

الجمهورية الجزائرية الديمقراطية الشعبية

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND
SCIENTIFIC RESEARCH

UNIVERSITY OF 8 MAI 1945 GUELMA

Faculty of Mathematics, Computer
science, and Material Sciences

Department of Computer Science



وزارة التعليم العالي
و البحث العلمي

جامعة 8 ماي 1945 قالمة

كلية الرياضيات، الإعلام الآلي
و علوم المادة

قسم: الإعلام الآلي

Lecture notes

Algorithmics and Data Structures 1

Intended for first-year undergraduate students in Mathematics

Established by :

Dr. Abderrahmane KEFALI

kefali.abderrahmane@univ-guelma.dz

kefali.inf@gmail.com



Data Structures
And Algorithms

2023/2024

Syllabus (Course Plan)

Course Unit: UEF11 (Fundamental)

Subject: Algorithmics and Data Structures 1

Field/Branch: Mathematics and Computer Sciences

Semester: 1, **Academic Year:** 2023/2024

Credits: 6, **Coefficient:** 4

Total Weekly Hours: 07H30

- Lectures (03H per week)
- Tutorials (01H30 per week)
- Practical Work (03H per week)

Language of Instruction: English

Course Instructor: Dr. Abderrahmane KEFALI, **Title:** MCA

Office:, Teachers Rooms, Department of Computer Science (E8)

Email: kefali.abderrahmane@univ-guelma.dz

Course Objectives:

To present the concepts of algorithms and data structures.

Recommended Prerequisite Knowledge:

Basic knowledge of computer science and mathematics.

Bureau : E8. Salle des Enseignants département d'informatique

Course Content:

Chapter 1: Introduction

- Brief history of computer science.
- Introduction to algorithms.

Chapter 2: Simple Sequential Algorithm

- Notion of language and algorithmic language.
- Parts of an algorithm.
- Data: variables and constants.
- Data types.
- Basic operations.
- Basic instructions.
- Construction of a simple algorithm.
- Representation of an algorithm using a flowchart.
- Translation into the C language.

Chapter 3: Conditional Structures (in algorithmic language and in C)

- Introduction.
- Simple conditional structure.
- Compound conditional structure.
- Multiple choice conditional structure.
- Branching.

Chapter 4: Loops (in algorithmic language and in C)

- The While loop.
- The Repeat loop.
- The For loop.
- Nested loops.

Chapter 5: Arrays and Strings

- Introduction.
- The array type.
- Multidimensional arrays.
- Strings.

Chapter 6: Custom Types

- Introduction.
- Enumerations.
- Records (Structures).
- Other type definition possibilities.

Assessment Method: Knowledge Assessment & Weightings

Assessment Method	Weight (%)
Final examen	60%
Tutorial works	20%
Practical works	20%
Total	100%

Bibliography:

- Thomas H. Cormen, Algorithmes Notions de base *Collection : Sciences Sup, Dunod*, 2013.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest *Algorithmique - 3ème édition - Cours avec 957 exercices et 158 problèmes Broché, Dunod*, 2010.
- Rémy Malgouyres, Rita Zrour et Fabien Feschet. *Initiation à l'algorithmique et à la programmation en C : cours avec 129 exercices corrigés*. 2ième Edition. Dunod, Paris, 2011. ISBN : 978-2-10-055703-5.
- Damien Berthet et Vincent Labatut. *Algorithmique & programmation en langage C - vol.1 : Supports de cours*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.232.
- Damien Berthet et Vincent Labatut. *Algorithmique & programmation en langage C - vol.2 : Sujets de travaux pratiques*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.258. <cel- 01176120>
- Damien Berthet et Vincent Labatut. *Algorithmique & programmation en langage C - vol.3 : Corrigés de travaux pratiques*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.217. <cel-01176121>
- Claude Delannoy. *Apprendre à programmer en Turbo C*. Chihab- EYROLLES, 1994.

Preface

This document serves as the course and exercise material for the "*Algorithmics and Data Structures 1*" course. It is primarily intended for first-year students in the first-year Mathematics students in the Department of Mathematics at the Faculty of Mathematics and Computer Science and Material Sciences at the University of May 8, 1945, Guelma. Nevertheless, this handout can also be useful for any non-computer science student wishing to learn the fundamentals of algorithmics and programming.

