الجمهورية الجزائرية الديمقراطية الشعبية

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

Higher Normal School of Technological Education. SKIKDA

Department of Mathematics and Computer Science

وزارة التعليم العالـي و البحث العلمـي

المدرسة العليا لأساتذة التعليم التكنولوجي-سكيكدة-

قسم: الرياضيات و الإعلام الآلي

Lecture notes

Algorithmics 1

Intended for first-year undergraduate students in Computer science

Established by :

Dr. Rida MEZGHACHE

r_mezghache@enset-skikda.dz

r_mezghache@yahoo.fr

2024/2025

Preface

This document serves as the course and exercise material for the "Algorithmics 1" course. It is primarily designed for first-year computer science students in the Department of Mathematics and Computer Science at the Higher Normal School of Technological Education, SKIKDA. However, this handout can also benefit non-computer science students interested in learning the basics of algorithms and programming.

The "Algorithmics 1" course is scheduled in the first year of Computer science. It is characterized by a coefficient of 5, and a weekly hourly volume of four and a half hours, divided as follows: one hour and a half of lectures, one and a half hour of tutorials, and one and a half hour of practical work.

Algorithmics is the science whose subject of study is the algorithm. This discipline, at the intersection of mathematics and computer science, focuses on the creation, description, and analysis of algorithms. This field has particularly flourished with the construction of computers and the invention of programming languages. Today, algorithmics represents the first step for a computer scientist to enter the world of automated processing. In computer science, algorithms are indeed ubiquitous. They are, in fact, the backbone of computing because an algorithm provides the computer with a specific set of instructions. It is these instructions that enable the computer to perform tasks. As for data structures, they are ways of organizing data in memory to facilitate processing. The first pertains to the dynamic aspect of task automation, and the second to the static aspect. The study of algorithmics is essential for learning computer science. However, beyond computer science, algorithmic thinking is crucial in various fields. It involves the ability to define clear steps to solve problems.

The main objective of this course is to gradually guide students in assimilating and using the concepts and techniques necessary for constructing algorithms to solve encountered problems. This involves developing analytical skills in students and teaching them algorithmic reasoning so that they can understand and analyze the problem at hand, describe it in terms of algorithms (in pseudo-code) and programs in the C++ language, choose the appropriate data structures, and effectively address the inherent challenges of programming.

The content of this course material is in accordance with the latest curriculum framework established for the first year of Computer Science, as proposed by the national pedagogical committee of the Computer Science domain.

However, I have endeavored to present the content of this course material in a clear and explicit manner, taking into account a simple pedagogical approach supported by examples and solved exercises. This is done to ensure maximum clarity and to make the necessary knowledge and concepts related to algorithmics more accessible to novice students. As such, no prerequisites are required to follow this document.

To effectively present its content and align closely with the compliance framework, this handout is organized into four chapters.

The first chapter, titled "Introduction," serves as an introduction to computer science and algorithmics. It presents the definitions of basic computer science concepts, introduces the notion of an algorithm through concrete examples, and outlines the various steps in problem-solving in computer science. It explains the general structure of an algorithm in pseudocode and introduces its basic elements (variables, expressions, instructions, etc.).

The second chapter, “Elements of a programming language – C++". C++ is a powerful programming language widely used in finance, security, networking, gaming, and other fields. The chapter introduces the fundamentals of programming and the C++ language. It covers the foundations of programming topics, such as variables, data types, operators, conditional instructions and loops, building a solid understanding of C++.

The third chapter “Structure Objects” introduces arrays, multidimensional arrays, and strings, which are composite types that allow multiple values of the same type to be grouped into a single variable. For each of these types, the details of declaration and manipulation are described in the C++ language. It also introduces structures and sets

The last chapter, titled "Functions," addresses the possibility of defining new reusable blocks of code that perform specific tasks. They allow you to encapsulate code logic into named sections, making your code more organized, modular, and easier to maintain. Functions take inputs (called *parameters* or *arguments*), process them, and return a result.

Finally, I hope that this modest handout proves to be a valuable addition to the educational resources and serves as a valuable course material for our students.

Table of Contents

List of Figures

List of Tables

Chapter I. Introduction

I.1) Natural language vs programming language

Let's consider for a moment what a language is. We could say that a language is a tool for expressing and recording human thoughts. In other words, it's a mechanism known to us and our partners, enabling us all to understand and be understood. We use language to speak, write, read, listen and think.

At least one language accompanies us throughout our lives - our mother tongue, which we learn almost unconsciously from the start. Many of us will also learn other languages. The languages we use to communicate with others are called natural languages.

However, there are languages whose creation and development are dictated by specific needs.

Such languages include programming languages. A programming language is defined by a certain set of rigid rules.

These rules determine which symbols (letters, numbers, punctuation marks, etc.) can be used in the language. This part of the language definition is called the lexicon.

Another set of rules determines the appropriate ways of combining symbols: this is the syntax of the language.

We must also be able to recognize the meaning of each statement expressed in the given language, and this is what we call semantics.

Any program we write must be error-free in these three ways: lexically, syntactically and semantically. Otherwise, the program won't work, or it will produce unacceptable results.

Why do we need a programming language?

A computer, even the most technically sophisticated, is devoid of the slightest trace of intelligence. It only responds to a predetermined set of known commands. The recognized commands are very simple. We can imagine the computer responding to commands such as "take this number, add to another and save the result".

A complete set of well-known commands is called an instruction list. It's an alphabet commonly referred to as machine language. It's the simplest, most basic language we can use to give commands to our computer. You could say it's the computer's mother tongue.

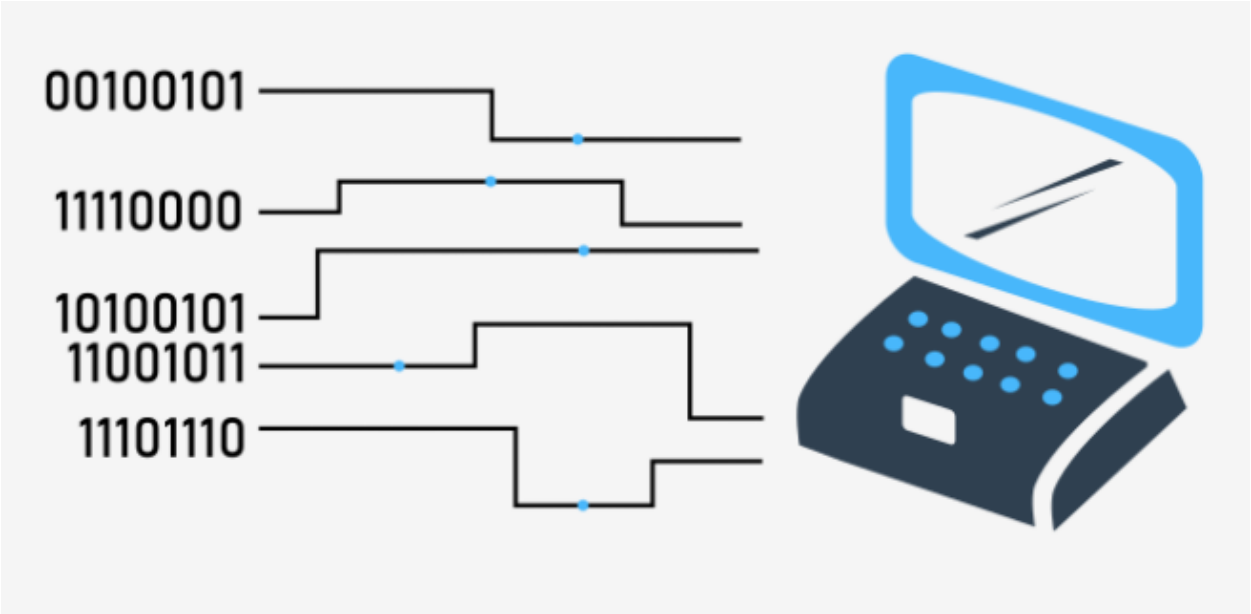

Figure 1 - Binary language

It is possible to code a computer program directly in machine language, using elementary instructions (commands). This type of programming is tedious, time-consuming and highly susceptible to programmer error. In the early days of computer technology, this was the only programming method available, and it soon revealed its serious shortcomings.

Thus, a program written for a specific type of computer could be completely useless for other computers and vice versa. Secondly, programs written in machine language are very difficult for humans to understand, even for experienced programmers. It's also true that developing a program in machine language is time-consuming, expensive and cumbersome.

All these circumstances lead to a need for some kind of bridge between the language of people (natural language) and computer language (machine language). This bridge is also a language - a common intermediate language for humans and computers working together. Such a language is often called a high-level programming language.

A language like this is at least somewhat similar to a natural language: it uses human-readable symbols, words and conventions. This language enables humans to express complex commands to computers.

You can translate your program into machine language. What's more, the translation can be carried out by computer, making the whole process fast and efficient.

All you need to know is a high-level programming language. If there's a translator designed for a specific computer, you can run your program without any problems. In other words, programs written in high-level languages can be translated into an unlimited number of different machine languages, enabling them to be used on many different computers. This is known as portability.

I.2) Algorithms

Let's move on to the basics. The aim is to explain how to specify algorithms. We begin with a few definitions of the term algorithm.

The word algorithm comes from the name of the famous Arab mathematician Muhammad ibn Musa Al Kwarizmi (780-850).

An algorithm is a sequence of elementary operations performed in a given order to solve a problem or accomplish a task. As a science, we speak of algorithmics.

A computer is a machine capable of automatically executing a sequence of operations. To solve a problem using a computer, you need to:

1. Analyze the problem (what): define what I have as data (inputs) and what I need as results (outputs).
2. Determine the solving method (how): determine the sequence of operations to be performed to solve the problem. Several methods can be found to solve a single problem, but the most efficient must be chosen.
3. Formulate the final algorithm: represent the method of resolution by an algorithm written in an algorithmic language, also known as an algorithm description language (ADL) or pseudo-code.
4. Translate the algorithm into a suitable programming language.

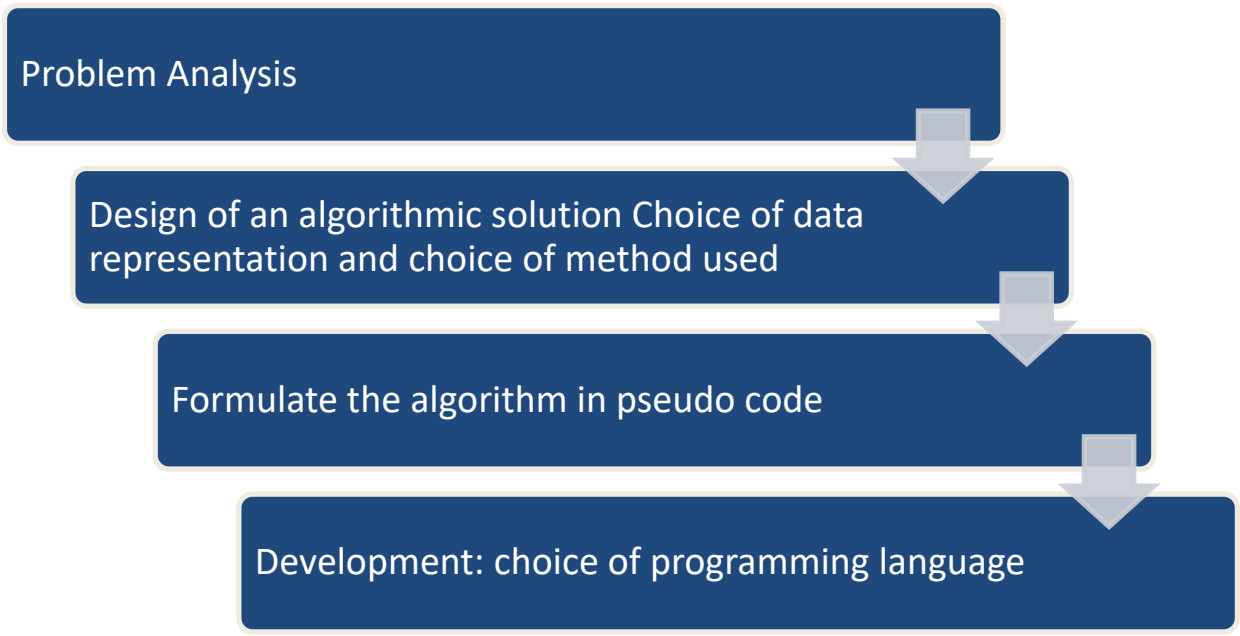

Figure 2 - Structured approach to problem-solving in programming and algorithm development.

I.3) Definitions

I.3.1) Algorithmics

is the science focused on the study and application of algorithm design, analysis, and implementation. It involves the principles and techniques for developing step-by-step procedures (algorithms) designed to solve specific problems or carry out tasks efficiently and logically.

I.3.2) Algorithm

1. A finite, sequential sequence of rules that can be applied to a finite number of data items, enabling similar classes of problems to be solved.
2. Well-defined computational procedure which takes as input a value, or a set of values, and gives as output a value, or a set of values. An algorithm is therefore a sequence of computational steps that transform input into output.
3. An algorithm is the composition of a finite set of steps, each step consisting of a finite number of operations, each of which is:

- Rigorously defined and unambiguous, after executing an instruction, there should be no doubt as to which instruction to execute next.
- Effective (can be executed in a finite time).
- Irrespective of the programming language used, the notion of algorithm is more general than that of program.

4. An algorithm can also be seen as a tool for solving a well-specified computational problem. The problem statement specifies, in general terms, the desired relationship between input and output. The algorithm describes a specific computational procedure for obtaining this input/output relationship.

I.3.3) Instance

A particular value in the set of input values is called a problem instance. This input consists of all the particular data needed for the algorithm to perform its task.

Instances are crucial because they enable us to evaluate an algorithm’s performance across a range of conditions. Varying instances can impose different computational demands, influencing the algorithm’s execution time and memory consumption.

I.3.4) Correctness

In algorithmics, correctness refers to an algorithm's ability to produce the correct output for every valid input within its defined problem space. It involves ensuring that an algorithm performs its intended function without errors, consistently delivering expected results.

Total Correctness implies that a totally correct algorithm will always terminate and produce the correct output for all valid inputs.

An algorithm is said to be **correct** if, for each input instance, it terminates and produces the correct output. A correct algorithm is said to **solve** the given problem. An incorrect

algorithm may not terminate for certain input instances, or may even terminate with an answer other than the desired one.

I.3.5) Data structure

Is a specific way to store and organize data in a computer to facilitate access for use and modification. For example, an array. In the case of priority queues.

Example

Let's suppose, for example, that we need to sort a sequence of numbers in ascending order.

Here's how we formally define the sorting problem:

Problem: Sort a sequence of integers in ascending order.

Input: sequence of n numbers a_1, a_2, ..., a_n.

Output: permutation (reorganization) a'_1, a'_2, ..., a_n of the sequence given as input, so that $a'_1 \leq a'_2 \leq ... \leq a_n$.

Thus, from the sequence (6,9,2,4), a sorting algorithm will provide the result (2,4,6,9). this is called an instance of the sorting problem. In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) required to compute a solution to the problem.

The value (6,9,2,4) is an instance of the problem.

I.4) Algorithm description language

It is essential to have a language that is not tied to implementation. This enables more precise description of data structures, as well as more flexible and "readable" algorithm writing.

An ADL can be composed of alphanumeric character strings, operating signs (+,-,*,/,<,<=,>=,>,<>,==,=,or,not,and), reserved keywords, and punctuation marks: "= ; ( ) start end //" Start and end tags can be replaced by { and }.

Algorithms are the choice of an algorithm (Calculation) and the choice of a data structure. The two are inseparable

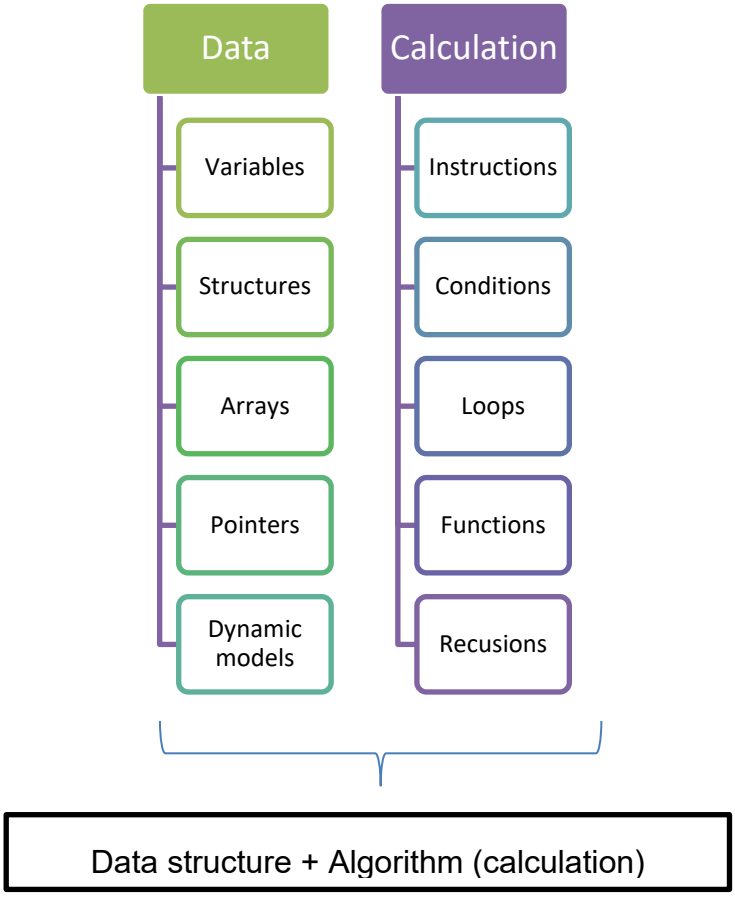

Figure 3 - Algorithmic basis

I.5) Some themes

- Sorting: rearranges and classifies data. There are many different ways of sorting a set of data, differing in the sequence of steps performed.
- Search: locate data in a file. Methods are highly varied and depend on how the data is organized in memory.
- Mathematical algorithms: methods derived from numerical analysis and arithmetic. Problems dealing with integer arithmetic, polynomials and matrices.

I.6) Example

Consider the problem of calculating the sum of two numbers. This problem can be solved as follows:

A. Analysis

To calculate the sum:

As input, we need two values: value1 and value2.

At the output, we'll have the sum of these two values.

B. Solution design

Calculating the sum involves:

1. Have the two values (read value1 and value2).
2. Add the two values.
3. Display the result (write the sum).

This way of representing an algorithm is called "Enumeration of steps".

