, the string "**The value of P is** " is displayed on the screen, followed by the value **3.14** stored in the variable **a**.

9.2.4) Example of a C Program

The following program calculates the sum of two integers, as described in the previous algorithm.

```
#include <stdio.h>
main(){
 int x,y,s;
 printf("Enter 2 numbers: ");
 scanf("%d%d",&x,&y);
 s = x + y;
 printf("The sum of the 2 numbers is %d",s);
}
```