The "Algorithmics and Data Structures 1" course is scheduled in the first semester of the first year for Mathematics in the fundamental teaching unit UEF11. It is characterized by a coefficient of 4, 6 credits, and a weekly hourly volume of 7 and a half hours, divided as follows: 3 hours of lectures, one and a half hours of tutorials (TD), and 3 hours of practical work (TP).

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. Thus, these two concepts (algorithms and data structures) are complementary. The first pertains to the dynamic aspect of task automation, and the second to the static aspect. The study of algorithmics and data structures 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 a problem.

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 Common Core in Mathematics, Applied Mathematics, and Computer Science, as proposed by the national pedagogical committee of the MI domain since the academic year 2018 - 2019.

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 and data structures more accessible to novice students. As such, no prerequisites are required to follow this document.

In order to present its content effectively and to adhere as closely as possible to the compliance framework, this handout is structured into six chapters, each concluding with a set of solved exercises.

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.

The second chapter, "Simple Sequential Algorithm," describes the fundamentals of a simple algorithm written in pseudocode. It explains the general structure of an algorithm in pseudocode and introduces its basic elements (variables, expressions, instructions, etc.). The chapter also covers the rules for constructing a flowchart and a C language program based on an algorithm.

The third chapter introduces conditional structures. These are special instructions that allow for the handling of complex problems involving multiple cases, where each case requires separate processing. Various types of conditional structures are presented in this chapter, along with explanations of their use.

The fourth chapter, titled "Loops," introduces repetitive structures or loops. These are control structures that allow the execution of a sequence of instructions to be repeated a finite number of times, which may be known in advance or not. The three main types of loops are described in this chapter, including their use and the differences between them.

The fifth chapter 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 both algorithmics and the C language.

The last chapter, titled "Custom Types," addresses the possibility of defining new data types in algorithmics and in C language. Several custom data types are presented in this chapter, with a focus on record and enumeration types. Other types are briefly described.

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

Table of Contents

Syllabus (Course Plan)	i
Preface	iii
Table of Contents	1
List of Figures	7
List of Tables	9
Chapter I. Introduction	10
I.1) Introduction.....	10
I.2) Computer Science: Definitions and a Brief History.....	10
I.2.1) Definitions	10
I.2.1.1) What is Computer Science?.....	10
I.2.1.2) What is Information?	11
I.2.1.3) Information Processing.....	12
I.2.2) Computer	12
I.2.2.1) Components of a Computer	12
I.2.2.2) Basic Computer Architecture.....	14
I.2.3) Utility of Computing	14
I.2.4) A Brief History of Computer science.....	15
I.3) Introduction to Algorithms.....	19
I.3.1) Introductory Examples	19
I.3.1.1) Example 1: Fax Machine User Manual	19
I.3.1.2) Example 2: Calculating the Average of 3 Numbers with a Calculator.....	19
I.3.1.3) Example 3: Giving Directions	20
I.3.2) Notion of Algorithm	20
I.3.2.1) What is an Algorithm?.....	20
I.3.2.2) Object of Algorithmics	21
I.3.2.3) Qualities of an Algorithm.....	21
I.3.3) Problem-Solving Steps.....	22
I.3.3.1) Problem Definition and Analysis	22
I.3.3.2) Establishment of the Corresponding Algorithm	23
I.3.3.3) Translating the Algorithm into a Computer Program	23
I.3.3.4) Compiling the Program.....	23
I.3.3.5) Program Execution and Testing.....	23
I.4) Exercises	23
I.5) Solution of the exercises	24
I.6) Conclusion	26