C. Writing in ADL

In computing, we use an algorithmic language to write an algorithm; and the algorithm above becomes.

```
Algorithm Sum
Variable
 value1, value2, sum : integer
Begin
 Read (value1, value2)
 sum ← value1 + value2
 Write (sum)
End
```

I.7) Presentation of the adopted formalism

I.7.1) Algorithm structures

An algorithm consists of three main parts (Figure 5):

- **Header**: this part gives the algorithm a name. It is preceded by the word Algorithm;
- **Declarative part:** contains variables and constants;
- **Algorithm body:** this is the processing part, containing the algorithm's instructions. It is delimited by the words Begin and End.

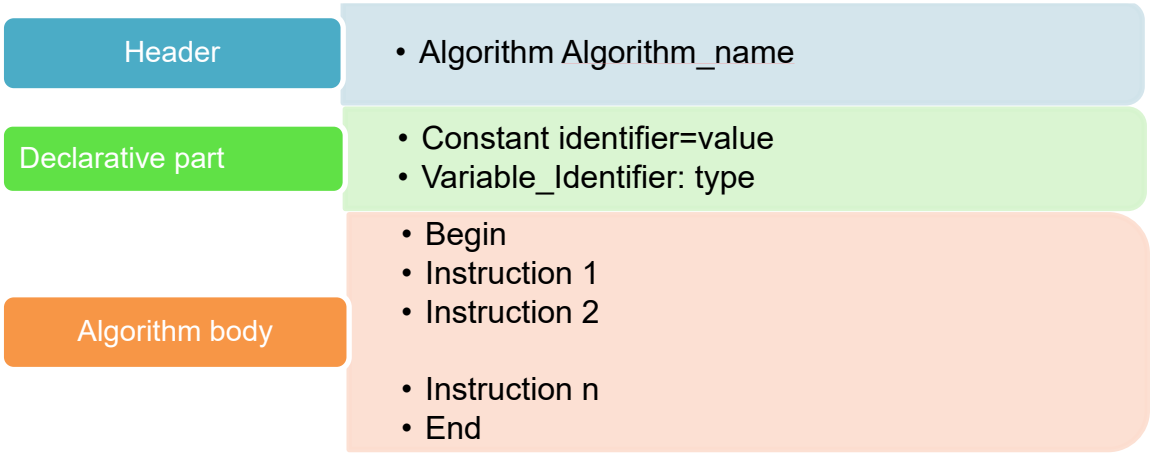

Figure 4 - Algorithm structure

I.7.1.1) Declarative part

Data are the objects manipulated in the algorithm. In an algorithm, any data used must be declared. Data can be variables or constants.

A. Declaration of constants

Like a variable, a constant corresponds to a reserved memory location which is accessed by the name assigned to it, but whose stored value will never be modified during the program.

Syntax

```
Constant constant_identifier= value
```

Example

Constant PI = 3.14

B. Variable declaration

Data and the results of intermediate or final calculations are stored in **memory cells** corresponding to **variables.**

Thus, a variable (see next figure) is stored in a named memory location, of fixed size (or not), taking on an indefinite number of different values as the algorithm runs.

The declaration part consists of listing all the variables that will be needed during the course of the algorithm.

The value of a variable can vary during the course of the algorithm. A variable is characterized by:

- Name (identifier): must be explicit, i.e. indicates the variable's role in the algorithm.
- Type: indicates the values that can be taken by a variable.
- The value: indicates the magnitude taken by the variable at a given moment.

Technically speaking, a variable corresponds to a memory cell with: the variable name is the address of the memory cell; the type indicates the size (number of bytes) of the cell; the value represents the contents of the cell.

Each declaration must include the variable name (identifier) and its type.

Syntax

```
Variable identifier: type
```

Example

```
Variable Surface: real
a, b, c, d: integer
Last_Name: string
absent: logical
```

C. Identifier

An identifier is the name given to a variable, function, etc. This name must begin with a letter, followed by a sequence of letters and numbers, and must not contain

spaces.

D. Types

The type corresponds to the type or nature of the variable you wish to use. It indicates the values that can be taken by this variable. Declaring a variable involves associating it with a type.

Each given type can be manipulated by a set of operations. There are simple types and structured types.

A.1 Simple Types

Simple types are types whose values are primitive, elementary and non-decomposable. In turn, simple types can be classified into two categories:

A. Numeric types:

- ✓ **Integer type**: Integer type: An integer variable can take any signed integer as its value. (Positive or negative). For example: 5,-15, etc.
- ✓ **Real type**: A real-type variable can take the value of any real number, for example: 3.15, -15.5, etc.

B. Symbolic types

- ✓ **Type Character**: includes alphanumeric data, symbols, punctuation, etc., containing a single character, e.g. 'a', '? ', '3', ' ; '. Note that a character is enclosed in single quotation marks.
- ✓ **Logical type (Boolean)** : uses logical expressions. There are only two Boolean values: True and False.

A.2 Structured types

Structured types are any type whose definition refers to other types; i.e., based on simple types. These types are also known as complex or compound types. These types include: arrays, strings, records, trees, chained lists, stacks, files ... all of which will be discussed later.

Example

```
Variable n: integer
r: real
a,b : logical
student_name : string
```

I.7.1.2) The body of the algorithm

A. Basic actions

A.1 Assignment

Assignment is used to assign a value to a variable. In algorithmic terms, it is symbolized by "←".

The sign "←" specifies the direction of the assignment.

Syntax

```
Variable ← Expression
```

Expression can be either:

- Identifier;
- Constant;
- Arithmetic expression;
- Logical expression.

A.2 Semantics

An assignment can be defined in two stages:

- Evaluation of the expression on the right-hand side of the assignment;

- Place this value in the variable.

Example 1

```
0 - Algorithm Calculate
1 - Variable A, B, C, D: integer
2 - Begin
3 -          A ← 10
4 -          B ← 30
5 -          C ← A+B
6 -          D ← C*A
7 - End
```

Note Lines are numbered for ease of explanation.

This can be explained by the following table:

Table 1 – Example 1 Values of variables

Variable	**Line Number**					
	1	2	3	4	5	6
A	?	?	10	10	10	10
B	?	?	?	30	30	30
C	?	?	?	?	40	40

D	?	?	?	?	?	400

Note

Numeric variables are not necessarily initialized to zero. Their value can be anything, which is why the question mark appears before the first value is assigned to the variable.

Example 2

```
0 - Algorithm Logic
1 - Variable A, B, C : logical
2 - Begin
3 -          A ← True
4 -          B ← False
5 -          C ← A and B
6 - End
```

Table 2 – Example 2 Values of variables

Variable	**Line number**				
	1	2	3	4	5
A	?	?	True	True	True
B	?	?	?	False	False
C	?	?	?	?	False

A.3 Reading

The input(read) instruction allows the user to enter data using the keyboard. The value entered will be assigned to a variable.

Syntax

```
Read(identifier)
```

Example

```
Read(A)
Read(A, B, C)
```

The Read(A) instruction allows the user to enter a value from the keyboard. This value will be assigned to variable A.

Note: When the program encounters this instruction, execution stops and waits for the user to type in a value. This value is stored in the designated variable.

A.4 Writing

Before reading a variable, it is advisable to write labels on the screen, to warn the user what to type (otherwise, the user spends his time wondering what the computer expects of him).

The output (write) instruction is used to display information on the screen.

Syntax

```
Write(expression)
```

Expression can be a value, a result, a message, the content of a variable, etc.

Example 1

```
Write(A)
```

This instruction displays the value of variable A on the screen.

Example 2

```
A ← 2
Write ("The value of A is ", A)
```

The last instruction shows on the screen:

The value of A is 2

A.5 Expressions

For a given type, there is a set of operations defined for this type. These operations can be arithmetic, logical or relational:

Table 3 – Types' Operations

Type	Possible operations	Corresponding symbol or word
Integer	Square root	sqrt
	Change of sign	– (unary)
	Addition	+
	Subtraction	-
	Multiplication	*
	Division	/
	Integer division	div
	Modulo (remainder of integer division)	mod
	x exponent y	^
	Comparisons (Relational)	<,=,>,<=,>=,<>
In algorithms, we symbolize integer division by div and the remainder of integer		

division by mod		
Real	Square root	sqrt
	Change of sign	– (unary)
	Addition	+
	Subtraction	-
	Multiplication	*
	Division	/
	Exponent	^
	Comparisons(Relational)	<,=,>,<=,>=,<>
Character	Comparisons	<,=,>,<=,>=,<>
String	Concatenation	+
	Comparisons	<,=,>,<=,>=,<>
Boolean	Logical	And, Or, Not, Xor

When evaluating expressions, operators have the following priorities, in descending order

Table 4 – Operators priorities

unary operators	**– ; not**
multiplicative operators	*** ; / ; div ; mod ; and**
additive operators	**+ ; – ; or**
relational operators	**= ; < ; <= ; > ; >= ; <>**

Expressions in brackets are fully evaluated before being used in further calculations.

Example

5/2 = 2.5

5 div 2 = 2

5 mod 2 = 1

5 ^ 2 = 25

"Bonjour" + " " + "Monsieur" gives "Bonjour Monsieur".

Expression 5>2 is true.

Expression 7<4 is false.

The operations defined for the Boolean type are:

- Logical **AND**
- Logical **OR**
- Logical **NOT**
- **XOR** (exclusive OR called Xor)

We summarize in a truth table the results obtained according to the values of two operations:

Table 5 - Truth table

P	Q	Not P	Not Q	P and Q	P or Q	P xor Q
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

not (P and Q) <=> (not P) or (not Q)

not (P or Q) <=> (not P) and (not Q)

B. Control structures

B.1 Sequencing

An algorithm is made up of a series of instructions, which can be chained together or grouped in different ways.

✓ **Sequence**

The simplest form of instruction flow is the sequence. Instructions are written one after the other, separated by a line break. It's preferable (and essential for some programming languages) that they should all be at the same **indentation** level, i.e. preceded by the same number of spaces.

Indentation is very important for the readability of the algorithm or program. It quickly shows the beginning and end of each **alternative** or **repetitive** instruction, as well as the beginning and end of the algorithm or program.

The instructions in a sequence are all executed in the order in which they are written.

For example, consider the following algorithm, calculating the first three powers of a number:

```
Algorithm Three-power
```

```
Variable x : real
Begin
 Write("Enter the value of x")
 Read(x)
 p2←x*x
 p3←p2*x
 p4←p3*x
 Write (p2, p3, p4)
End
```

The sequence of instructions in this algorithm can be represented by the diagram below:

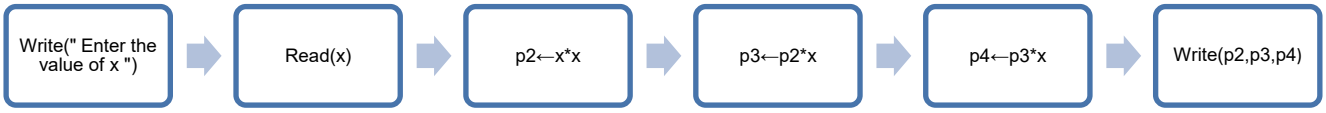

Figure 5 - Sequence of instructions

B.2 Conditional instructions

In some cases, it is necessary to execute different instructions depending on whether or not a condition is met (true or false). In such cases, a conditional sequence is used.

- ✓ **The structure If ... then ... Else**

Syntax

```
If <condition> then
<Instruction1>
Else
<Instruction2>
EndIf
```

The condition must be expressed in such a way as to offer only 2 possible answers: yes or no (true or false).

If the answer is **True**, the sequence of instructions (Instruction1) following the **then** symbol is executed.

If the answer is **False**, the instruction sequence (Instruction2) following the **else** symbol is executed.

When either of the two instruction sequences has been executed, the instruction following the alternative **EndIf** structure is executed in turn.

Note: The **else** <Instruction2> part is optional. It is omitted if you want to execute an instruction only if a condition is true and do nothing if the condition is false.

A series of instructions is a sequence. It is expressed as follows:

```
If <condition> Then
Begin
 Instruction1
 Instruction2        Sequence1
 Instruction3
End
Else
Begin
 Instruction4
 Instruction5        Sequence2
End
EndIf
```

Example

Write an algorithm that displays whether an integer entered on the keyboard is even or odd.

Solution

An integer n is said to be even if the remainder r of the integer division of n by 2 is 0. Otherwise, it is odd.

```
Algorithm Parity
Variable n, r: integer
Begin
 Write ("Enter the value of n: ")
 Read(n)
 r ← n mod 2
 If r = 1 Then
   Write (n, " is odd")
 Else
   Write (n, " is even")
 EndIf
End
```

✓ **Compound conditions**

Some problems require the formulation of conditions that cannot be expressed in the simple form shown above.

For example: the condition "x is included in the interval]10, 20[" is made up of two simple conditions, "x is greater than 10" and "x is less than 20", linked by the logical operator AND.

Example

Write an algorithm that tests whether a note entered on the keyboard is between 0 and 20.

Solution

```
Algorithm TestNote
Variable   Note: real
           Message: String
Begin
 Write ("Enter the note:")
 Read(Note)
 If Note >= 0 And Note <= 20 Then
   Write ("The note ", Note," is correct")
 Else
   Write ("The note ", Note," is incorrect")
 EndIf
End
```

✓ **The structure If ... Else ... ElseIf ... Else**

Syntax

```
If <condition1> Then
 <Instruction1>
ElseIf <condition2> Then
 <Instruction2>
ElseIf <condition3> Then
 <Instruction3>
Else
 <Instruction4>
Endif
```

The instruction that will be executed is the one whose condition is true. If no condition has the value true, the instruction following **Else** will be executed.

Example

Write an algorithm to display the maximum of two numbers entered on the keyboard

Solution

```
Algorithm Max
Variable   A, B: real
Begin
 Write ("Enter the value of A:")
 Read(A)
 Write("Enter the value of B :")
 Read(B)
 If A > B Then
   Write ("The maximum is ", A)
 ElseIf A = B Then
   Write("Equality")
```

```
     Else
       Write ("The maximum is ", B)
     EndIf
End
```

Note

A conditional structure can in turn contain another conditional structure, i.e. one within the other. In this case, the instructions to be executed after **Then** and/or **Else** are, in turn, **If...Then...Else** instructions.

Rule

The else always belongs to the nearest if that does not already have another else.

Example

Write an algorithm that reads 3 integer values A, B, C and prints the largest.

Read data;

Find the largest value by pairwise comparison;

Display result.

Solution

```
    Algorithm maximum
    Variable A,B,C,max :Integer
    Begin
     Read (A, B, C
     If A>B Then
       If A>C Then
         max←A
       Else
         max←C
       EndIf
     Else
       If B>C Then
         max←B
       Else
         max←C
       EndIf
     EndIf
     Write ("The greatest value is : ", max)
    End
```

B.3 Repetitive operation

The principle is to repeat a set of operations and stop the repetition according to a condition. This is called iterating the code, or simply looping.

- ✓ **For loop**

The For loop is a bounded iteration. It is generally associated with a control variable so that the code is not executed indefinitely. In practice, it's only used if you know how many times you need to iterate.

The role of the counter variable is to control the number of repetitions.

The variable is of integer type. It is initialized to the initial value. The counter automatically increases its value by one (1) at each loop turn, until it reaches its final value.

Syntax

```
For counter ← initial_value to final_value Do
<Instruction1>
EndFor
```

For each value taken by the counter variable, the list of instructions is executed.

When the counter variable reaches its final value, processing is executed one last time and the program exits the loop.

Example

Write an algorithm that displays the multiplication table of 7 on the screen.

Solution

```
Algorithm multiplication_table
Variable k: integer
Begin
 For k  ← 0 to 10 Do
   Write ("7 * ",k ," = ", 7*k)
 EndFor
End
```

✓ **While loop**

Execution of the loop depends on the value of the condition. If it's true, the program executes the instructions that follow, until it reaches the EndWhile line. It then returns to the While line, performs the same examination, and so on.

Syntax

```
While condition   Do
<Instruction1>
EndWhile
```

The loop only stops when the condition takes the value false, in which case the program moves on to the instruction following EndWhile.

Example

Write an Algorithm that displays numbers from 1 to 10 on the screen:

Solution

```
Algorithm counter
Variabl cpt: integer
Begin
 cpt←1
 While cpt<=10 Do
   Write(cpt)
   cpt←cpt+1
 EndWhile
End
```

✓ **Repeat loop**

This loop is used to repeat an instruction until a condition is true.

Note

This loop is generally only used for menus, and is dangerous because there is no condition check before entering!

Syntax

```
Repeat
    <Instruction1>
Until condition
```

The instruction list is executed, then the condition is evaluated. If it is false, the loop body is executed again, the condition is re-evaluated and if it is true, the program exits the loop and executes the instruction following **Until**.

Remarks

- The "Repeat" and "While" loops are used when you don't know how many times you'll need to execute these loops.
- Unlike the "While" loop, the "Repeat" loop is executed at least once.
- The stop condition of the "Repeat" loop is the negation of the continue condition of the "While" loop.
- The "For" loop is used when the number of iterations (repetitions) is known in advance.

Example

Using the Repeat ... Until loop, write an algorithm that calculates the sum of the first N integers. N is assumed to be strictly positive.

For example, if N = 6, the program must calculate: 1 + 2 + 3 + 4 + 5 + 6 = 21

Solution

```
Algorithm Sum
Variable S, Cnt, N: Integer
Begin
```

```
    Write ("Enter a strictly positive value:")
    Read(N)
    S ← 0
    Cnt ← 1
    Repeat
      S ← S + Cnt
      Cnt ← Cnt +1
    Until Cnt > N
    Write ("The sum of the first", N, " integers is: ", S)
   End
```

Chapter II. Elements of a programming language - C++

II.1) Program structure

We'll start with a very simple program written in C++. The purpose of this example is to introduce some basic rules governing the language.

We want this program to display a short text on the screen. Suppose the text says:

```
It’s me, your first program.
```

So our first program will perform the following steps:

1. **Start;**
2. **Display text on screen;**
3. **Stop**

Let's take a close look at each line of the program and discover its meaning and purpose.

II.1.1) #include

```
#include <iostream>
using namespace std;
int main()
{
  cout << "It's me, your first program.";
  return 0;
}
```

Note the # character (pound sign) at the beginning of the first line. This means that the content of this line is the so-called **preprocessor directive** - for the moment, we'll just say that it's a separate part of the compiler whose task is to pre-read the program text and make a few modifications. The prefix "pre" suggests that these operations are carried out before the complete processing (compilation) takes place.

The changes introduced by **the preprocessor** are entirely controlled by its directives. In our program, we deal with the include directive. When the preprocessor encounters this directive, it replaces the directive with the contents of the file whose name is listed in the directive (in our case, the file named iostream). The changes made by the preprocessor do not in any way modify the contents of your source file. All modifications

are made to a volatile copy of your program, which disappears immediately after the compiler has finished its work.

So why does the preprocessor need to include the contents of a completely unknown file called iostream? Writing a program is similar to building with ready-made blocks. In our program, we're going to use one such block and it will occur when we want to write something to the screen. This block is called cout, but the compiler doesn't know about it yet. The compiler must be notified of this.

The compiler needs header files. These files contain a collection of preliminary information about ready-to-use blocks that can be used by a program to write text on the screen or read letters from the keyboard. So, when our program is going to write something, it will use a block called cout. The compiler developers place a set of this anticipated information in the iostream file. All we have to do is use the file. This is exactly what we expect from the include directive.

II.1.2) Using namespace

In the C++ language, all elements of the standard C++ infrastructure are declared in a namespace called std. A namespace is a container or abstract environment created to hold a logical grouping of unique entities (blocks). An entity defined in a namespace is associated only with that namespace. If you wish to use many standard C++ entities, you must insert the using namespace statement at the top of each file, outside any function.

The instruction must specify the name of the desired namespace (std in our case). This will make standard facilities available throughout the program.

If you don't use **using namespace**, you must inform the compiler where the cout block comes from, and you must do so every time you use one of the entities derived from the std namespace. In other words, you must write:

```
    std::cout
```

instead of:

```
    cout
```

II.1.3) the main function

The C++ language standard assumes that among the many different blocks that can be placed in a program, one specific block must always be present, otherwise the program will not be correct. This block is always a function with the same name: main.

Every function in C++ begins with the following set of information:

- The result of the function;
- The function name;
- How many arguments the function accepts and what their names are.

Let's take a close look at our program and try to read it correctly.

- The function's result is an integer value (it can be read from the word int, which is short for integer).
- The function name is main (we already know why)
- The function requires no arguments (which can be deduced from the absence of anything in brackets).

This set of information is sometimes called a prototype, and is like a label affixed to a function announcing how you can use this function in your program. The prototype says nothing about what the function is supposed to do. How it works is written inside the function, also known as the function body. The function body begins with the first opening parenthesis { and ends with the corresponding closing parenthesis }. If the function body is empty, this means that the function does nothing.

The result of the function can be `void`. The word void in front of the function name (main) tells the compiler that the function provides no result.

In the body of the main function, we have to write what our function (and therefore the program) is supposed to do. If we look inside, we find a reference to a block called `cout`.

First of all, note the **semicolon** at the end of the line. Every **instruction** in C++ must end with a semicolon - without it, the program will be incorrect.

This particular instruction says: ask the entity named `cout` to display the following text on the screen (as indicated by the `<<` symbol, which specifies the direction in which the text is sent).

The cout entity must be fed with something that is intended to be displayed on the screen. In our example, the feed is just text (string). Program strings in C++ are always enclosed in quotation marks - this way, the compiler recognizes the text sent to the program user, and the text intended for compiling is translated into machine language. This distinction is very important.

```
    int main();
```

The line above is the prototype of the main function (i.e. a hint indicating what we can expect from a particular function).

```
    "int main();"
```

The line above is not the prototype of the main function, but a string that only looks like part of a source code. The compiler isn't interested in what's between the quotation marks, so it doesn't recognize these strings as code.

In C++, you don't have to write a single instruction per line. You can place two (or more) statements on the same line, or divide a statement into several lines, but bear in mind that readability (for humans) is a very important factor.

```
    cout
    <<
    "It's me, your first program."
```

```
;
```

II.1.4) return

Only one line remains in our program.

```
return 0;
```

Its name is just return and that's what it does. When used inside a function, it causes the function's execution to end. If you perform return somewhere inside a function, that function immediately stops execution.

The zero after the word return is the result of your main function. This is important - it's how your program tells the operating system: I did what I had to do, nothing stopped me from doing it, and everything's fine.

If you were to write:

```
return 1;
```

This would mean that something went wrong, that your program didn't run successfully and the operating system could use this information to react appropriately.

In the main function, return is not **mandatory**. In this case, the compiler assumes that return 0 has been implicitly used.

In summary

- A C++ program is made up of instructions, which are the commands you give to the computer.
- These instructions are grouped together in blocks called functions. We've introduced the main function (the one that starts every C++ program) into our program - it will be executed when you start the program;
- We've used the standard library for the first time, with the iostream file, to display a message on the screen using the cout function.
- The program terminates immediately after printing, indicating that everything you expected has been achieved.

II.2) The program environment

II.2.1) Variables

The C++ language lets you write numbers and perform arithmetic operations with them: add, subtract, multiply and divide. The results of these operations are stored in **"containers"** called variables.

During program execution, each variable is attached to an address: which indicates where in memory the value of this variable is stored.

Each variable is characterized by:

- **A name** (identifier);
- **A type** (specifying the nature of the variable (integer, character, object, etc.);
- **A value** (which can be modified at any time);

II.2.1.1) Identifier

When naming a variable, it's important to follow a few strict rules:

- The variable name must be composed of upper- or lower-case Latin letters, numbers and the _ character (underscore);
- The variable name must begin with a letter;
- The underscore character is a letter;
- Uppercase and lowercase letters are considered different. Cube and CUBE are two different variable names, therefore two different variables);

The same restrictions apply to all entity names used in C++ (files, functions, etc.).

The C++ language standard imposes no restrictions on the length of variable names, but a specific compiler may have a different opinion on this subject.

Here are some correct, but not always practical, variable names:

- variable
- i
- t10
- Exchange_Rate
- counter
- DaysToTheEndOfTheWorld
- TheNameOfAVariableWhichIsSoLongThatYouWillNotBeAbleToWriteItWithoutMistakes
- _

Now some incorrect names:

- 10t (doesn't start with a letter)
- Adiós_señor (contains illegal characters)
- Exchange Rate (contains a space)

II.2.1.2) Types

The type is an attribute that uniquely defines the values that can be stored in the variable. Computers use the binary system to store numbers and perform any operation on them.

The numbers managed by modern computers are of two types:

- **Integers, i.e. those without the fractional part;**
- **Floating-point numbers**, which contain (or can contain) the fractional part.

These two types of numbers differ considerably in the way they are stored in a computer's memory and in the range of acceptable values. These characteristics determine the type.

So we now know two types from the C++ language: an integer type (called **int**) and a floating-point type (called **float**).

II.2.1.3) Declaration

In C++, all variables must be declared before they can be used.

General form of a declaration:

```
<type> <list of variables>;
```

Where:

<type> is a type or class name,

<list of variables> is one or more variable names, separated by commas.

The entire statement ends with a semicolon.

Examples

Declaration of the variable counter of type int.

```
int counter;
```

Declaration of three variables of type int named (respectively) variable_1, account_balance and factures.

```
int variable_1, account_balance, bills ;
```

We are allowed to use as many variable declarations as necessary.

And how do you assign a value to the newly declared variable? You need to use the **assignment operator.**

```
=
```

Let's look at some examples

```
counter = 1 ;
```

The instruction above says: assign a value of 1 to a variable named counter. Or counter **receives 1.**

We are allowed and even encouraged to place the variable declaration and its first assignment in the same place. Example:

```
int variable;
variable = 1 ;
```

Can be compacted in the following form:

```
int variable = 1 ;
```

This solution has a number of important advantages. The most important is that declaring a variable with immediate assignment of its initial value is **less error-prone** than performing these two activities in two different places. The code is more readable, the programmer's intentions are easier to recognize, and if you're used to declaring and initializing a variable in a single step, you minimize the risk of using an uninitialized variable. This is why we prefer such declarations in our code.

In particular, you can declare a constant by adding **const** in front of the type name, for example:

```
const float PI = 3.14159265358979323846;
const int MAX = 100;
```

Constant values cannot be modified.

The part of the declaration to the right of the = sign is called an **initializer**

The **initializer** can be a more complex expression, like the one below.

```
int twosquared = 2 * 2;
```

Example

```
result = 100 + 200 ;
```

In this case, the new value of the result variable will be the result of adding 100 to 200.

```
x = x + 1 ;
```

In the "C++" language, the "=" sign doesn't mean is equal to, but **assigns** a value.

In effect, the value of variable x has been incremented by one, which has nothing to do with comparing the variable to any value.

II.2.2) Numbers and how computers see them

II.2.2.1) Integers

Using single quotation marks as separators improves the readability of integers. Take, for example, the number eleven million one hundred and eleven thousand one hundred and eleven. If you want your number to be extremely legible - you can write it as follows:

```
11'111'111
```

To code negative numbers in C++, simply add a minus.

```
-11'111'111
```

Positive numbers do not need to be preceded by a plus sign, but you can do so if you wish. The following lines describe the same number:

```
+123
```

```
123
```

There are three additional conventions, unknown to the mathematical world.

1. **The first** of these allows numbers to be used in the octal system. If a whole number is preceded by the digit 0, it will be treated as an octal value. This means that the number must contain digits between 0 and 7 only.

```
0123
```

Is an octal number with a (decimal) value equal to 83.

2. **The second** allows us to use hexadecimal numbers. This number must be preceded by the prefix 0x or 0X.

0x123

Is a hexadecimal number whose (decimal) value is equal to 291.

3. **The third** allows you to enter binary numbers preceded by the prefix 0b or 0B.

```
0B1111
```

Is a binary number equal to the decimal value 15.

II.2.2.2) Floating-point numbers

One of the basic types known in the C++ language is the integer type named int. Let's talk about another type, designed to represent and store numbers that have a non-empty decimal fraction.

Taking "Two and a half" as an example, you need to make sure that your number contains dots and not commas. The compiler won't accept it, as the comma itself has its own reserved meaning in the C++ language.

If you wish to use a value of only "two and a half", you must write it as shown here:

```
    2.5
```

Note again, there's a period between "2" and "5", not a comma.

You can write the value of "zero point four" in C++ as :

```
    .4
```

You can omit zero when it's the only digit before or after the decimal point. In essence, you can write the value 0.4 as on the right.

For example: the value of 4.0 could be written as 4. Without changing its type or value.

Note: the decimal point is essential for recognizing floating-point numbers in C++. Take a look at these two numbers:

```
    4
    4.0
```

The C++ compiler sees them completely differently:

`4` is an int.

`4.0` is a float.

When you want to use very large or very small numbers, you can use so-called scientific notation. Take, for example, the speed of light expressed in meters per second. Written directly, it would look like this

```
    300000000.
```

As you can probably imagine, you are also allowed to write the value in a more readable form:

```
300'000'000.
```

Because single quotes can be used inside floats too.

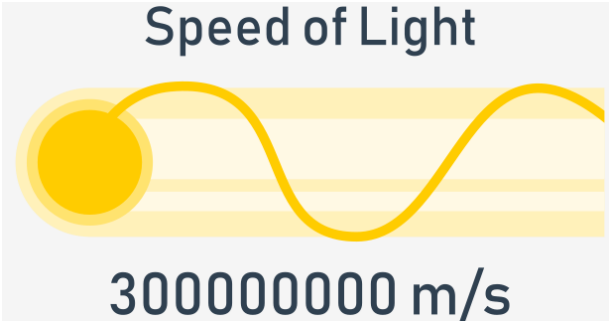

Figure 6 - Speed of light

To avoid writing so many zeros, we can use the form:

$3 \cdot 10^8$

It reads as follows: "three times ten to the power of eight".

In the C++ language, the same effect is obtained in a slightly different form - look at:

```
3E8
```

The letter E (you can also use the lower-case letter e - it comes from the word exponent) is a concise version of the expression "times ten to the power of".

Note:

The exponent (the value after the "E") must be an integer.

The base (the value before of the "E") may or may not be an integer.

This convention is also used to store very small numbers. For example, the value:

```
6.62607 x 10-34
```

In a program, we'll write it like this:

```
6.62607E-34
```

Declaring a floating-point variable is done as follows:

We do this using the float keyword. Knowing that we can declare two floating-point variables, named PI and Champ :

```
float PI, Champ;
```

As you can see, the difference in the declaration of integer and float variables is quite small from a syntactical point of view.

From a semantic point of view, however, it's quite significant. With a little thought, we can understand that the symbol (or more precisely - the operator) that performs the mathematical division is a single character / (slash).

```
int i = 10 / 4;
float x = 10.0 / 4.0 ;
```

It may seem surprising, but this statement gives i the value 2, while x takes the value 2.5.

It is possible to convert integer values to float values or vice versa, but this may result in a loss of precision. Consider the example below:

```
int i = 100 ;
float f = i;
```

After switching from int to float, the value of variable f is 100.0, because the int value (100) is automatically converted to float (100.0).

The transformation affects the internal (machine) representation of these values, as computers use different methods to store floats and integers in their memory.

Now let's consider the **opposite** situation.

As you've probably guessed, these substitutions will result in a loss of precision - the value of variable i will be (100) because 0.25 makes no sense in the world of ints. What's more, converting a float to an int is not always feasible.

```
float f = 100.25 ;
int i = f;
```

Integer variables (like floats) have a limited capacity. They cannot hold arbitrarily large (or arbitrarily small) numbers.

For example, if a certain type of computer uses four bytes (i.e. 32 bits) to store integer values, you can only use numbers between -2147483648 and 2147483647.

Consider the following code:

```
float f = 1E10;
int i = f;
```

Variable i is unable to store such a large value, during assignment a loss of precision will occur, but the value assigned to variable i is not known in advance.

In some systems, it may be the maximum permissible int value, while in others, an error will occur, and in others, the assigned value may be completely random.

This is what we call an implementation-dependent problem. This is the second face of software portability.

II.3) The body of the program

II.3.1) Comments

The developer may wish to add a few words addressed not to the compiler but to humans, usually to explain to other readers of the code how tricks used in the code work, or the roles of variables and functions, and possibly, to store information about who the author is and when the program was written.

Good practice dictates describing each important newly-created entity; in particular, explaining the role of parameters and values modified or returned as results, as well as what the code actually does.

This is done by inserting remarks into the program, which are called comments.

Whenever the compiler encounters a comment in your program, the comment is completely transparent to it.

C++ supports two ways of inserting comments:

1. **Single-line** comments

```
// comments
```

A **single-line comment** ignores everything from the location of the pair of slashes (//) to the end of the same line.

Here's an example in which everything from the pair of slashes onwards is ignored by the compiler. The single-line comment can start anywhere on the line. It could also be an empty line, with no content at all.

```
int counter; //counts the number of sheep in the meadow
```

2. **A block** comment (also known as C-style comments, as they have been used since the very beginning of the C programming language, the ancestor of C++).

```
/* comments */
```

In the C++ language, a block comment is text that begins with a pair of the following characters:

```
/*
```

and ends with a pair of the following characters:

```
*/
```

The comment may extend over several lines, or may occupy a single line or part of a line.

Here's another example of a similar comment

```
    /* the counter variable counts the number of sheep in the
meadow */
    int counter;
```

The developer will read the code more quickly and it will take less time to understand it.

Developers often place a note at the beginning of the code indicating who modified it and why. The note might look something like this:

```
    /*************************************************************
***
        Counting A version 2.0
```

```
        Author: fromy link 2015
        email: sam@newpipe.org

        Changes:
        2015-06-11: fromy: A site second access
    ************************************************************
***/
```

The condition indicating how the comment should be started and ended is fully met.

When testing, it's common to use comments to mark a piece of code you don't currently need for some reason. Like isolating the place where an error might be hidden.

II.3.2) Mathematical operators

An **operator** is a programming language symbol that can operate on values. For example, an assignment operator is the = sign. We already know that it can assign values to variables.

Now let's take a look at some of the other operators available in the C++ language, and find out what rules govern their use and how to interpret the operations they perform. Let's start with the operators associated with the widely recognized arithmetic operations.

II.3.2.1) Multiplication

An asterisk * is a **multiplication operator**. If you take a look at the code here, you'll see that variable k will receive the value 120, while variable z will receive 0.625.

```
    int i = 10, j = 12;
    float x = 1.25, y = 0.5 ;

    int k = i * j;
    float z = x * y ;
```

II.3.2.2) Division

A slash ("/") is a **division** operator. The value in front of the slash is a **dividend**, the value behind the slash a **divisor**. Look at the program below: of course, k will be set to 2, z to 0.5.

```
    int i = 10, j = 5;
    float x = 1.0, y = 2.0 ;

    int k = i / j;
    float z = x / y ;
```

Note: division between two integers gives the integer quotient.

Division by zero

Dividing by zero is strictly forbidden. As a general rule, you must not divide by zero.

```
    float x = 1.0 / 0.0 ;
```

In the following example, the compiler won't tell you anything, but when you try to execute this code, the result of the operation can be surprising. It's not a number. It's a special value called inf (as in infinity).

```
float x = 0.0 ;
float y = 1.0 / x ;
```

II.3.2.3) Addition

The addition operator is the plus sign "+". In this example as you may expect, k is equal to 102 and z to 1.02.

```
int i = 100, j = 2;
float x = 1.0, y = 0.02 ;

int k = i + j;
float z = x + y ;
```

II.3.2.4) Substraction

The **subtraction** operator is obviously the "-" (minus) sign, although you should note that this operator also has another meaning: it can change the sign of a number. This is a good time to show you a very important distinction between **unary** and **binary** operators.

As usual, let's familiarize ourselves with an example - you can guess that k will be equal to -100, while z will be equal to 0.0.

```
int i = 100, j = 200 ;
float x = 1.0, y = 1.0 ;

int k = i - j;
float z = x - y ;
```

II.3.2.5) Remainder of the division

The remainder operator is quite special, as it has no equivalent among traditional arithmetic operators.

Its graphical representation in the C++ language is the % (percentage) character.

% is a binary operator (it performs the modulo operation) and the two arguments cannot be floats. Take a look at the example:

```
int i = 13 ;
int j = 5;
int k = i % j;
```

The variable k takes the value 3 (since 2 * 5 + 3 = 13).

II.3.2.6) Unary operators

A. Unary minus

In "subtraction" applications, the minus operator expects two arguments. For this reason, the subtraction operator is considered one of the binary operators, just like addition, multiplication and division. But the minus operator can be used in a different way.

```
int i = -100 ;
int j = -i;
```

As you've probably guessed, variable j will be assigned the value 100. We've used the minus operator as a unary operator, as it expects only one argument.

B. Unary plus

Similarly, the "+" operator can be used as a unary operator.

```
int i = 100 ;
int j = +i;
```

Although such a construction is syntactically correct, its use doesn't make much sense and it would be difficult to find a good justification for it.

II.3.2.7) Priorities

Up to now, we've treated each operator as if it had no connection with the others. In programming, we very often find more than one operator in an expression, and things can get very complicated very quickly. Consider the following expression:

```
2 + 3 * 5
```

Remember that multiplication precedes addition. You'll probably remember that you need to multiply 3 by 5, store the 15 in your memory, add it to 2 and get the result of 17.

The phenomenon that causes some operators to act before others is known as the priority hierarchy. The C++ language precisely defines the priorities of all operators, and assumes that operators with higher priority perform their operations before operators with lower priority.