Chapter II. Simple sequential algorithm.....	27
II.1) Introduction.....	27
II.2) Concept of Language and Algorithmic Language	27
II.2.1) Language.....	27
II.2.2) Computer Language	28
II.2.3) Machine Language	28
II.2.4) Algorithmic Language	28
II.2.4.1) What is an Algorithmic Language?	28
II.2.4.2) Elements of Algorithmic Language	29
II.2.4.3) Difference between algorithm and program	29
II.3) Parts of an algorithm	29
II.3.1) Algorithm Header.....	30
II.3.2) The Declaration Section	30
II.3.3) Algorithm Body.....	30
II.4) Data: Variables and Constants	30
II.4.1) Variables	30
II.4.2) Constants.....	31
II.4.3) Notion of Identifier	32
II.5) Data Types	32
II.5.1) Elementary types.....	33
II.5.1.1) Standard types	33
II.5.1.2) Non-standard Types (ou non-prédéfinis)	34
II.5.2) Declaration of variables and constantes.....	34
II.5.2.1) Declaration of variables	34
II.5.2.2) Declaration of constants.....	35
II.6) Basic Operations	36
II.6.1) Operator and Operand	36
II.6.1.1) Arithmetic Operators	36
II.6.1.2) Logical Operators	36
II.6.1.3) Comparison Operators	36
II.6.2) Expression.....	37
II.6.2.1) Validity of an expression.....	37
II.6.2.2) Evaluation of an expression.....	37
II.7) Basic instructions	38
II.7.1) Assignment instruction	38
II.7.2) Input/Output instructions	39
II.7.2.1) Reading (input) instruction.....	39
II.7.2.2) Writing (output) instruction	40
II.8) Building a simple algorithm.....	41
II.9) Representation of an Algorithm Using a Flowchart	42
II.10) Translation into C language.....	43
II.10.1) The C language: Presentation.....	43

II.10.2)	Why the C language ?	44
II.10.3)	Basic Elements of the C language	44
II.10.3.1)	Structure of a C program	44
II.10.3.2)	Declaration section	45
II.10.3.3)	Processing Section	46
II.10.3.4)	Example of a C Program	49
II.11)	Exercises	49
II.12)	Solution of the exercises.....	50
II.13)	Conclusion.....	52
Chapter III. Conditional Structures	53	
III.1)	Introduction	53
III.2)	Notion of Condition	53
III.2.1)	Simple Conditions.....	53
III.2.2)	Compound Conditions	54
III.3)	Simple Conditional Structures (IF Statements)	54
III.3.1)	Algorithmic Syntax	54
III.3.2)	Flowchart	55
III.3.3)	C language Syntax	56
III.4)	Compound Conditional Structures (IF - ELSE statements)	57
III.4.1)	Algorithmic Syntax	57
III.4.2)	Flowchart	58
III.4.3)	C language Syntax	58
III.5)	Nested conditional Structures	59
III.5.1)	Algorithmic Syntax	59
III.5.2)	Flowchart	60
III.5.3)	C language Syntax	60
III.6)	Multiple-choice structure (CASE statement)	61
III.6.1)	Algorithmic Syntax	61
III.6.2)	Flowchart	62
III.6.3)	C language Syntax	63
III.7)	Branching statement	64
III.7.1)	Algorithmic Syntax	64
III.7.2)	C language Syntax	65
III.8)	Exercises	66
III.9)	Solution of the exercises.....	66
III.10)	Conclusion.....	68
Chapter IV. Loops.....	69	
IV.1)	Introduction	69
IV.2)	What is a loop ?.....	70
IV.2.1)	Definition	70
IV.2.2)	Components of a loop	70
IV.3)	While loop	71

IV.3.1)	Algorithmic Syntax	71
IV.3.2)	Flowchart	72
IV.3.3)	C language syntax.....	73
IV.4)	REPEAT loop	74
IV.4.1)	Algorithmic syntax.....	74
IV.4.2)	Flowchart	75
IV.4.3)	C language syntax.....	76
IV.4.4)	Difference between WHILE and REPEAT	76
IV.5)	For loop	77
IV.5.1)	Algorithmic Syntax	77
IV.5.2)	Flowchart	79
IV.5.3)	C language Syntax	80
IV.6)	Choice of the appropriate repetitive structure.....	81
IV.7)	Nested loops.....	81
IV.8)	Exercises	82
IV.9)	Solution of the exercises.....	83
IV.10)	Conclusion.....	85
Chapter V. Arrays and Strings	86	
V.1)	Introduction.....	86
V.2)	The Array type.....	87
V.2.1)	Definitions	87
V.2.2)	Declaration.....	88
V.2.2.1)	Algorithmic Syntax	89
V.2.2.2)	C language Syntax	90
V.2.3)	Manipulation of arrays	90
V.2.3.1)	Accessing Array Elements.....	90
V.2.3.2)	Filling an array	91
V.2.3.3)	Displaying the contents of an Array	94
V.3)	Multidimensional Arrays.....	95
V.3.1)	Definition	96
V.3.2)	Declaration.....	97
V.3.2.1)	Algorithmic Syntax	97
V.3.2.2)	C language Syntax	97
V.3.3)	Manipulation of Multidimensional Arrays	98
V.3.3.1)	Accessing elements of a multidimensional array	98
V.3.3.2)	Filling a multidimensional array.....	98
V.3.3.3)	Displaying the elements of a multidimensional array	101
V.4)	Strings of Characters	101
V.4.1)	Reminder about Characters.....	102
V.4.1.1)	Definition.....	102
V.4.1.2)	Presentation of Characters	102
V.4.2)	Definition of a String of Characters	103