So, if we know that * has a higher priority than +, we can understand how the final result will be calculated.

Table 6 - Operator precedence

+ - unary
* / %
+ - binary

II.3.2.8) Evaluation direction of an expression

The direction determines the order of execution performed by certain operators of equal priority, placed side by side in an expression.

In C++, expressions are evaluated from left to right, which means that the calculation of this example expression is performed from left to right, i.e. 3 will be added to 2 and 5 will be added to the result.

```
2 + 3 + 5
```

Try to evaluate this expression:

```
2 * 3 % 5
```

Both operators (* and %) have the same priority, so the result can only be guessed when you know the direction of the evaluation.

The result is 1.

II.3.2.9) Parentheses

We are always allowed to use parentheses, which can change the natural order of calculation. As with arithmetic rules, sub-expressions within parentheses are always calculated first. You can use as many parentheses as you like, and we often use them to improve the readability of an expression.

An example of an expression involving several parentheses is shown below. Try calculating the value given to the variable l.

```
int i = 100 ;
int j = 25;
int k = 13;
int l = (5 * ((j % k) + i) / (2 * k)) / 2;
```

The result is `10`.

II.3.3) Incrementing and decrementing (++ --)

Here are a few operators in the C++ language. Some of them are frequently used to increment a variable by one. This is often done when we're counting something. Consider the following:

```
int counter = 0;
counter = counter + 1;
```

Similar operations appear very frequently in typical programs, so the creators of the C++ language introduced a set of special operators for these actions.

II.3.3.1) "++" operators (incrementing)

A. Prefixed form

In its prefixed form, the theory is that the associated variable is first incremented and then returned.

```
    ++counter;
```

B. Suffixed form

In its suffixed form, the theory is that the associated variable is first returned and then incremented.

```
    counter++;
```

II.3.3.2) "--" operators (decrementing)

A. Prefixed form

In its prefixed form, the theory is that the associated variable is first decremented and then returned.

```
    --days_until_holiday;
```

B. Suffixed form

In its suffixed form, the theory is that the associated variable is first returned and then decremented.

```
    days_until_holiday--;
```

Example

```
    int i = 1;
    int j = i++;
```

First, the variable i is given the value 1. In the second instruction, we'll see the following steps:

The value of i will be taken (as we're using the suffixed incrementing form);

Next, variable i will be increased by 1.

In the end, j will receive the value 1 and i the value 2.

Things work a little differently here.

```
    int i = 1;
    int j = ++i;
```

Variable i is set to 1. In the second instruction, we'll take the following steps:

Variable i is incremented to 2 (as we're using the prefixed form of incrementing);

The increased value is then assigned to variable j.

In the end, both i and j will be equal to 2.

Take a close look at this program. Let's retrace its execution step by step.

```
    int i = 4;
    int j = 2 * i++;
    i = 2 * --j;
```

At the end, variable i is given the value 14;

We take the original value of i (4), multiply it by 2, assign the result (8) to j and then (suffixed incrementing) increment variable i (now worth 5);

We decrement (prefixed decrementing) the value of j (it's now 7); this reduced value is taken and multiplied by 2 and the result (14) is assigned to variable i.

These operators have a higher priority than the *,/ and % operators. The order of execution then depends on whether the operator is prefixed (left to right) or suffixed (right to left).

Our priority table is now as follows:

Table 7 - Operator precedence

++ -- + - unaire
* / %
+ - binary

II.3.4) Assignment operators

II.3.4.1) Simple assignment operator

The simple assignment operator "=" means "assign to".

<Destination> = <Source>

<Source> and <Destination> must be of the same kind, <Source> is an expression and <Destination> is a variable;

```
int x, y;
x = 5; //x = 5
y = 3; //y = 3
x = y; //x = 3
```

II.3.4.2) Compound assignment operators

In the C++ language, there's a short way of writing arithmetic operations.

Compound assignment operators modify the current value of a variable by performing an operation on it.

A. The "+=" add assignment operator

It means "assign by adding to".

<Destination> += <Source>; Where <Source> and <Destination> must be of the same kind, <Source> is an expression and <Destination> is a variable;

```
int x, y;
x = 5;
y = 3;
x += y; //x = 8
```

Equivalent

```
x=x+y;
```

B. The "-=" subtractor assignment operator

It means "assign by subtracting from".

<Destination> -= <Source>; Where <Source> and <Destination> must be of the same kind, <Source> is an expression and <Destination> is a variable;

```
int x, y;
x = 5;
y = 3;
x -= y; //x = 2
```

Equivalent

```
x=x-y;
```

C. The "=" multiplier assignment operator*

It means "assign by multiplying to".

<Destination> *= <Source>; Where <Destination> and the result of the operation must be of the same kind, <Source> is an expression and <Destination> is a variable;

```
int x, y;
x = 5;
y = 3;
x *= y; //x = 15
```

Equivalent

```
x=x*y;
```

D. The "/=" divisor assignment operator

It means "assign by dividing to".

<Destination> /= <Source>; Where <Destination> and the result of the operation must be of the same kind, <Source> is an expression and <Destination> is a variable;

```
int x, y;
x = 5;
y = 3;
x /= y; //x = 1 (See the "/" operator)
```

Equivalent

```
x=x/y;
```

E. The modulo assignment operator "%=".

It means "assign the modulo of, to".

<Destination> %= <Source>; Where <Destination> and the result of the operation must be of the same kind, <Source> is an expression and <Destination> is a variable;

```
int x, y;
x = 5;
y = 3;
x %= y; //x = 2 (See the "%" operator)
```

Equivalent

```
x=x%y ;
```

II.3.5) Relational and comparison operators

The computer executes a program and provides answers to any questions it may contain. The program must be able to react according to the answers it receives. Fortunately, computers only know two types of answer: yes, it's true, or no, it's false.

To ask questions, the C++ language uses a very special set of operators.

Table 8 - Relational operators

Operator	Description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater or equal to

Let's take a look at some simple examples.

x and y can be variables, values or even expressions.

Question: is x equal to y?

To ask this question, you use the == (equal equals) operator.

Note

= is an assignment operator

== is the question "Are these values equal?"

This is a binary operator. It needs two arguments and checks whether they are equal.

Now let's ask some questions. Try to guess the answers.

2 == 2;

This is a simple question. 2 is equal to 2. The computer will answer **true.**

1 == 2;

The answer will be **false.**

i == 0;

To answer this question, you need to know the value of variable i at the time of comparison. If the variable has been modified several times during the execution of your program, the answer to this question can only be given by the computer and only at runtime.

x == 2 * y;

Due to the low priority of the == operator, this question should be treated as equivalent to this one:

x == (2*y);

Question: Is x not equal to y?

To ask this question, we use != (equal exclamation). It's also a binary operator and has the same priority as ==. Let's imagine we want to ask whether the number of days left until the vacations is currently not equal to zero:

```
    days_until_holidays != 0 ;
```

Question: is x greater than y?

You can ask this question using the > (greater than) operator. If you want to know if one value is greater than another, you can write it as follows.

x > y

The true answer confirms the statement; the false answer denies it.

Question: is x greater than or equal to y?

The "greater than" operator has another special, non-strict variant, but it's noted differently in classic arithmetic notation: >= (greater than or equal to). There are two signs, not one.

x>=y

Question: is x less than y?

You can ask this question using the < (less than) operator. If you want to know whether a value is less than another value, you can write it as follows.

```
    current_speed < 110
```

Question: is x less than or equal to y?

The operator we use in this case is: <= (less than or equal to).

```
    Current_speed <= 110
```

II.3.5.1) Boolean type

How do we use the answer we got?

The C++ language defines a very special data type designed to store the values resulting from one of the comparisons presented above. The type is called: **bool**

Its name comes from the famous English mathematician George Bool (1815 -1864), the creator of Boolean algebra.

Here are a few characteristics of the Boolean type:

- It's actually an integer type, using the minimal storage offered by a computer (usually a byte, as in the char type);
- It can be assigned integer values (but its maximum value is always 1);
- It can take part in all integer evaluations, and when used in this way behaves like an ordinary int;

There are two literals representing two possible Boolean values:

1. **false (Boolean equivalent of integer 0);**
2. **true (Boolean equivalent of integer 1);**

Although it's worth mentioning here that any non-zero integer value is interpreted as true by the C++ language).

To use this type, we need to store the comparison results in a variable and use them later. How do we do this? Well, we'd use a bool variable, like this one:

```
int val1=0,val=0 ;
bool answer = val1 >= val2 ;
```

If the answer is **true** because val1 is greater than or equal to val2, the computer will assign **true** to the answer (although if you take a look inside the variable, you'll find an integer 1). If val1 is less than val2, the answer variable will be assigned **false** (actually 0).

We can use the response to make a decision about the future of our program. We use a special instruction for this purpose, and we'll tell you what it is shortly.

We now need to update our priority table.

Table 9 - Operator precedence

++ -- + - unaire
* / %
+ - binaire
< <= > >=
== !=
= += -= *= /= %=

II.3.6) Character type

So far, we've treated the C++ language (and the computer itself) as a tool for performing calculations on numbers. The computer can also easily be used for word processing.

A word can be defined as a string of characters (letters, numbers, punctuation marks, etc.).

Now, however, we'll ignore the multi-character string and focus our attention on single characters. We'll come back to the problem of string handling when we start working with arrays, because in the C++ language, strings behave in a very similar way.

To store and manipulate characters, the C++ language provides a special data type. This type is called char, which is an abbreviation of the word character and according to Bjarne Stroustrup should be pronounced as "tchar", not "kar".

Let's try declaring a variable to store a single character.

```
    char character;
```

Now let's talk a little about how computers store characters.

II.3.6.1) ASCII code

Computers store characters as numbers. Each character used by a computer corresponds to a unique number, and vice versa. Some of these characters are called white spaces. An example of white space that is completely invisible to the naked eye is a special code used to mark the end of lines in text files. People don't see this sign (or these signs), but they can see their effect where lines are broken.

https://www.lookuptables.com/text/ascii-table

II.3.6.2) Character type values

How do we use char values in the C++ language? We can do so in two ways, which are not quite equivalent.

The first allows us to specify the character itself, but surrounded by single quotes (apostrophes). Suppose we want a variable to receive the value of the capital letter A.

We proceed as follows:

```
    character = 'A';
```

Now let's assign an asterisk * to our variable. We proceed as follows:

```
    character = '*';
```

The second method consists in assigning a non-negative integer value, which is the code of the desired character. This means that the assignment below will put an "A" in the character variable.

```
    character = 65 ;
```

The second solution, however, is less recommended and requires knowledge of the

ASCII code values.

II.3.6.3) Literal

A literal is a symbol that uniquely identifies its value.

x: this is not a literal; it's probably a variable name; when you look at it, you can't guess what value is currently assigned to this variable;

'A': this is a literal; when you look at it, you can immediately guess its value; you even know that it's a char literal;

100: also a literal (of type int);

100.0: this is another literal, this time of floating-point type;

i + 100: this is a combination of a variable and a literal joined by the + operator; this structure is called an expression.

If a char literal is given as a character surrounded by apostrophes, how do we code the apostrophe itself?

The C++ language uses a special convention that extends to other characters, not just apostrophes. But let's start with an apostrophe. An apostrophe looks like this:

```
character = '\'';
```

The \ character (called backslash) acts as an escape character, because by using \ we can escape the normal meaning of the character that follows the backslash. In this example, we escape the usual role of the apostrophe (i.e. delimiting char literals).

II.3.7) Input and output

Now we're going to look at two important and extremely useful features that we use to provide connectivity between the computer and the outside world.

Sending data from human (user) to the computer program is called input. The flow of data transferred in the opposite direction, i.e. from the computer to the human, is called output.

II.3.7.1) C++ output

In C++, cout sends formatted output to standard output devices, such as the screen. We use cout with the << operator to display the output.

cout only needs the variable name.

```
#include <iostream>
using namespace std;

int main() {
    cout << "This is a C++ program";
    return 0;
}
```

- First, we include the iostream header file, which allows us to display the output.

- cout is defined in the std namespace. To use the std namespace, we use the declaration using namespace std ;
- Every C++ program begins with the main() function. Code execution begins at the start of the main() function.
- cout prints the string in quotation marks " ". It is followed by the << operator.
- return 0; is the "exit state" of the main() function. The program ends with this instruction, **but it is not mandatory.**

To print numbers and characters, we use cout, but without quotation marks.

```
    #include <iostream>
    using namespace std;
    int main() {
        int num1 = 70;
        float num2 = 256.783;
        char ch = 'A';

        cout << num1 << endl;    // print the int
        cout << num2 << endl;    // print the float
        cout << "character: " << ch << endl;     // print the
char
        return 0;
    }
```

Remarks:

- endl is used to insert a new line. This is why each output is displayed in a new line.
- The << operator can be used several times if we want to print different variables, strings, etc. in a single statement. For example :

```
    cout << "character: " << ch << endl;
```

The **cout** statement can also be used with some member functions:

cout.precision(int n): Sets the decimal precision to N, when using float values.

```
    #include <iostream>
    using namespace std;
    int main()
    {
       double pi = 3.14159783;
       // Set precision to 5
       cout.precision(5);
       // Print pi
       cout << pi << endl;
       // Set precision to 7
```

```
    cout.precision(7);
    // Print pi
    cout << pi << endl;
    return 0;
}
```

Output:

3.1416

3.141598

II.3.7.2) C++ input

In C++, cin takes formatted input from standard input devices such as the keyboard. We use cin with the >> operator for input.

```
#include <iostream>
using namespace std;

int main() {
    int num;
    cout << "Enter an integer value : ";
    cin >> num;   // Number input
    cout << "The number is " << num;
    return 0;
}
```

The input is stored in the variable num. We use the >> operator with cin to enter the input.

The >> operator can be used several times if we want to read different variables in a single instruction. For example:

```
#include <iostream>
using namespace std;
int main() {
    char a;
    int num;
    cout << "Enter a char and a number: ";
    cin >> a >> num;
    cout << "character : " << a << endl;
    cout << "number : " << num;

    return 0;
}
```

II.3.8) Conditions and conditional execution (if)

We now know how to ask a question (with comparison operators), but we still need to know how to use answers. We need a mechanism that lets us do something if a condition is met (it evaluates to true), and not do it if it's not.

To make these decisions, the C++ language has a special instruction. Because of its nature and application, it's called a conditional statement.

There are several variants of the conditional statement. We'll start with the simplest and gradually move on to the more difficult ones.

II.3.8.1) The if statement

```
if (true_or_false) do_if_true;
```

This conditional statement includes the following elements, strictly necessary, and in that order only:

- **if** keyword ;
- Left (opening) parenthesis;
- An expression (a **question** or an **answer**) whose value will be interpreted only in terms of true (when its value is non-zero) and false (when it is equal to zero) ;
- Right (closing) parenthesis ;
- **An instruction** (**only one**, but we'll learn how to manage this limitation).

How does this statement work?

- If the expression true_or_false in brackets is true (i.e. its value is not zero), the instruction behind this condition (do_if_true) will be executed;
- If the expression true_or_false is false (i.e. its value is zero), the instruction behind this condition is **omitted** and the next instruction executed will be the one **after** the conditional instruction.

```
if (number > 0)
   cout << "This number is positive: " << number << endl;
```

Let's move on to another variant of the conditional statement, which also allows an action to be performed when the condition is not met.

This form is a little more complex and flexible.

II.3.8.2) The if…else statement

The C++ language allows us to express alternative plans. We do this with a second, slightly more complex form of the conditional statement - here it is:

```
if(true_or_false)
   Do_if_true;
else
   Do_if_false;
```

So we have a new word: **else**, which is a keyword (reserved word). An instruction beginning with else tells us what to do if the condition specified for **if** is **not met**.

Thus, if-else execution proceeds as follows:

- If the condition is **true** (its value is not zero), do_if_true is executed and the conditional statement ends;
- If the condition is **false** (its value is zero), do_if_false is executed and the conditional instruction ends;

```
    if (number >= 0)
       cout << " This number is positive: " << number << endl;
    else
       cout << " This number is negative " << number << endl;
```

As in other simplified forms of this statement, if and else can contain just one statement.

If you're going to write more than one statement, you'll need to use a block. We use braces {and} as in the example below:

```
    if (number >= 0) {
       cout << " You've entered the number : " << number << endl;
       cout << " This number is positive: "<<endl;
    }
    else {
       cout << " You've entered the number : " << number << endl;
       cout << " This number is negative :"<<endl;
    }
```

Note

In the style of **Bjarne Stroustrup**:

- A conditionally executed block is indented - it improves program readability and shows its conditional nature;
- The opening brace is on the same line as if;
- The closing brace is on a separate line.

II.3.8.3) The if...else if...else statement

The if ... else statement is used to execute a block of code between two alternatives. However, if we need to choose between more than two alternatives, we use the if ... else if ... else statement.

```
    if (number > 0) {
       cout << " You've entered a positive integer : " << number
<< endl;
    }
    else if (number < 0) {
       cout << " You've entered a negative integer " << number
<< endl;
    }
```

```
else {
   cout << "You've entered 0" << endl;
}
cout << "This ligne is always displayed.";
```

This way of assembling if statements is called cascading. Notice how indentation improves code readability.

In this program, the user enters a number. We then use if...else if...else to check whether the number is positive, negative or zero.

If the number is greater than 0, the code inside the if block is executed. If the number is less than 0, the code inside the else if block is executed. Otherwise, the code inside the else block is executed.

II.3.8.4) Nested If

Now it's time to discuss a special case of the conditional statement. First, think about when the statement **after if is another if.**

```
if (number != 0) {
   if ((number % 2) == 0) {
      cout << "This is an even number." << endl;
      }
   else {
      cout << "This is an odd number." << endl;
      }
}
else {
   cout << "This is zero." << endl;
}
cout << "This line is always displayed." << endl;
```

Let's look at two important points:

- Such use of the if statement is known as **nesting**; remember that each else refers to the nearest if, which does not correspond to any other else.
- Notice how **indentation** improves readability and emphasizes the nesting of internal conditional statements.

II.3.9) Computers and their logic

The conditions we've used so far are very simple, quite primitive in fact.

So it's clear that C++ needs logical operators to reinforce the expressive power of the language

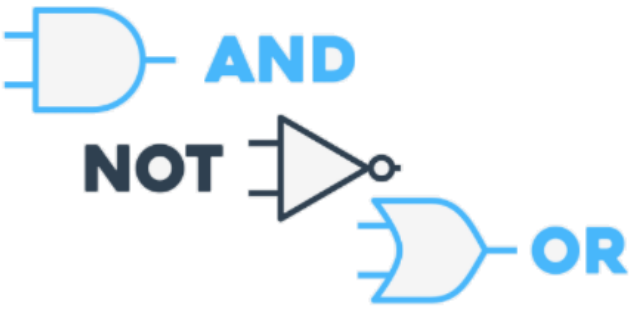

Figure 7 - Logical operators

II.3.9.1) The && - and operator

The logical conjunction operator in the C++ language is represented by &&.

It is a binary operator with a lower priority than comparison operators. It enables us to code complex conditions such as:

```
counter > 0 && value == 100
```

The result provided by the && operator can be determined on the basis of the truth table. If we consider the conjunction:

```
Left && Right
```

The set of possible argument values and corresponding conjunction values is as follows:

Table 10 - and Table

Left	Right	Lef && Right
False	False	False
False	True	False
True	False	False
True	True	True

Note

The && operator derives directly from C++'s ancestor, the C language. C++ has introduced a synonym, the and keyword. Both work in exactly the same way.

II.3.9.2) The || - or operator

The disjunction operator is represented by ||. It is a binary operator with a lower priority than && (just like "+" compared to "*"). Its truth table is as follows:

Table 11 - Or Table

Left	Right	Left || Right
False	False	False
False	True	True
True	False	True
True	True	True

Note

The || also has a synonym - the keyword **or**.

II.3.9.3) The ! - not operator

In addition, there's another operator that can be used to construct conditions. It's a unary operator performing a logical negation. Its operation is simple: it transforms true into false and false into true. This operator is written as a single character ! (exclamation mark) and has a very high priority: the same as the increment and decrement operators.

Its truth table is really quite simple:

Table 12 - Not Table

argument	!argument
False	True
True	False

The following conditions are respectively equivalent

```
variable > 0    !(variable <= 0)
variable != 0   !(variable == 0)
```

Note

The synonym for ! is the keyword **not.**

II.3.10) Loops

By now, you should be able to write a program that finds the largest of four, five, six or even ten numbers. But what if we ask you to write a program that finds the largest of a hundred numbers? Can you imagine the code?

The trick is based on the assumption that any part of the code can be executed more than once - in fact, as many times as necessary.

Executing a certain part of the code several times is called a loop. You probably already know what a loop is.

II.3.10.1) The "while" loop

This is one of the loops available in the C++ language. In general, the loop appears as follows:

```
while(conditional_expression)
   Instruction;
```

There's only one **syntactic** difference with the if statement: we've replaced the word if with the word while.

The **semantic** difference is more important: when the condition is met, **if** executes its instructions only **once**; while **repeats** execution as long as the condition is **true**.

If you want while to execute **more than one** statement, you must (as with the if statement) use a block.

An instruction or instructions executed inside the loop are called the loop body;

If the condition is false (equal to zero) from **the first test**, the body is not executed **at all**;

The body should be able to **change the value** of the condition, because if the condition is true at the start, the body could run continuously until **infinity**.

```
while(condition) {
   instruction_1;
   instruction_2;
   .
   .
   .
   instruction_n;
}
```

Here's an example of a loop that fails to complete its execution. This loop will print **I'm stuck in a loop** on the screen.

```
while(true) {
   cout << "I'm stuck in a loop" << endl;
}
```

See how the code above implements the greatest number algorithm.

```
#include <iostream>
using namespace std;
int main()
{
 /* temporarily store numbers */
 int number;
 /* Reading the first value */
 cin >> number;
```

```
     /* The actual maximum will be stored in this variable
(max)*/
     int max = number;
     /* If the number is different than -1 we continue */
     while (number != -1) {
          /* Is number greater than max ? */
          if (number > max)
               /* Yes – update max */
               max = number;
          /* Reading the next number */
          cin >> number;
     }
     /* Display the maximum */
     cout << "The greatest number is " << max << endl;
    }
```

The following program counts odd and even numbers from the keyboard.

```
    #include <iostream>
    using namespace std;
    int main()
    {
     /* variables for counting numbers */
     int odd = 0, even = 0;

     /* variable to store the new number */
     int number;

     /* Read the first number */
     cin >> number;

     /* 0 ends the loop */
     while (number != 0) {
          /* checks if the number is odd */
          if (number % 2 == 1)
               /* increments the odd counter */
               odd++;
          else
               /* increments the even counter */
               even++;
          /* Reading next number */
          cin >> number;
     }
     /* Display the result */
     cout << "even numbers: " << even << endl;
     cout << "odd numbers: " << odd << endl;
```

```
}
```

Some extracts can be simplified without changing the program's behavior.

When evaluating a condition, C++ interprets the truth. These two forms are equivalent.

```
while(number != 0) {...}
while(number) {...}
```

The condition that checks whether a number is odd can be coded as follows:

```
if(number % 2 == 1)...
if(number % 2)...
```

Example

```
int main()
{
   int counter = 5;
   while(counter != 0) {
      cout << " compact form of the condition " << endl;
               counter--;
   }
}
```

There are two things we can write more compactly. First, the while loop condition.

```
int main()
{
   int counter = 5;
   while(counter) {
      cout << " compact form of the condition " << endl;
                counter--;
   }
}
```

Another change requires some knowledge of how **post-decrementing** works. We're going to use it to compact our program again.

```
int main()
{
   int counter = 5;
   while(counter--)
      cout << " compact form of the condition " << endl;
}
```

II.3.10.2) The "do" loop, or execute at least once

We already know that the **while** loop has **two** important characteristics:

- It checks the condition before entering the body,
- The body will not be executed if the condition is false.

There's another loop in the C++ language that acts as a mirror image of the while loop.

We say this because in this loop :

- The condition is checked **at the end** of the body execution,
- The loop body is executed at least once, even if the condition is not met.

This loop is called the do loop. Its simplified syntax is as follows:

```
do
   instruction;
while(condition);
```

If you wish to execute a body containing several instructions, you must use a block.

```
do {
   instruction_1;
   instruction_2;
   :
   :
   instruction_n;
} while(condition);
```

Let's get back to the program we're looking for. Firstly, we'll be using the "**do**" loop instead of the "**while**" loop for pedagogical purposes. Secondly, we remove the vulnerability associated with over-reliance on the user's good will. Our new program will not be misled by entering the value -1 as the first number. Take a look at the editor. Here's our code.

We've used the counter variable to count the numbers entered so that we can tell the user that we can't search for the largest number if no number is given.

Since we need to read at least one number, it makes sense to use the do loop. We use this approach in the program.

```
#include <iostream>
using namespace std;
int main()
{
   int number;
   int max = -100000;
   int counter = 0;
   do {
      cin >> number;
      if (number != -1)
          counter++;
      if (number > max)
          max = number;
   } while (number != -1);
   if (counter)
```

```
        cout << " The greatest number is " << max << endl;
      else
        cout << " You haven't entered any numbers!"<< endl;
  }
```

II.3.10.3) The "for" loop

The last type of loop available in C++ comes from the fact that it is sometimes more important to count the "**iterations**" of the loop than to check the conditions.

Imagine that the body of a loop has to be executed exactly one hundred times. If you want to use the while loop for this purpose, it might look like this:

```
int i = 0;
while(i < 100) {
  /* The body of the loop */
  i++;
}
```

Three independent elements can be distinguished:

- Counter initialization (i=0)
- Status check or verification (i<100)
- Counter modification (i++)

It is possible to create a generalized scheme for this type of loop:

```
Initialisation;
while(verification) {
  /* The body of the loop */
  Modification;
}
```

This way of coding the loop is very common, so there's a special, brief way of writing it in the C++ language.

Below, we've put together the three decisive parts. The loop is clear and easy to understand. Its name is **for.**

```
for(initialization; verification; modification) {
  /* The body of the loop */
}
```

The variable used to **count loop repetitions** is often referred to as the **control variable.**

Note that if a control variable is declared in the initialization clause (as in the example), the variable is only available inside the loop body.

If this is not desirable and you wish to use the same variable somewhere outside the loop body, you must declare it outside the loop, like this:

```
int i;
```

```
/*
Further statements...
*/

for(i = 0; i < 100; i++) {
   // The body of the declaration
}
```

The **for** loop has an interesting **quirk**. If we omit one of its three components, we assume there's a 1 in its place.

One of the consequences of this is that a loop written in this way is an infinite loop (do you know why?).

Well, the conditional expression isn't there, so it's automatically assumed to be **true**. And because the condition never becomes false, the loop becomes infinite.

```
for(;;) {
  /* The body of the loop */
}
```

II.3.10.4) break and continue

So far, we've treated the loop body as a sequence of **indivisible**, **inseparable** instructions that are executed completely each time the loop is repeated. However, as a developer, you may be faced with the following choices:

- We no longer need to continue the loop as a whole; we should stop executing the body of the loop and continue with the rest of the program;
- It looks as if we need to restart the condition test without completing execution of the body.

The C++ language provides us with two special instructions to implement these two tasks. An experienced programmer can code any algorithm without these instructions.

These two instructions are:

- **break** - exits the loop immediately and unconditionally terminates the loop instructions; the program begins executing the instruction closest to the end of the loop body;
- **continue** - behaves as if the program had suddenly reached the end of the loop body; the end of the loop body is reached and the condition expression is immediately tested.

These two words are **keywords.**

Let's look at two simple examples. We'll return to our program which recognizes the largest number entered. We'll run it twice, using both instructions.

Note that the only way out of the body is to execute **break**, as the loop itself is **infinite** (for (;;)).

```
#include <iostream>
using namespace std;
int main()
{
 int number;
 int max = -100000;
 int counter = 0;
 for (;;) {
      cin >> number;
      if (number == -1)
           break;
      counter++;
      if (number > max)
           max = number;
 }
 if (counter)
      cout <<" The greatest number is " << max << endl;
 else
      cout << " You haven't entered any number!"  << endl;
}
```

And now the variant **continues.**

```
#include <iostream>
using namespace std;

int main()
{
 int number;
 int max = -100000;
 int counter = 0;

 do {
      cin >> number;
      if (number == -1)
           continue;
      counter++;
      if (number > max)
           max = number;
 } while (number != -1);
 if(counter)
      cout << " The greatest number is " << max << endl;
 else
      cout << " You haven't entered any number!"   << endl;
}
```

Chapter III. Structure Objects

III.1) One-dimensional arrays

A simple type of variable (int, float, char... etc.) can only contain a single value. If you want to store a set of values in memory, you need to use a data structure called an array, which allows you to store a list of elements. In this chapter, we'll look at static arrays, whose size is **fixed** once and for all.

III.1.1) Declaring a static array

As always in C++, a variable consists of a name and a type. As arrays are variables, this rule remains valid. All that needs to be added is an additional property: **the size** of the array. In other words, the number of cells our memory cell can hold.

Declaring an array is very similar to declaring a variable:

Syntax: `type identifier[size];`

We indicate the type, then the chosen name and finally, in square brackets, the size of the array.

Let's take a look at an example:

```
#include <iostream>
using namespace std;
int main()
{
   int tab[5];       //Declares an array of 5 int
   float anglesTriangle[3];   //Declares an array of 3 float
   return 0;
}
```

An array is made up of a set of cells. Each cell will contain an element whose type will be **type**. The name of the array will be **identifier**. The total number of cells in the array will be **size**. This variable must be a constant. This is referred to as a **static array**.

Constants

To define a constant.

Syntax

```
const type identifier=constant_value;
```

```
const int N=5 ;
```

We recommend you always use constants to express the size of your arrays, rather than directly indicating the size between square brackets. It's a good habit to get into.

We'll define a constant N which will be equal to the number of cells in the array, and we'll write our entire program as a function of N.

If we want to change the size of the array, we'll simply change the value of N at a single point in the program.

An example of an array

```
int tab[5];
```

tab is an array of 5 cells. Each cell contains an integer (type int). The first cell is tab[0]. The second cell is tab[1] and the fifth is tab[4].

Note: tab[5] does not exist! This is because the first cell has index 0. If you try to access a cell with an invalid index, you'll get an **error.**

The usual operations can be performed on each cell: addition, multiplication, assignment, etc.

III.1.2) Declaring and initializing a static array

An array can be declared and initialized as follows: int tab[]={8,7,6,4,8};

The array will always be a fixed-size array, in this case an array with 5 cells. The array tab is not a variable-size array.

This syntax avoids writing: int tab[5]; tab[0]=8; tab[1]=7; tab[2]=6; tab[3]=4; tab[4]=8;

III.1.3) Index of an element in an array

The contents of an array can be an int, a float, etc. This type is defined when the array is declared and cannot be changed. The index of an element in an array must be of **integer** type.

III.1.4) Accessing array elements

Each cell in an array can be used in the same way as any other variable - there's no difference. You just need to access them in a slightly special way. You must specify the name of the array and the number of the cell (the index). To access a cell, use the syntax **arrayname[cellnumber].** There is just one little subtlety: the first cell has the number 0, not 1. In other words, everything is shifted by 1.

To access the third cell of tab and store a value there, you'll need to write:

tab [2] = 5; the third cell therefore has the number 2.

III.1.5) Navigating an array

The great thing about arrays is that they can be browsed using a loop. This means you can perform an action on each of the cells in an array, one after the other: for example, display the contents of the cells.

We know the number of cells in the array a priori, so we can use a for loop. We can use the i variable in the loop to access the ith element of the array.

```
const int number=5;           //array size
int results[number];   //declaration of the array

results[0] = 1457;  // 1st case
results[1] = 2384;  // 2nd case
results[2] = 3096;   // 3rd case
results[3] = 4781;   // 4th case
results[4] = 5123;   // 5th case

for(int i=0; i<number; i++){
    cout << results[i] << endl;
}
```

The variable i successively takes the values 0, 1, 2, 3 and 4, which means that the values of results[0], then results[1], etc. are sent to cout.

You must be very careful not to exceed the size of the array in the loop, otherwise your program will **crash.** The last cell in this example has the index number minus one. Allowed values for i are all integers between **0** and (**number -1**) included.

A small example

We're going to use C++ to calculate the mean of your scores for the year. Let's start by putting all the scores in an array and using a for loop to calculate the average. Let's take a step-by-step look.

The first step is to declare an array to store the grades. As these are comma-delimited numbers, we need floats.

```
const int score_number=6;
float scores[score_number];
```

The second step is to fill this array with your scores.

```
int const score_number(6);
float scores[score_number];
scores[0] = 14.5;
scores[1] = 17.5;
scores[2] = 8.;
scores[3] = 10;
scores[4] = 11.5;
scores[5] = 12;
```

To calculate the average, we need to add up all the scores and then divide the result by the number of scores. We already know the number of scores, since we have the constant score_number. All that remains is to declare a variable to contain the average.

The sum is calculated in a for loop that runs through all the cells in the array.

```
    float mean=0;
    for(int i=0; i<score_number; i++)
    {
       mean += scores[i];   //All scores are added together
    }
```

At this point, the average variable contains the sum of the scores (73.5).

All that remains is to divide by the number of scores

```
    mean /= score_number;
```

With a short line to display the value, you get the desired result: a program that calculates the average of your grades.

```
    #include <iostream>
    using namespace std;
    int main()
    {
       const int score_number=6;
       float scores[score_number];
       scores[0] = 14.5;
       scores[1] = 17.5;
       scores[2] = 8.;
       scores[3] = 10;
       scores[4] = 11.5;
       scores[5] = 12;
       float mean=0;
       for(int i=0; i<score_number; i++)
       {
          mean += scores[i];   // We add up all the scores
       }
       //At this point mean contains the sum of the scores(73.5)
       //All that remains is to divide by the number of notes
       mean /= score_number;
       cout << "Your mean is : " << mean << endl;
       return 0;
    }
```

Let's see what it looks like when you run it:

Your mean is: 12.25

III.1.6) Array processing

We're now going to look at a number of standard algorithms that you need to know how to perform on arrays: searching for the smallest element, inverting an array...

III.1.6.1) Searching an array

We want to write a program to find the smallest element in an array containing 4 cells.

Algorithm used we'll store our smallest element in a variable sme. We start by initializing sme to the first element of the array (index zero). We then go through all the other elements in the array, comparing the current element with sme and updating the value of sme if necessary. Now we need to formalize this algorithm.

```
#include <iostream>
using namespace std;
int main()
{
   int t[4], i, sme;
   for(i=0; i<4; i++){
      cout << "Type value number " << i << ": ";
      cin >> t[i];
   }
   sme = t[0];
   for(i=1; i<4; i++) {
      if(sme>t[i]) {
         sme=t[i];
   }
   }
   cout << "The smallest value is "<< sme <<endl;
   return 0;
}
```

Explanations

- To calculate the smallest element of an array, initialize sme to t[0].
- We then go through the array from cell 1 to the last cell of index 3, comparing t[i] with sme. If t[i] is smaller than the smallest current sme, we copy t[i] into sme (t[i] becomes the new smallest current).

Otherwise, the value of sme remains unchanged.

- Finally, we display the value of sme outside the loop.

Execution

Type value number 0: 9

Type value number 1: 7

Type value number 2: 4

Type value number 3: 15

The smallest value is 4

III.1.6.2) Reversing the order of array elements

```
#include <iostream>
using namespace std;
int main()
```

```
{
   int t[6], i, a;
   for(i=0; i<6; i++){
cout << "Type value number " << i << ": ";
cin >> t[i];
}
   for(i=0; i<3; i++){
      a = t[i];
      t[i] = t[5-i];
      t[5-i] = a;
   }
   for(i=0; i<6; i++)
      cout << "Value number " << i << " is " << t[i] << endl;
   return 0;
}
```

Explanations

To invert the contents of the 6-slot array t there are 3 steps:

1. step 0: exchange t[0] and t[5].
2. step 1: swap t[1] and t[4]
3. step 2: exchange t[2] and t[3].

At step i, we exchange t[i] and t[5-i].

Each step is numbered from i=0 to i=2.

Execution

Type value number 0: 9

Type value number 1: 7

Type value number 2: 11

Type value number 3: 15

Type value number 4: 16

Type value number 5: 4

Value number 0 is 4

Value number 1 is 16

Value number 2 is 15

Value number 3 is 11

Value number 4 is 7

Value number 5 is 9

III.2) Two-dimensional arrays

In C/C++, we can define multi-dimensional arrays, arrays of arrays.

The syntax for declaring 2-dimensional arrays:

```
Type Array_name[size1][size2] ;
```

Type: type of data to be stored in the array.

array_name: array name

size1, size2: Number of rows and number of columns.

For example:

```
int x[3][4];
```

Here, x is a two-dimensional array. It can contain a maximum of 12 elements.

We can think of this as an array with 3 rows and each row has 4 columns, as shown below.

	Column1	Column2	Column3	Column4
Row1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Figure 8 - Elements in a two-dimensional array in C++ programming

III.2.1) Declaration and initialization of a two-dimensional array

There are two ways of initializing a two-dimensional array.

First method:

```
int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}
```

The table above has 3 rows and 4 columns. Elements in braces from left to right are stored in the array also from left to right. The elements will be filled in the array in order, the first 4 elements from the left in the first row, the next 4 elements in the second row, and so on.