V.4.3)	Strings in Algorithmics.....	104
V.4.3.1)	Declaration.....	104
V.4.3.2)	Memory Representation.....	104
V.4.3.3)	Manipulating strings	104
a)	Accessing a character in the string.....	104
b)	Reading.....	105
c)	Writing.....	105
d)	Assignment	106
e)	Operations specific to strings of characters	106
V.4.4)	Strings in C language	108
V.4.4.1)	Declaration.....	108
V.4.4.2)	Memory Representation.....	108
V.4.4.3)	Manipulating strings	109
a)	Accessing a character in the string.....	109
b)	Reading.....	109
c)	Writing.....	110
d)	Assignment	110
e)	Operations specific to strings of characters	111
V.5)	Exercises.....	112
V.6)	Solution of the exercises	113
V.7)	Conclusion	115
Chapter VI.	Custom Types	116
VI.1)	Introduction	116
VI.2)	Concept of Data Type.....	116
VI.2.1)	Definition.....	116
VI.2.2)	Type declaration.....	117
VI.3)	Enumerations.....	117
VI.3.1)	Definition.....	117
VI.3.2)	Enumerations in algorithmics.....	117
VI.3.2.1)	Declaration.....	117
VI.3.2.2)	Manipulation.....	118
a)	Assignment	118
b)	Predefined Functions	119
c)	Using.....	119
VI.3.3)	Enumerations in the C language	120
VI.3.3.1)	Declaration.....	120
a)	Declaration of an Enumeration.....	120
b)	Declaration using the typedef keyword.....	121
VI.3.3.2)	Manipulation.....	121
a)	Reading and writing.....	121
b)	Assignment	122
c)	Using.....	122

VI.4)	Records (structures)	123
VI.4.1)	Definition	123
VI.4.2)	Records in algorithmics.....	124
VI.4.2.1)	Declaration.....	124
VI.4.2.2)	Manipulating a record	125
a)	Accessing a Field of a Record.....	125
b)	Reading and writing.....	126
c)	Assignment	126
d)	The WITH...DO statement.....	127
VI.4.2.3)	Nesting of Records	128
VI.4.2.4)	Arrays of records.....	129
a)	Declaration.....	129
b)	Accessing Fields of a Record in an Array	129
c)	Manipulating an Array of Records	129
VI.4.3)	Records in the C language	130
VI.4.3.1)	Declaration of a Structure	130
a)	Declaration of a structure model.....	130
b)	Declaration by Defining Type Synonyms	131
VI.4.3.2)	Manipulating a structure	132
a)	Accessing a Field of a Structure	132
b)	Reading and writing.....	132
c)	Assignment	133
VI.4.3.3)	Nesting of structures	134
VI.4.3.4)	Arrays of structures	134
a)	Declaration.....	134
b)	Manipulation.....	135
VI.5)	Other possibilities for type definition	135
VI.5.1)	Interval type.....	135
VI.5.1.1)	Declaration.....	136
VI.5.1.2)	Manipulation.....	136
VI.5.2)	Set type	136
VI.5.2.1)	Declaration.....	136
VI.5.2.2)	Manipulation.....	137
VI.5.2.3)	Set Operations.....	137
VI.6)	Exercises	138
VI.7)	Solution of the exercises.....	140
VI.8)	Conclusion.....	142
References	143

List of Figures

Figure I.1. Von Neumann model.....	14
Figure I.2. Example of Ancient Writing	15
Figure I.3. An Abacus.....	15
Figure I.4. El-Khawarizmi	15
Figure I.5: Antique Printing Press.....	15
Figure I.6. Pascaline	16
Figure I.7: Babbage's Calculating Machine.....	16
Figure I.8. George Boole.....	16
Figure I.9. Alan Turing.....	16
Figure I.10. Konrad Zuse.....	17
Figure I.11. ENIAC	17
Figure I.12. Von Neumann.....	17
Figure I.13. Transistor	17
Figure I.14. The TRADIC	18
Figure I.15. Integrated Circuit	18
Figure I.16. The Intel 4004 processeur	18
Figure I.17. The first microcomputer Micral N.....	18
Figure I.18. First PC (Personnal Computer).....	19
Figure II.1. Basic Structure of a Pseudo-Code Algorithm	29
Figure II.2. Example of a Representation of a Variable in Main Memory.	31
Figure II.3. Organigramme du calcul de la somme de deux nombres.....	43
Figure III.1. Flowchart of a simple conditional structure.	56
Figure III.2. Flowchart corresponding to the algorithm that tests if a number is negative	56
Figure III.3. Flowchart of a compound conditional structure.....	58
Figure III.4. Flowchart corresponding to the algorithm that tests the parity of a number.....	58
Figure III.5. Flowchart corresponding to nested structures.	60
Figure III.6. Flowchart of a multiple-choice structure.	63
Figure IV.1. Formalism of the WHILE loop in a flowchart	72

Figure IV.2. Flowchart for repeating age input using WHILE loop.....	73
Figure IV.3. Formalism of the REPEAT loop in a flowchart.....	75
Figure IV.4. Flowchart for Repeating Age Input Using the REPEAT Loop.....	75
Figure IV.5. Formalism of the FOR loop in a flowchart.....	79
Figure IV.6. Flowchart for Displaying Integers from 1 to 5 Using a FOR Loop.....	80
Figure V.1. Diagram of an array with 5 elements.....	88
Figure V.2. Diagram of the logical array A.....	90
Figure V.3. Example of an array T	91
Figure V.4. Diagrams of the 4 mark arrays.....	95
Figure V.5. Example of an array of arrays grouping the marks of all students in all courses.....	95
Figure V.6. Diagram of a 3-dimensional array representing all marks of all students in all courses...	96
Figure V.7. A Matrix with 4 Rows and 5 Columns	97
Figure V.8. The 8-bit ASCII table.	103
Figure V.9. Representation of the string "Box" in algorithmics.....	105
Figure V.10. Representation in the C language of the string "Guelma" in an array of size 10.....	108
Figure VI.1. Diagram of a record of type Date	125

List of Tables

Table II.1. Symbols used in a flowchart.....	42
Table II.2. Basic Data Types in the C Language.	45
Table II.3. Input and Output Formats in the C Language.....	48
Table IV.1. Differences between the WHILE and REPEAT loops.....	77
Table IV.2. Example of step-by-step execution of an algorithm.	82

Chapter I. Introduction

I.1) Introduction

Today, the realm of computer science permeates every facet of our daily lives. This widespread presence is attributed to the rapid advancement of computer technology, driven by breakthroughs across various technical disciplines. It's also a testament to the pivotal role played by algorithmics and, subsequently, programming, as the catalysts behind this evolution.

In this initial chapter, we endeavor to acquaint new students with the field of computer science. We accomplish this by elucidating essential definitions and offering a historical panorama of the progression of computer science throughout the years. Furthermore, we delve into the foundational principles of algorithmics through the lens of practical, real-world examples.

I.2) Computer Science: Definitions and a Brief History

The origins of computer science can be traced back to humanity's inherent desire for efficiency, seeking ways to enhance their calculations to minimize errors and save time. Nowadays, computer science is omnipresent in every facet of our daily lives. Thus, having even a basic understanding of this field is indispensable.

I.2.1) Definitions

I.2.1.1) *What is Computer Science?*

The status of computer science as a discipline is ambiguous and often misunderstood. Is it to be sought on the side of **science** or on the side of **technology**? What is the specific focus of study for computer scientists, and what are their true competencies?

In fact, the appellation "**computer science**" was first introduced in the mid-20th century, reflecting a pivotal moment in the digital revolution. The term gained prominence as the field rapidly evolved and expanded, fundamentally reshaping our relationship with information and technology. It has since become the cornerstone of a discipline that spans the realms of science and technology.

Below, we present several definitions of computer science from authoritative sources:

- As per the Oxford English Dictionary and the Cambridge English Dictionary, Computer Science is defined as "*the study of computers and how they can be used*".
- The world's leading digital dictionary, available at <https://www.dictionary.com> defines the computer science as "*the science that deals with the theory and methods of processing information in digital computers, the design of computer hardware and software, and the applications of computers*".