Best method:

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}} ;
```

This type of initialization uses nested braces. Each set of inner braces represents a line. In the example above, there are a total of three lines, so there are three sets of inner braces.

III.2.2) Access the elements of a two-dimensional table

The elements of two-dimensional arrays are accessed using row indexes and column

indexes.

Example:

```
int x[2][1];
```

The above example represents the element present in the third row and second column.

Note: If the size of an array is N. Its index will be from 0 to N-1. Therefore, the index value in row 2 represents the 3rd row (2 + 1 = 3).

To display all the elements of a two-dimensional array, we can use **nested** for loops. We'll need two for loops. One to run through the **rows** and another to run through the **columns.**

```
#include<iostream>
using namespace std;

int main()
{
    // a 3 rows and 2 columns array.
    int x[3][2] = {{0,1}, {2,3}, {4,5}};
     // Display each element's value
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cout << "Element x[" << i<< "][" << j << "]: ";
            cout << x[i][j]<<endl;
        }
    }
     return 0;
}
```

Output:

```
Element x[0][0]: 0
Element x[0][1]: 1
Element x[1][0]: 2
Element x[1][1]: 3
Element x[2][0]: 4
Element x[2][1]: 5
```

III.3) Sorting

Sorting in C++ refers to the process of arranging elements in a specific order, typically in ascending or descending order. It is a fundamental operation in programming, used to organize data for efficient searching, processing, or presentation.

Since sorting can often help to reduce the algorithmic complexity of a problem, it finds important uses in computer science.

III.3.1) Selection sort (in ascending order)

The algorithm divides the array into two parts:

1. **Sorted part**: Initially empty, elements are added one by one.
2. **Unsorted part**: Initially contains all elements, which shrink as elements are moved to the sorted part.

At each step:

1. Find the smallest (or largest) element in the unsorted portion.
2. Swap it with the first element in the unsorted portion.
3. Move the boundary of the sorted part one element forward.

The following example explains the above steps:

First pass

```
arr[] = 58 25 12 22 11

// Find the smallest element (11) of the array arr[0...4]

// and swap it with the first element (58) of the array arr[0...4]

11 25 12 22 58
```

Second pass

```
// Find the smallest element (12) of the array arr[1...4]
// and swap it with the second element (25) of the array arr[1...4]

11 12 25 22 58
```

Third pass

```
// Find the smallest element (22) of the array arr[2...4]
// and swap it with the third element (25) of the array arr[2...4]

11 12 22 25 58
```

Fourth Pass

```
// Find the minimum of the array arr[3...4]
// swap it with the fourth element (58) of the array arr[3...4]

11 12 22 25 58
```

Fifth Pass

```
// Sorted array
11 12 22 25 58
```

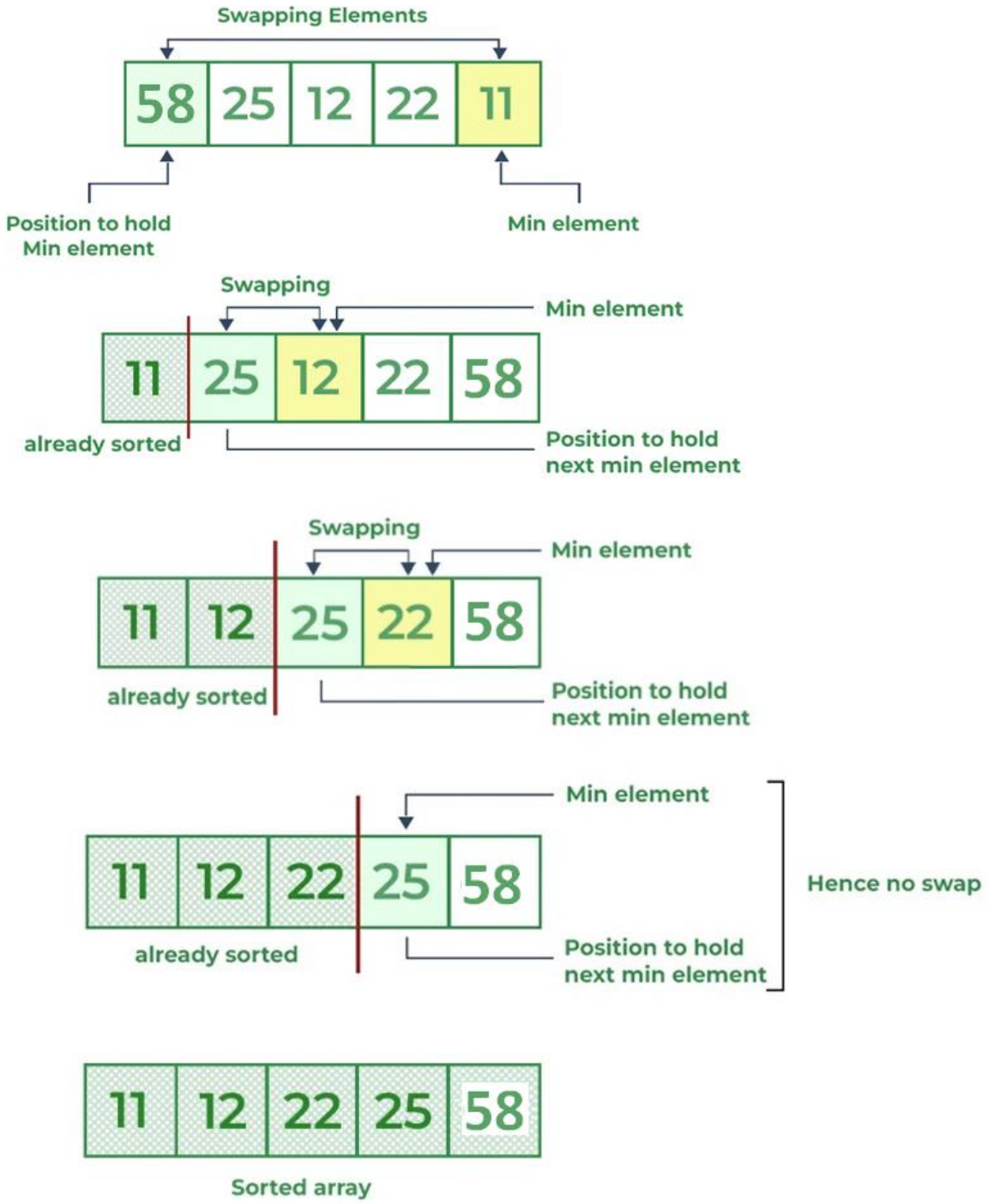

Figure 9 - Selection sort Example

```
#include <iostream>
using namespace std;
int main()
{
     int i, j, min_idx;
 int arr[] = {58, 25, 12, 22, 11};
 int n = sizeof(arr)/sizeof(arr[0]);
```

```
    //Display
    for (i = 0; i < n; i++){
         cout << arr[i] << " ";
    }
    cout<<endl;
       // Browse the array
    for (i = 0; i < n-1; i++){
         // Find the minimum of the unsorted part
         min_idx = i;
         for (j = i+1; j < n; j++)
              if (arr[j] < arr[min_idx])
                   min_idx = j;
    //Exchange the minimum with the first element
         int temp = arr[min_idx];
         arr[min_idx] = arr[i];
         arr[i] = temp;
   }
         cout<<"Sorted array: \n";
   //Display
    for (i = 0; i < n; i++){
         cout << arr[i] << " ";
    }
    return 0;
   }
```

III.3.2) Bubble sort

Bubble sorting is the simplest sorting algorithm, it repeatedly steps through the list compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted

Example
First pass

(**5 1** 4 2 8) –> (**1 5** 4 2 8), here the algorithm compares the first two elements, and exchange them since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), exchange since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), exchange since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Since the elements are already in the right order (8 > 5), the algorithm doesn't exchange values.

Second Pass

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), exchange since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

At this point, the array is already sorted, but the algorithm is unaware of it. It requires a full pass without any swaps to confirm that the array is sorted.

Third Pass

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

```
#include <iostream>
using namespace std;
int main()
{
     int i, j,temp;
 int arr[] = {64, 34, 25, 12, 22, 11, 90};
 int n = sizeof(arr)/sizeof(arr[0]);
 //Display
 for (i = 0; i < n; i++){
      cout << arr[i] << " ";
 }
 cout<<endl;

 for (i = 0; i < n-1; i++)
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]){
                //exchange
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1]= temp;
                //
            }
    cout<<"Sorted array: \n";
 //Display
 for (i = 0; i < n; i++){
      cout << arr[i] << " ";
 }
 return 0;
}
```

III.3.3) Odd-Even sort

Known as Brick Sort, this algorithm is a modified version of bubble sort, designed to

operate in two distinct stages: the **odd phase** and the **even phase**.

At the Odd Phase, the algorithm compares and swaps adjacent elements at odd indices with the elements at the next even indices. Example: Compare element at index 1 with index 2, then index 3 with index 4, and so on.

At the Even Phase, the algorithm compares swaps adjacent elements at even indices with the next odd indices. Example: Compare element at index 0 with index 1, then index 2 with index 3, and so on.

The algorithm alternates between the odd and even phases until the array is completely sorted. Then the process stops when no swaps are needed in both phases during a full iteration.

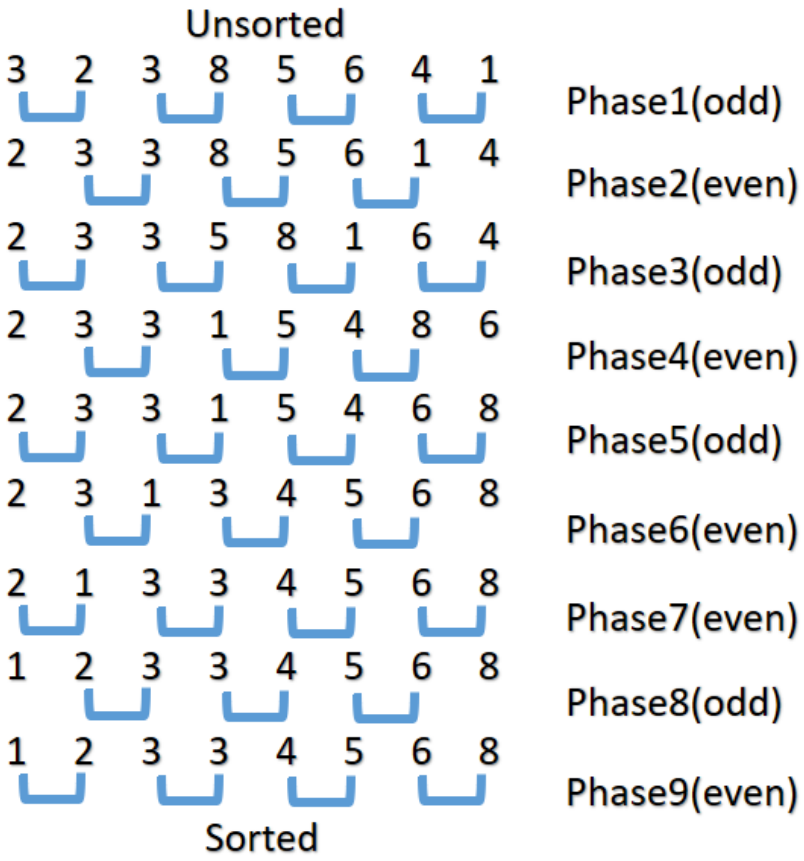

Figure 10 - Odd-Even sort Example

III.3.4) Insertion sort

Insertion sort is a simple sorting algorithm. The array is virtually divided into a sorted part and an unsorted part. Values from the unsorted part are selected and placed at the correct position in the sorted part.

Algorithm

1. Begin with the second element, as the first is considered already "sorted."
2. Compare the current element to the elements in the sorted section of the array.
3. Move all larger elements in the sorted section one position to the right to create space for the current element.
4. Place the current element in its appropriate position.
5. Continue this process for each element in the array.

Example

Array: [5, 3, 8, 6, 2]

1. Start with index 1 (value 3):
   - Compare 3 with 5. Shift 5 to the right → [5, 5, 8, 6, 2].
   - Insert 3 at index 0 → [3, 5, 8, 6, 2].
2. Move to index 2 (value 8):
   - Compare 8 with 5. No shift needed.
   - The array remains [3, 5, 8, 6, 2].
3. Move to index 3 (value 6):
   - Compare 6 with 8. Shift 8 to the right → [3, 5, 8, 8, 2].
   - Compare 6 with 5. No more shifts needed.
   - Insert 6 at index 2 → [3, 5, 6, 8, 2].
4. Move to index 4 (value 2):
   - Compare 2 with 8. Shift 8 → [3, 5, 6, 8, 8].
   - Compare 2 with 6. Shift 6 → [3, 5, 6, 6, 8].
   - Compare 2 with 5. Shift 5 → [3, 5, 5, 6, 8].
   - Compare 2 with 3. Shift 3 → [3, 3, 5, 6, 8].
   - Insert 2 at index 0 → [2, 3, 5, 6, 8].

Sorted array: [2, 3, 5, 6, 8].

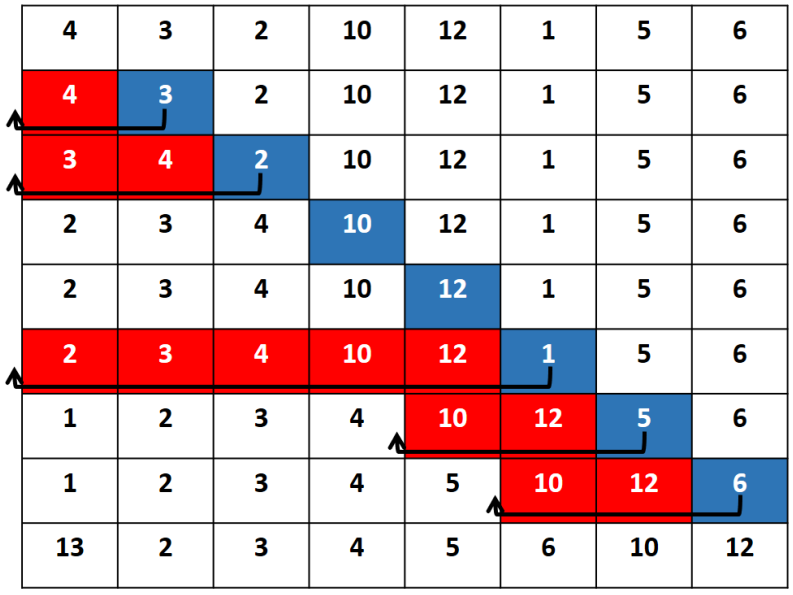

Figure 11 - Insertion sort example

```
    #include <iostream>
    using namespace std;
    int main()
    {
     int i, key, j;
     int arr[] = { 12, 11, 13, 5, 6 };
     int n = sizeof(arr) / sizeof(arr[0]);
    //Display initial array
     for (i = 0; i < n; i++)
     cout << arr[i] << " ";
     cout << endl;
    //Insertion sort
        for (i = 1; i < n; i++){
          key = arr[i];
          j = i - 1;
          /* Move elements from arr[0..i-1], that are > than
key, by one position*/
          while (j >= 0 && arr[j] > key)
          {
               arr[j + 1] = arr[j];
               j = j - 1;
          }
          arr[j + 1] = key;
     }
        cout<<"Sorted array: \n";
    //Display the final result
     for (i = 0; i < n; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

III.4) Strings

char variables are useful when we want to process single characters, but when we need to process data containing text. It's tedious and slow. It's much easier to treat all characters as a whole, stored, allocated and processed at the same time.

The C++ string type is a way of storing, representing and working with text and characters, with the possibility of accessing one character at a time. The string is represented as an object of the C++ String class (std::string).

A string variable contains a set of characters enclosed in quotation marks.

Let's start with a simple declaration.

```
// Include string library
#include <string>
// create a string variable
string name ;
```

We have declared a variable named name. It is intended for storing names.

Note: The **string** type is not a built-in type (like **int**, which can be used at any time without special preparation). If we want to use strings in our code, we must place the **#include** directive at the top of our program and request that a header file named **string** be included at compile time.

Failure to do so will result in a compilation error.

A constant can be of type string.

```
// Include string library
#include <string>
// Create a string constant
const string name="Algorithm" ;
```

III.4.1) Initializing a character string

A string can be initialized in the same way as other regular types, i.e. by using an assignment operator followed by a string literal (a set of characters enclosed in quotation marks). But remember: apostrophes are reserved for character literals. Strings always use quotation marks.

```
string name = "Ali" ;
```

In fact, the new variable name will be assigned a string composed of the characters "Ali".

There's another way to initialize string variables:

```
string name("Ali") ;
```

We don't use the assignment operator here. We use a syntax that clearly resembles the invocation of a function, with a pair of parentheses and an argument inside.

Both forms of initialization are the same, and their results are exactly the same.

III.4.2) Operations on strings

III.4.2.1) The length of a string

Every string has a length. Even an empty string (containing no characters) has a length of zero. Obviously, at some point we may want to know the length of a particular string.

This information is provided by two twin functions. Their names are different, but their behaviors are identical. Both functions return a value equal to the number of all characters currently stored in a string.

Their prototypes look like this:

```
string name = "Ali" ;
int sos = name.size() ;
int sos = name.length() ;
```

Of course, these two int variables are equal.

Example

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ;
cout << "The length of the string is : " << txt.length() ;
```

Example

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ;
cout << "The length of the string is : " << txt.size() ;
```

What is the result of this program?

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string str = "12345";
  cout << str.size() << endl;
}
```

III.4.2.2) Character access

You can access the characters of a string by referring to its index number in square brackets [].

This example prints the first character of myString :

Example

```
string myString = "Hello";
cout << myString[0] ;
// Output H
```

Note: String indices begin with 0: [0] is the first character. [1] is the second character, etc.

This example prints the second character of myString :

Example

```
string myString = "Hello";
cout << myString[1];
// Output e
```

III.4.2.3) Modifying characters in a string

To modify the value of a specific character in a string, refer to the index number and use single quotes:

Example

```
string myString = "Hello";
myString[0] = 'J' ;
cout << myString ;
// Displays Jello instead of Hello
```

III.4.2.4) Concatenation of character strings

A. String operator +

Like any other data type, the string type has its own operators. One of the most important and frequently used is the + operator. It doesn't add strings in the same way as traditional arithmetic addition. Instead, it concatenates strings.

The + (concatenation) operator has one important limitation. It cannot concatenate literals. It can concatenate a variable with a literal, a literal with a variable, and obviously a variable with another variable, but concatenating literals is something the operator will never do.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string the_question = "To be ";
  the_question += "or not to be";
  cout << the_question << endl;
}
```

B. Adding numbers and strings

C++ uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number.