- Merriam-Webster's Collegiate Dictionary, gives the following definition of computer science: “*a branch of science that deals with the theory of computation or the design of computers*”.
- According to Encyclopedia Britannica, computer science is: “*the study of computers and computing, including their theoretical and algorithmic foundations, hardware and software, and their uses for processing information*”.

In other countries, specific terms have been proposed to describe the field of computer science. For instance, in French, the term "**informatique**" is widely used. In Germany, it is referred to as "**Informatik**," while in Spanish, it is known as "**Informática**", In Italian, it is termed "**Informatica**". However, since we are in a country that could be considered as Francophone, where French is the most common foreign language, we are particularly interested in the French translation.

Etymologically, the French word "informatique" is a neologism (a newly coined term) formed by blending the words "information" and "automatique". It was first introduced by Philippe Dreyfus, who, in 1962, used it for the first time to name his company, "Société d'Informatique Appliquée." Subsequently, the French Academy officially adopted this term in 1967, making "informatique" the standard reference. Today, "informatique" is a word widely embraced in French-speaking countries.

Here, we offer several definitions of the word "informatique" from original language sources.

- One of the most comprehensive definitions of the French word "Informatique", is provided by the *French Academy*: « *The science of the rational processing, particularly by automatic machines, of information regarded as a knowledge and communication medium in the technical, economic and social fields* ».
- The Larousse dictionary provides two definitions of the word "Informatique":
 - *The science of the automatic and rational processing of information, regarded as the medium of knowledge and communication.*
 - *The set of applications of this science, using hardware (computers) and software.*
- According to the "Le Petit Robert" dictionary, informatique is defined as the "*Theory and processing of information using programs implemented on computers*".

The previous definitions of computer science contain three key concepts that require further clarification: processing, information, and computer.

1.2.1.2) What is Information?

In fact, it's not possible to discuss the term *information* independently of the term *data*. However, these two concepts are sometimes conflated, and some interpretations consider information as data in every sense of the word.

On the contrary, the majority of definitions make a clear distinction between these two concepts.

a) Data

Data is raw information that has not yet been interpreted or placed in context.

According to Wikipedia: « *Data is a basic description of reality. It can be, for example, an observation or a measurement* ». Data can be collected by a monitoring tool, a person, and may take the form of numerical values, personal names, images, and more.

b) Information

The Oxford English Dictionary and the Cambridge English Dictionary define information as « facts or details about someone (person) or something (company, product, etc.) ».

In simpler terms, information is an interpreted form of data. In other words, adding context to data creates added value, transforming it into information.

Examples:

- When we say "24," it's data; "24 is the code for the Wilaya of Guelma " is information.
- "5000" is data, but if we add "meters" to it, making it "5000 meters", it becomes information.
- "110011" is a random sequence, but when treated as a PIN code, it transforms into information.

1.2.1.3) Information Processing

Information processing involves gathering data and subjecting it to a series of operations to obtain a result. When these operations are carried out by a machine, it is referred to as *Automatic Information Processing*.

1.2.2) Computer

Computer is an electronic programmable machine used for information processing following sequences of instructions (programs). In this process, information, whether it's textual, graphic, image, or sound data, is represented and encoded as sequences of binary digits, i.e., 0 and 1.

In France, the term “**ordinateur**” is used to refer to a computer, and it's not merely a direct translation from English. This designation has an interesting historical origin. In April 1955, IBM France approached Jacques Perret, a professor of Latin philology at the Sorbonne, with a unique request. They asked him to propose a word that would accurately capture the essence of what was then commonly referred to as a “*calculateur*”, a literal translation of the English word 'computer.' The term “*calculateur*” was deemed too limiting, given the expansive capabilities of these machines. In response, Jacques Perret introduced the term “*ordinateur*”, a word that better encapsulated the multifaceted nature of these machines.

The main advantage of a computer lies in its ability to **quickly** and **accurately** manipulate a large amount of information, **store** numeric or alphabetic data, and search, compare, or organize stored information.

1.2.2.1) Components of a Computer

A computer consists of two complementary parts: the immaterial programs (software) that describe the tasks to be performed and the physical machines (hardware) that execute these tasks.

a) Hardware

Computer hardware constitutes a collection of physical components essential for the automatic processing of information. At the heart of this machinery is the **microprocessor**, functioning as the central processing unit (CPU). This compact chip is akin to the brain of the computer, executing arithmetic and logic operations. Complementing the microprocessor is the **Central Memory Unit**, commonly known as **RAM** (Random Access Memory). RAM

serves as a high-speed, volatile memory that facilitates rapid data access for the processor, temporarily storing actively used program instructions and data to enhance overall system performance.

Another integral component is the **hard drive**, a storage device responsible for the long-term retention of data. It stores the operating system, software applications, and user files, ensuring their persistence even when the computer is powered off.

Furthermore, expanding on the interaction between users and machines, **input and output devices** play a pivotal role. Input devices are peripherals that enable users to provide data and instructions to the computer. Common examples include keyboards, which allow the entry of text and commands, mice and pointing devices facilitating cursor movement, and scanners converting physical documents or images into digital form. Microphones capture audio input, enabling voice commands or audio recording. Input devices collectively empower users to interact with the computer, initiating processes and conveying information to the machine. Conversely, output devices deliver processed information from the computer to users in human-readable forms. Monitors or displays present visual information, including text, images, videos, and graphics. Printers produce hard copies of documents and images. Speakers or headphones output audio information, encompassing system sounds, music, or other audio content. Output devices ensure that the results of computations and processed data are comprehensible and accessible to users.

b) Software

Software consists of a collection of programs and data that collaborate to provide services to the user.

At its core, a program is composed of a structured set of instructions that describe a task to be carried out by computer hardware. These instructions, crucial for the functioning of the computer, are encoded in binary, the sole language comprehensible by the machine.

However, there are two overarching categories: **system applications** and **user applications**.

System applications, often referred to as **system software**, form the foundational layer responsible for managing and controlling computer hardware. These applications provide vital services to the computer system, ensuring seamless operation and facilitating communication between hardware components and other software entities. Examples of system applications include operating systems (such as Windows, macOS, or Linux), device drivers, and utility programs. System applications serve as the backbone of the computing environment, handling tasks like memory management, process scheduling, and hardware interaction.

In contrast, user applications, also known as **application software**, are tailored programs designed to meet specific user needs and preferences. Unlike system applications, user applications are task-oriented and serve as the interface through which individuals interact with computers to accomplish various activities. This category encompasses a diverse range of software, including word processors, web browsers, graphic design tools, video games, and more. User applications leverage the underlying system software to execute functions personalized to the user's demands, offering a dynamic and user-centric computing experience.

I.2.2.2) Basic Computer Architecture

The hardware architecture of a computer is based on the Von Neumann model, which was introduced in 1946. In this model of a universal machine, the instructions of a program as well as the data it requires or generates are stored in its memory. This model typically distinguishes four distinct components, as illustrated in Figure I.1:

- Central Processing Unit (CPU): It consists of an Arithmetic Logic Unit (ALU), and a Control Unit:
 - The ALU : responsible for performing basic arithmetic and logic operations
 - The Control Unit: in charge of sequencing operations, acting as a coordinator.
- Memory, which stores both data and the program that instructs the Control Unit on the calculations to perform on this data.
- Input/Output Devices, enabling communication with the external world.