Example

```
int x = 10 ;
int y = 20 ;
int z = x + y ; // z is equal to 30 (an integer)
```

If you add two strings, the result will be a concatenation of strings:

Example

```
string x = "10" ;
string y = "20" ;
string z = x + y ; // z is equal to 1020 (a string)
```

If you try to add a number to a string, an error occurs:

Example

```
string x = "10" ;
int y = 20 ;
string z = x + y ;
```

C. Append

A string in C++ contains functions for performing certain operations on strings. For example, you can also concatenate strings with the append() function:

Example

```
string firstName = "Bjarne ";
string lastName = "Stroustrup";
string fullName = firstName.append(lastName);
cout << fullName;
```

The append() function is much faster. It is able to append not only a string, but also a substring of the string, like this:

```
string str1 = "content"; str1.append("tail",1,3);
// str1 now contains "contentail"
```

And finally, append can add a character (possibly repeated n times), like this:

```
string str1 = "content"; str1.append(3, 'x') ;
// str1 now contains "contentxxx".
```

III.4.2.5) Reading strings

Outputting strings is easy and generally trouble-free. You can place a string variable among other expressions to be displayed on screen using the cout flow, and its contents will be sent character by character.

However, cin considers a space (white space, tab, etc.) as an end character, which means it can only display a single word (even if you type several words). This means you may run into difficulties if you want to enter and store a string containing blank characters.

Type a few words on the keyboard, separated by spaces. You could write something like this:

To be or not to be

and press Enter.

Don't be surprised if you only see the first word printed on the screen (e.g. To). The cin stream is convinced that you've entered the space to mark the end of the string.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string line;
    cin >> line;
  cout << line << endl;
}
```

Example

```
string fullName ;
cout << "Write your full name : " ;
cin >> fullName ;
cout << "Your full name is : " << fullName ;
// Write your full name : Bjarne Stroustrup
// Your name is : Bjarne
```

If you want to enter an entire line of text and treat blank characters as any other character, you need to use the getline() function. This function retrieves/reads all characters entered as they are, and does not favor any character except the character representing the enter key, which marks the end of the line. As a result, all characters entered before the key is pressed will be entered as a single string.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string line;
  getline(cin, line);
  cout << line << endl;
}
```

Example

```
string fullName ;
cout << "Enter your full name : " ;
getline (cin, fullName) ;
cout << "Your name is : " << fullName ;
// Enter your full name : Bjarne Stroustrup
// Your name is : Bjarne Stroustrup
```

III.4.2.6) String comparison

Strings (just like any other data) can be compared. The simplest case is when you want to check whether two string variables contain identical strings.

Something similar happens when you enter your password when logging into an account. The password you have entered is compared with the password stored in the system, and if the two strings are equal, access is granted.

A. Using operators

If you want to check whether two string variables contain the same character string, you can use the == operator.

Example

```
#include <iostream>
#include <string>
using namespace std;
int main() {
 string secret = "itsasecret";
 string password;
 cout << "Enter password:" << endl;
 getline(cin, password);
 if (secret == password)
      cout << "Access granted" << endl;
 else
      cout << "Sorry";
}
```

Of course, you can also compare two strings in a more flexible way. All the operators for comparing data are at your disposal: > < >= <= !=. You can check whether one of the strings is greater or smaller than the other, but remember that these comparisons are made in lexicographical order, where "a" is greater than "A" and obviously "z" is greater than "a", but less obviously, "a" is greater than "1". See the ASCII table.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Figure 12 - ASCII table

You can use this type of string comparison, for example, to determine alphabetical order.

The example takes two lines of text and prints them in alphabetical order.

```
    #include <iostream>
    #include <string>
    using namespace std;
    int main()
    {
      string str1, str2;
      cout << " Enter 2 lines of text:" << endl;
      getline(cin, str1);
      getline(cin, str2);
      cout << "You've entered:'" << endl;
      if (str1 == str2)
        cout << "\"" << str1 << "\" == \"" << str2 << "\"" <<
endl;
      else if (str1 > str2)
```

```
        cout << "\"" << str1 << "\" > \"" << str2 << "\"" <<
endl;
      else
        cout << "\"" << str1 << "\" < \"" << str2 << "\"" <<
endl;
    }
```

B. Using a function

Character strings offer another method of comparison.

Let's start with a very simple example. We're going to rewrite the password verification program to show you one of the many functions that exist in every string.

The function is called "compare", and is designed to compare one string with another string. The function returns 0 (zero) if the strings are identical.

```
    #include <iostream>
    #include <string>
    using namespace std;
    int main()
    {
     string secret = "abracadabra";
     string password;
     cout << "Enter password:" << endl;
     getline(cin, password);
     if (secret.compare(password) == 0)
          cout << "Access granted" << endl;
     else
          cout << "Sorry";
    }
```

The "compare" function can also diagnose all possible relationships between two strings. Here's how it works:

```
    str1.compare(str2) == 0 when str1 == str2
    str1.compare(str2) > 0 when str1 > str2
    str1.compare(str2) < 0 when str1 < str2
```

Once again, let's rewrite one of our previous programs.

```
    #include <iostream>
    #include <string>
    using namespace std;
    int main()
    {
      string str1, str2;
      cout << "Enter 2 lines of text :" << endl;
      getline(cin, str1);
      getline(cin, str2);
```

```
      cout << "You've entered :'" << endl;
      if (str1.compare(str2) == 0)
        cout << "\"" << str1 << "\" == \"" << str2 << "\"" <<
endl;
      else if (str1.compare(str2) > 0)
        cout << "\"" << str1 << "\" > \"" << str2 << "\"" <<
endl;
      else
        cout << "\"" << str1 << "\" < \"" << str2 << "\"" <<
endl;
    }
```

III.4.2.7) Sub-strings

All the string operations you've seen so far refer to whole strings, i.e. both strings are taken in their entirety, from the first to the last character. Part of a string is called a substring.

If we want to create a new string made up of characters taken from the substring of another string (or even the same one), we can use a function called substr, and its simplified prototype looks like this:

```
new_string = old_string.substr(start_position, substring_size)
```

The substring of a character string is defined by two "coordinates":

A location where the substring begins (specified by the position_begin parameter) ;

Its length (specified by the sub-string_size parameter).

Note that the characters in a string are numbered, and that the first character is number 0. Therefore, if a string contains n characters, the last character is number n - 1.

Both parameters have default values. This allows us to use the function more flexibly. For example

s.substr(1,2) describes a substring of string s, starting at its second character and ending at its third character. s.substr(1) describes a substring starting at the second character of string s and containing all the remaining characters of s, including the last; the omitted string_size parameter covers all the remaining characters of string s by default.

s.substr() is just a copy of the entire string s (the start_position parameter defaults to 0).

Take a look at the following example and guess the result:

```
    #include <iostream>
    #include <string>
    using namespace std;
    int main()
    {
```

```
     string str1 = "ABCDEF";
     string  str2  =  str1.substr(1,  1)  +  str1.substr(4)  +
str1.substr();
          cout << str2 << endl;
    }
```

Yes, it's BEFABCDEF.

Note: getting a substring requires you to be precise. You mustn't define a substring that doesn't fit entirely within the original string (for example, that starts next to the end of the original string). If you do, your instruction will be rejected.

The substr function returns a string. Which has its own functions, such as substr() or size().

III.4.2.8) Searching within a string

Sometimes we need to find a substring within another string, bearing in mind that the search may fail.

We can search for a substring or a single character. To do this, we need to use one of the variants of the **find** function.

```
    int start = full_string.find(sub_string,start_pos) ;
    int start = full_string.find(character, start_pos) ;
```

In both variants, the start_pos parameter defaults to 0, so if you omit it, the string will be searched from the beginning.

The result returned by the functions points to the first location in the string where the searched string begins, or to the location of the searched character (depending on the variant). If the search fails, both functions return a special value called string::npos.

```
    #include <iostream>
    #include <string>
    using namespace std;
    int main()
    {
      string string1 = "my name is Albert, Albert einstein.";
      string string2 = "Albert";
      if (string1.find(string2) != string::npos)
        cout << "Yes, it is him!" << endl;
      else
        cout << "It is not him<< endl;
        int comma = string1.find(',');
      if (comma != string::npos)
        cout << "There is a comma in the string" << endl;
    }
```

III.4.2.9) Emptying a string

First, we can empty the string, completely deleting all the characters currently stored in it. This is equivalent to assigning an empty string to the string, but it could be a bit faster.

Emptying the string is done by a function called **clear().** It takes no parameters.

III.4.2.10) Adding a character

If you want to add a single character to a string, you can do so using the append function, but there's a more efficient way, using the push_back member function.

When you compile and run the code, you'll see the classic Latin alphabet.

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
  string the_string;
  for (char c = 'A'; c <= 'Z'; c++)
    the_string.push_back(c);
  cout << the_string << endl;
  return 0;
}
```

III.4.2.11) Inserting a (sub)string or character

Inserting a string into a string means distending its content from within. A new set of contents is simply "pushed" into the old one. For example: the following code snippet will print "to be or not to be" in the cout flow:

```
string quote = "to be " ;
quote.append(quote) ;
quote.insert(6, "or not ") ;
cout << quote << endl ;
```

The first parameter specifies where the insertion is to take place, while the second tells you what is to be inserted. This is just one of the various possibilities offered by the function.

There's also a function for inserting a single character, which can be duplicated several times. These two functions are used in the following example. Look at the code in the editor.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string quote = "Whyserious?", anyword = "monsoon";
  quote.insert(3, 2, ' ').insert(4, anyword, 3, 2);
```

```
    cout << quote << endl;
}
```

III.4.2.12) Replacing a (sub)string

The **replace()** function can replace part of a string with another string.

The function needs to know which part of the string it's going to replace, and you need to specify this by supplying two numbers:

The first describing the starting position;

The second indicating the number of characters to be replaced.

The first overloaded function needs a string (as a third parameter) to replace the old content (it can be longer or shorter, or of equal size to the original). The second function specifies the substring to be used as a substitution.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string to_do = "I'll think about that in one hour";
  string schedule = "today yesterday tomorrow";
  to_do.replace(22, 12, schedule, 16, 8);
  cout << to_do << endl;
}
```

III.4.2.13) Deleting a(sub)string

We can also delete part of a string, making it shorter than before. To do this, we use a function called **erase().** This function needs two numbers to do its job:

The place where the substring to be deleted begins (this value is zero by default) ;

The length of the substring (this value defaults to the length of the original string).

This means that an invocation like this

string word.erase() ;

Erases all characters from the string, leaving it empty.

The following example displays:

I think we're in Ramadan

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string place = "I think we’re in Shaban or Ramadan";
  place.erase(17, 10);
```

```
    cout << place << endl;
}
```

III.4.2.14) Exchanging the contents of two strings

When it comes to sorting, we need to be able to exchange the contents of two selected cells in an array. We need an auxiliary variable for this purpose, as the operation is akin to replacing the liquid contents of two glasses - you can't do it successfully without using a third glass. So, if we want to replace a string stored in the variables glass1 and glass2, we need a third variable (glass) to do it. Here's how it works:

glass = glass1;

glass1 = glass2;

glass2 = glass;

Unfortunately, this method is very inefficient when it comes to strings. It would be much simpler (and also much faster) not to transfer the entire contents of the two strings, but just to replace the **pointers** that identify them (it's like replacing the owners of the glasses rather than exchanging the contents of the glasses between them). This action is performed by a **swap()** function. This function is several times faster than an ordinary swap (performed physically).

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string glass1 = "milk";
  string glass2 = "coffee";
  cout << glass1 << ". " << glass2 << "." << endl;
  glass1.swap(glass2);
  cout << glass1 << ". " << glass2 << "." << endl;
}
```

III.5) C++ Structures

We often face situations where we need to store a group of data, whether of similar types or not. We have seen arrays in C++ which are used to store a set of data of similar types in adjacent memory locations.

Unlike arrays, structures in C++ are user-defined data types that are used to store a group of elements of different data types.

What is a structure?

A structure is a user-defined data type in C/C++. A structure creates a data type that can be used to group elements of possibly different types into a single type.

```
struct student
{
char name[10] ;
int id[5] ;
float salary ;
}
```

III.5.1) Creating a structure

The 'struct' keyword is used to create a structure. The general syntax for creating a structure is as follows:

```
struct structureName{
    member1;
    member2;
    member3;
    .
    .
    .
    memberN;
};
```

Structures in C++ can contain two types of members:

- **Data Member**: These members are normal C++ variables. We can create a structure with variables of different data types in C++.
- **Functions Member**: These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.

Example

```
// Data Members
int num;
int age;
float average;
// Member Functions
void printDetails()
{
    cout<<"num = "<<num<<"\n";
    cout<<"age = "<<age<<"\n";
    cout<<"average = "<<average;
}
```

In the above structure, the data members are two integer variables to store number and age and a float variable to store the average of any student and the member function is printDetails() which is printing all of the above details of any student.

III.5.2) Declaration of structure variables

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
    struct Point
    {
       int x, y;
    };
     int main()
    {
       struct Point p1;  // The variable p1 is declared like a
normal variable.
    }
```

Note: In C++, the struct keyword is optional before declaring a variable. In C, it is mandatory.

III.5.3) Initializing members of a structure

Structure members can be initialized with declaration in C++.

```
    #include <iostream>
    using namespace std;
     struct Point {
        int x = 0;
        int y = 1;
    //It is Considered as Default Arguments
    };
     int main()
    {
        struct Point p1;
        // Accessing members of point p1
        // If no value is Initialized then default value is
considered. x=0 and y=1 ;
        cout << "x = " << p1.x << ", y = " << p1.y<<endl;
        // Initializing the value of y = 20 ;
        p1.y = 20;
        cout << "x = " << p1.x << ", y = " << p1.y;
        return 0;
    }
```

Output

```
x=0, y=1

x=0, y=20
```

Structure members can be initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point {
    int x, y;
};
 int main()
{
    // A valid initialization. member x gets value 0 and y
    // gets value 1.  The order of declaration is followed.

    struct Point p1 = { 0, 1 };
}
```

III.5.4) Accessing structure elements

Structure members are accessed using dot (.) operator.

```
#include <iostream>
using namespace std;
struct Point {
    int x, y;
};
int main()
{
    struct Point p1 = { 0, 1 };
    // Accessing members of point p1
    p1.x = 20;
    cout << "x = " << p1.x << ", y = " << p1.y;
    return 0;
}
```

Output

```
x = 20, y = 1
```

III.5.5) Array of structures

Like other primitive data types, we can create an array of structures.

```
#include <iostream>
using namespace std;
struct Point {
    int x, y;
};
 int main()
{
    // Create an array of structures
    struct Point arr[10];
```

```
        // Access array members
        arr[0].x = 10;
        arr[0].y = 20;
        cout << arr[0].x << " " << arr[0].y;
        return 0;
    }
```

Output

```
10 20
```

III.5.6) Example of C++ structure

```
    #include <iostream>
    using namespace std;

    struct Person
    {
        //char name[50];
        string name;
        int age;
        float salary;
    };
    int main()
    {
        Person p1;
        cout << "Enter Full name: ";
        getline(cin,p1.name);//String
        //cin.getline(p1.name,50);//char
        cout << "Enter age: ";
        cin >> p1.age;
        cout << "Enter salary: ";
        cin >> p1.salary;

        cout << "\nDisplaying Information." << endl;
        cout << "Name: " << p1.name << endl;
        cout <<"Age: " << p1.age << endl;
        cout << "Salary: " << p1.salary;

        return 0;
    }
```

Output

Enter Full name: Abd al-Razzaq al-Badr
Enter age: 60
Enter salary: 1024.4

Displaying Information.
Name: Abd al-Razzaq al-Badr
Age: 60
Salary: 1024.4

III.6) Set in C++

The **std::set** class is the part of C++ Standard Template Library (STL)

Sets are a type of container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending.

Syntax:

set<type> set_name ;

Data type: Set can take any data type depending on the values, eg. int, char, float, etc.

Example

Let's take the example of a declaration with initialization

```
set<char> s1={'C', 'D'};
```

Or separate declaration and initialization

```
set<int> val;//defining an empty set
val={6,10,5,1};//initialization of a set with values
```

Properties

- The set stores elements in a **sorted** order.
- All elements in a set have **unique values.**
- The value of the element cannot be modified once it has been added to the set (elements are always constant), although it is possible to delete and then add the modified value of this element. Thus, the values are **immutable** (not modifiable).
- Set values are **not indexed**.
- By default, the set is sorted in **ascending** order. However, we have the option to change the sorting order by using the following syntax.

```
set <data_type, greater<data_type>> set_name;
```

Example

```
// C++ program to demonstrate the creation of descending
// order set container
#include <iostream>
#include <set>
using namespace std;
```

```
int main()
{
 set<int, greater<int> > s1;
 s1.insert(10);
 s1.insert(5);
 s1.insert(12);
 s1.insert(4);
 for (auto i: s1) {
      cout << i << ' ';
 }
 return 0;
}
```

Output

12 10 5 4

III.6.1) Browsing a set

Iterators make it easy to browse a set from one end to the other.

- **begin()** - Returns an iterator to the first element in the set.
- **end()** - Returns an iterator to the theoretical element that follows the last element in the set.
- **size()** - Returns the number of elements in the set.
- **max_size() -** Returns the maximum number of elements that the set can hold.
- **empty()** - Returns whether the set is empty.
- **++**: increments the iterator to the next element.

To display values

```
set<char>::iterator itr;
for (itr=s1.begin();itr!=s1.end();itr++)
 cout<<*itr<<" ";
```

Output

C D

There are no duplicates in the sets.

```
set<char> s1={'C','D','C','D','A'};
set<char>::iterator itr;
for (itr=s1.begin();itr!=s1.end();itr++)
 cout<<*itr<<" ";
```

Output

A C D

III.6.2) Insert a value into a set

To insert into a set

```
s1.insert('B') ;
```

III.6.3) Erase a value from a set

To erase from a set

```
s1.erase('C') ;
```

III.6.4) Search in a set

To search for an element in a set

```
if (s1.find('C')==s1.end())
cout <<"not found"<<endl;
else cout <<"found";
```

Example

```
// C++ program to demonstrate various functions of sets
#include <iostream>
#include <iterator>
#include <set>
using namespace std;
int main()
{
 // s1 empty set
 set<int,greater<int>> s1;
 // insert elements in random order
 s1.insert(40);
 s1.insert(30);
 s1.insert(60);
 s1.insert(20);
 s1.insert(50);
 // only one 50 will be added to the set
 s1.insert(50);
 s1.insert(10);
 // printing set s1
 set<int,greater<int>>::iterator itr;
 cout << "The set s1 is : "<<endl;
 for (itr = s1.begin(); itr != s1.end(); itr++) {
      cout << *itr << " ";
 }
 cout << endl;

 // assigning the elements from s1 to s2
 set<int> s2(s1.begin(), s1.end());
```

```
     // Print all elements of the set s2
     cout << "The set s2 after assign s1 is : "<<endl;
     for (itr = s2.begin(); itr != s2.end(); itr++) {
          cout << *itr << " ";
     }
     cout << endl;
     // remove all elements up to 30 in s2
     cout << "s2 after removal of elements less than 30 :
"<<endl;
     s2.erase(s2.begin(), s2.find(30));
     for (itr = s2.begin(); itr != s2.end(); itr++) {
          cout << *itr << " ";
     }
     // remove element with value 50 in s2
     s2.erase(50);
     cout <<endl<< "s2.erase 50 : "<<endl;
     for (itr = s2.begin(); itr != s2.end(); itr++) {
          cout << *itr << " ";
     }
     cout << endl;

    // lower bound and upper bound for set s1
        cout << "s1.lower_bound(40) : "
             << *s1.lower_bound(40) << endl;
        cout << "s1.upper_bound(40) : "
             << *s1.upper_bound(40) << endl;
         // lower bound and upper bound for set s2
        cout << "s2.lower_bound(40) : "
             << *s2.lower_bound(40) << endl;
        cout << "s2.upper_bound(40) : "
             << *s2.upper_bound(40) << endl;
     return 0;
    }
```

Output

The set s1 is:

60 50 40 30 20 10

The set s2 after assign from s1 is:

10 20 30 40 50 60

s2 after removal of elements less than 30:

30 40 50 60

s2. erase (50): 1 element removed

30 40 60

s1.lower_bound(40) : 40

s1.upper_bound(40) : 30

s2.lower_bound(40) : 40

s2.upper_bound(40) : 60

Table 13 - Different Function of Set in C++ STL

Function	Description
begin()	Returns an iterator to the first element in the set.
end()	Returns an iterator to the theoretical element that follows the last element in the set.
rbegin()	Returns a reverse iterator pointing to the last element in the container.
rend()	Returns a reverse iterator pointing to the theoretical element right before the first element in the set container.
crbegin()	Returns a constant iterator pointing to the last element in the container.
crend()	Returns a constant iterator pointing to the position just before the first element in the container.
cbegin()	Returns a constant iterator pointing to the first element in the container.

Function	Description
cend()	Returns a constant iterator pointing to the position past the last element in the container.
size()	Returns the number of elements in the set.
max_size()	Returns the maximum number of elements that the set can hold.
empty()	Returns whether the set is empty.
insert(const g)	Adds a new element ‘g’ to the set.
iterator insert (iterator position, const g)	Adds a new element ‘g’ at the position pointed by the iterator.
erase(iterator position)	Removes the element at the position pointed by the iterator.
erase(const g)	Removes the value ‘g’ from the set.
clear()	Removes all the elements from the set.
key_comp() / value_comp()	Returns the object that determines how the elements in the set are ordered (‘<‘ by default).
find(const g)	Returns an iterator to the element ‘g’ in the set if found, else returns the iterator to the end.

Function	Description
count(const g)	Returns 1 or 0 based on whether the element ‘g’ is present in the set or not.
lower_bound(const g)	Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the set.
upper_bound(const g)	Returns an iterator to the first element that will go after the element ‘g’ in the set.
equal_range()	The function returns an iterator of pairs. (key_comp). The pair refers to the range that includes all the elements in the container which have a key equivalent to k.
emplace()	This function is used to insert a new element into the set container, only if the element to be inserted is unique and does not already exist in the set.
emplace_hint()	Returns an iterator pointing to the position where the insertion is done. If the element passed in the parameter already exists, then it returns an iterator pointing to the position where the existing element is.
swap()	This function is used to exchange the contents of two sets but the sets must be of the same type, although sizes may differ.
operator=	The ‘=’ is an operator in C++ STL that copies (or moves) a set to another set and set::operator= is the corresponding operator function.

Function	Description
get_allocator()	Returns the copy of the allocator object associated with the set.

Chapter IV. Functions

IV.1) Introduction

A function is a set of instructions that takes inputs, performs a specific calculation and produces an output.

The idea is to group together some common or repeated tasks and make a function out of them. So, instead of writing the same code over and over again for different inputs, we can call the function.

The advantage of using functions is that you can reduce the size of a program by calling and using them at different points in the program.

IV.1.1) Benefits

- Support for modular programming
- Reduced program size.
- Code duplication is avoided.
- Code reuse is assured.
- Functions can be called repeatedly.
- A set of functions can be used to form libraries.

IV.1.2) Types of functions

IV.1.2.1) Built-in functions

Found in packages supplied with the compiler.

They are part of the standard library provided by the compiler.

They can be used in any program by including the corresponding header file.

IV.1.2.2) User-defined functions

Created by the user or programmer.
Created according to program requirements.

IV.2) Function definition

The general form of a function is :

```
type identifier(parameters) { code }
```

- "**Type**" is the type of the returned value.
- "**Identifier**" is the name of the function.
- "**Parameters**" is a list of parameters [type_param1 nom_param1,…].

A definition = header + body

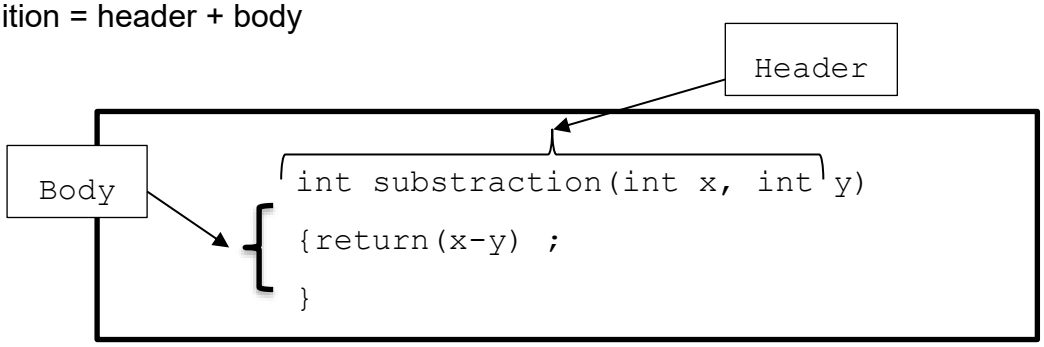

Figure 13 - Function definition

IV.3) Function declaration (Prototypes)

The prototype describes the function header to the compiler, giving details such as the function **name**, the **number** and **type** of arguments and the type of **return values.**

A template is used when declaring and defining a function.

When a function is called, the compiler uses the template to ensure that the appropriate arguments are passed and that the return value is processed correctly.

Example :

```
    float volume(int x, float y, float z) ;
    float volume(int, float, float) ; //is also acceptable in
the the declaration
    void display() ; is identical to void display(void) ;
```

A function must be declared before it is called.

```
    // function delcaration
    int substraction(int,int);
    int main(){
    int x=11, y=10, z=0;
    // function call
    z = substraction(x,y);
    cout<<"z = "<<z<<endl;
    return 0;
    }// function definition
    int substraction(int a, int b) {
    return (a-b);
    }
```

To call a function, enter its name and parameter values in brackets.

IV.4) Prototype and definition at the same time

Write the definition before the first function call (its use).

// Prototype and definition of the function

```
    int substraction(int a, int b) {
    return (a-b);
    }
    int main(){
    int x=11, y=10, z=0;
    // function call
    z = substraction(x,y);
    cout<<"z = "<<z<<endl;
    return 0;
    }
```

Exemple

```
    #include <iostream>
    using namespace std;
    // An example of a function that takes two parameters 'x'
and 'y'.
    // and returns the maximum of the two numbers entered.
    int max(int x, int y)
    {
     if (x > y)
     return x;
     else
     return y;
    }
    // main function that receives no parameters and
    // returns an integer.
    int main(void)
    {
     int a = 10, b = 20;
     // Call the above function to find the maximum of 'a' and
'b'.
     int m = max(a, b);
     cout<< "m= "<<m<<endl;
     return 0;
    }
```

IV.5) Function categories

IV.5.1) Function without parameters but with a return value

Just don't put anything in brackets.

int function_test()

Or

int function_test(void)

```
    int add(void)
```

```
    {
    int a,b ;
    cout<< "enter a and b values "<<endl ;
    cin>> a>>b ;
    return(a+b) ;
    }
    int main()
    {
    int c ;
    c=add();
    cout<< "The sum is "<<c<<endl ;
    return 0 ;
    }
```

IV.5.2) Function with parameters but no return value

In C++, declaring the type of the returned value is **MANDATORY.**

```
    //void function_test(int a,double b);
    void add(int x,int y)
    {
    int c;
    c=x+y;
    cout<< "The sum is "<<c<<endl;
    }
    int main()
    {
    int a=0,b=0;
    cout<< "enter a and b values "<<endl;
    cin>> a>>b;
    add(a,b);
    return 0;
    }
```

IV.5.3) Function with no parameters and no return value

```
    /*void function_test()
    Or
    void function_test(void)*/
    void sqr()
    {
    int nb ;
    cout<< "enter a number "<<endl ;
    cin>> nb ;
    cout<< "The square of "<<nb<<" is "<<nb*nb;//The function
has no return value
    } ;
```

```
int main()
{
sqr() ; //No parameter
return 0 ;
}
```

IV.5.4) Function with parameters and return value

```
//int function_test(int,int)
int sqr(int x)
{
return (x*x);
}
int main()
{
int a,res;
cout<< "enter a number "<<endl;
cin>> a;
res=sqr(a);
cout<< "The square of "<<a<<" is "<<res;
return 0;
};
```

IV.6) Function calls

There are two most popular ways to pass parameters when calling a function.

IV.6.1) Pass by value

In this parameter passing method, values of actual parameters are copied to the function's newly created functional parameters. The actual function parameters are stored in different memory locations so any changes made in the functions are not reflected in the actual parameters of the caller.

Example

```
#include <iostream>
using namespace std;
void swap(int x,int y)
{
int temp;
temp=x;
x=y;
y=temp;
};
int main()
{
int a,b;
cout<<"Enter a and b values"<<endl;
```

```
    cin>>a>>b;
    cout<<"before"<<endl;
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    swap(a,b);
    cout<<"after"<<endl;
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    return 0;
    }
```

Output

Enter a and b values

2

5

Before

a = 2

b = 5

After

a = 2

b = 5

IV.6.2) Pass by reference

When we pass arguments by reference, the arguments of the called function become **aliases** (pseudonyms) of the "real" arguments. The function works on the **original** data.

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting **'&'** in the declaration.

Declaration example

```
    int C=0 ;
    int a=100,b=20 ;
    int &ref=a;
    C=a+b ; //In both cases C will have the same values
    cout<<"C = "<<C<<endl;
    C=ref+b;
    cout<<"C = "<<C<<endl;
```

Output

C = 120

C = 120

Example

```
int C=0;
int a=100,b=20;
int &ref=a;
C=a+b;
cout<<"C = "<<C<<endl;
ref=90;
C=a+b;
cout<<"C = "<<C<<endl;
```

Output

C = 120

C = 110

Function example

```
#include <iostream>
using namespace std;
void swap(int &x,int &y)
{
int temp;
temp=x;
x=y;
y=temp;
} ;
int main()
{
int a,b;
cout<<" Enter a and b values"<<endl;
cin>>a>>b;
cout<<"before"<<endl;
cout<<"a = "<<a<<endl;
cout<<"b = "<<b<<endl;
swap(a,b);
cout<<"after"<<endl;
cout<<"a = "<<a<<endl;
cout<<"b = "<<b<<endl;
return 0;
}
```

Output

Enter a and b values

2

5

Before

a = 2

b = 5

After

a = 5

b = 2

IV.6.3) Calls summary

Table 14 - Calls Summary

Pass by value vs Pass by reference	
A copy of the value is passed to the function thus Changes made inside the function are not reflected on original values	In call by reference, the address of the variables is passed to the function. Consequently, changes made to variables within the function will modify the original value...
Changing values	
When calling by value, the original value remains unchanged.	In call by reference, the original values change.

IV.6.4) Passing Array to Function

Passing an array as a parameter to a function is impossible as a value: copying the array would take too much time and space. We therefore we need to provide only the array name, which will enable the function to read and write DIRECTLY TO THE ARRAY.

IV.7) Scope of variables

In programming the scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes:

1. **Local Variables**
2. **Global Variables**

IV.7.1) Local Variables

Variables defined within a function or block are said to be local to those functions.

- Anything between '{' and '}' is said inside a block.
- Local variables do not exist outside the block in which they are declared, i.e.

they can not be accessed or used outside that block.

- Declaring local variables: Local variables are declared inside a block.

In particular, local variables have no connection with global variables of the same name. For example, the following program

```
//global variable
int n = 10;
void func()
{
//local variable with the same name
int n = 0;
n++;
cout<<"Call number "<<n<<endl;
}
int main()
{
int i;
for (i = 0; i < 5; i++)
func();
cout<<"Global variable n="<<n<<endl;
return 0;
}
```

Output

Call number 1

Call number 1

Call number 1

Call number 1

Call number 1

Global variable n=10

Local variables to a function have a lifetime limited to a single execution of that function. Their values are not retained from one call to the next.

IV.7.2) Variables globales

As the name suggests, Global Variables can be accessed from any part of the program.

- They are available through out the life time of a program.
- They are declared at the top of the program outside all of the functions or blocks.
- **Declaring global variables**: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

Global variables are always permanent. In the following program, n is a global variable:

```
//global variable
int n=0;
void func()
{
n++;
cout<<"Call number "<<n<<endl;
}
int main()
{
int i;
for (i = 0; i < 5; i++)
func();
cout<<"Global variable n="<<n<<endl;

return 0;
}
```

The variable n is initialized to zero by the compiler and is a permanent variable. In fact, the program displays

Output

Call number 1

Call number 2

Call number 3

Call number 4

Call number 5

Global variable n=5

IV.8) Difference between Argument and Parameter

IV.8.1) Parameters

The parameter is referred to as the variables that are defined during a function declaration or definition. These variables are used to receive the arguments that are passed during a function call. These parameters within the function prototype are used during the execution of the function. These are also called **Formal Parameters.**

Example Suppose a Mult() function is needed to be defined to multiply two numbers. These two numbers are referred to as the parameters and are defined while defining the function Mult().

```
#include <iostream>
using namespace std;
// Mult: Function definition
// a et b are the parameters
```

```
int Mult(int a, int b)
{
    // returning the multiplication
    return a * b;
}
int main()
{
    int num1 = 10, num2 = 20, res;
    // Mult()
    // num1 & num2 are arguments.
    res = Mult(num1, num2);
    // Displaying the result
    cout << "The multiplication is" << res;
    return 0;
}
```

Output

The multiplication is 200

IV.8.2) Arguments

An argument refers to the values that are passed into a function when the function is called. These values are generally the source of the function that requires the arguments during the execution process. These values are assigned to variables in the definition of the function called. The type of values passed into the function is the same as that of the variables defined in the function definition. They are also known as **real parameters.**

Example: Suppose a sum() function is needed to be called with two numbers to add. These two numbers are referred to as the arguments and are passed to the sum() when it is called from somewhere else.

```
#include <iostream>
using namespace std;
// sum: function definition
// a and b are the parameters
int sum(int a, int b)
{
    // calculate the sum
    return a + b;
}
int main()
{
    int num1 = 10, num2 = 20, res;
    // sum() call
    // num1 & num2 are arguments.
    res = sum(num1, num2);
```

```
        // Displaying the result
        cout << "The sum is " << res;
        return 0;
    }
```

Output

The sum is 30

IV.8.3) Summary

Table 15 - Argument vs Parameter

Argument	Parameter
When a function is called, the values that are passed during the call are called as arguments.	The values which are defined at the time of the function prototype or definition of the function are called as parameters.
During the time of call each argument is always assigned to the parameter in the function definition.	Parameters are local variables which are assigned value of the arguments when the function is called.
They are also called Real Parameters	They are also called Formal Parameters

int num = 20;
Call(num)
 	
// num is an argument
	int Call(int num)
	{
	cout<<"num = "<<num<<endl;
	}
	
	// num is a parameter

Conclusion

In conclusion, this course offers a comprehensive foundation in algorithmics and programming for first-year computer science students. Beginning with essential algorithmic principles, students are introduced to the basics of problem-solving through algorithmic structures. The transition to C++ allows students to apply these foundational concepts in a practical programming context, reinforcing their understanding of program structure, control flows, and language-specific features. Advanced topics such as arrays, strings, and functions deepen students' understanding, equipping them with the skills to handle complex data types and functional programming techniques. Overall, the course prepares students with both the theoretical and practical skills necessary for further studies in computer science.

References

[1] B. Bessaa, *Exercices corrigés d'Algorithmique*. Algiers: Pages Bleues, The LMD Booklets, 2018.

[2] C. Haro, *Algorithmique raisonner pour concevoir*. ENI Edition, 2015.

[3] T. H. Cormen, *Algorithms Unlocked*. MIT Press, 2013.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[5] D. Bouchicha, *Initiation à l'algorithmique et à la programmation en Pascal*. Sidi Bel Abbes, Algeria: Rached Edition, 1st ed., 2019.

[6] D. Zegour, *Apprendre et enseigner l'algorithmique (Tome 1): Cours et annexes*. European University Editions, 2013.

[7] Edube Interactive, "C++ essentials, part 1 (intermediate)," Edube, Nov. 13, 2024. [Online]. Available: https://edube.org/study/cppe1

[8] GeeksforGeeks, "C++ tutorial," Nov. 13, 2024. [Online]. Available: https://www.geeksforgeeks.org/c-plus-plus/?ref=shm

[9] J. Tisseau, *Initiation à l'algorithmique*. University Press - National School of Engineers Brest, 2009.

[10] J. Farrell, *Programming Logic and Design: Comprehensive Version*, 8th ed. Cengage Learning, 2015.

[11] D. E. Knuth, *The Art of Computer Programming, Volumes 1-4*. Addison-Wesley, 1997.

[12] L. Baba Hamed and S. Hocine, *Algorithmique et structure de données statiques: cours et exercices avec solutions*. University Publications Office, Algiers, 2006.

[13] M. Amad, *Algorithmique et Structures de Données*. Abderrahmane Mira University of Bejaia, Course Material and Directed Exercises, 1st and 2nd year License, 2016.

[14] R. Malgouyres, R. Zrour, and F. Feschet, *Initiation à l'algorithmique et à la programmation en C - Cours avec 129 exercices corrigés*, 2nd ed. DUNOD, 2015.

[15] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley, 2011.

[16] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley, 2013.

[17] T. H. Cormen, *Algorithmes Notions de base*. DUNOD, 2013.

[18] V. Felea and V. Felea, *Introduction à l'informatique: Apprendre à concevoir des algorithmes - Cours et problèmes corrigés*. Vuibert, 1st ed., 2013.

[19] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Pearson, 2013.