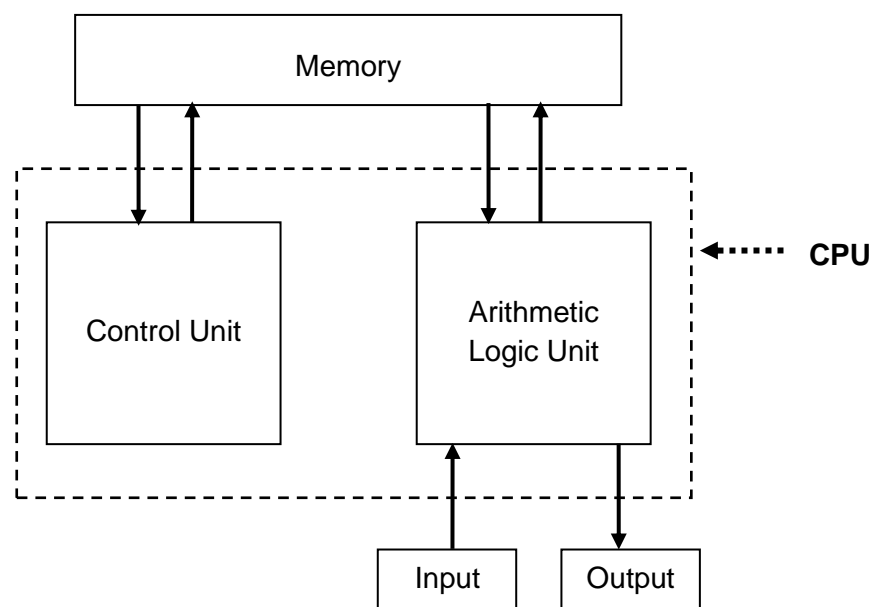


Figure I.1. Von Neumann model

I.2.3) Utility of Computing

Today, computing is omnipresent in all aspects of everyday life. This is primarily due to the increasing power of computers used for information processing. It offers several advantages across various application domains. Computing allows, among other things, for:

- Computing saves a significant amount of time, as it can complete tasks in seconds that used to take hours.
- Computing is reliable and error-free.
- Computing stores documents in a compact format, reducing paper consumption.
- Computing enables instant communication, such as real-time stock market quotes.
- Computing precisely and tirelessly manages machine tools that once required a large and costly skilled workforce.
- Computing doesn't take vacations and is never on sick leave.

I.2.4) A Brief History of Computer science

Computer science didn't emerge recently; its earliest fundamental concepts date back over 10,000 years, but the development of this science has only taken place in the past half-century.

The history of computer science is the result of the conjunction of scientific discoveries from various technical and social disciplines. Advancements in automation, electronics, and calculation techniques have played a pivotal role in the birth and rapid evolution of computer science.

In the following part of this section, we will attempt to provide an overview of the most significant scientific events in the history of computing.

By 3500 BC, writing emerged as another means of information storage and memorization.



Figure I.2. Example of Ancient Writing

Approximately 1000 years BC, the invention of the Abacus. An abacus consists of small beads moving on rods, serving as an early calculating device that enables numerical representation (memory) and the execution of operations on those numbers (calculation).

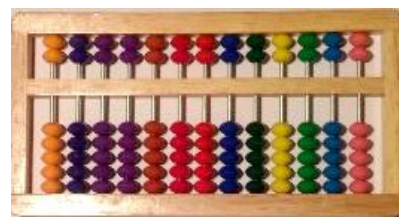


Figure I.3. An Abacus

In 830 AD, the renowned Arab mathematician *Al-Khwarizmi* introduced the concept of an algorithm, a method for solving equations.



Figure I.4. El-Khwarizmi

In 1454, the invention of the printing press by *Gutenberg* in Germany.



Figure I.5: Antique Printing Press

In 1642, *Blaise Pascal* created a machine (called the **Pascaline**) capable of performing additions and subtractions. It was designed to assist his father, who worked as a tax collector.



Figure I.6. Pascaline

In 1834, the British mathematician *Charles Babbage* designed his analytical engine, a computing machine controlled by a program stored on punched cards. It can be regarded as the precursor to modern computers.

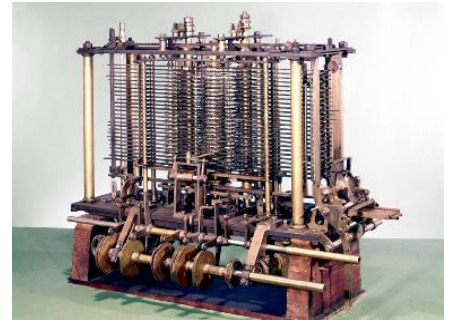


Figure I.7: Babbage's Calculating Machine

In 1854, the British mathematician and logician *George Boole* introduced binary algebra. He explained that the thought process can be encoded using three operations: AND, OR, NOT. His work would prove highly valuable for the advancement of electronics and logical gates.

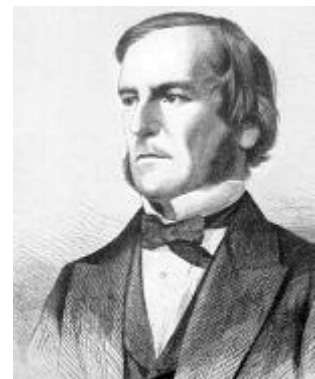


Figure I.8. George Boole

In 1936, the British mathematician *Alan Turing* published an article outlining the principles of the abstract machine that bears his name, the *Turing machine*. This machine is theoretically capable of following an algorithm. To achieve this, a computer's control unit must be able to direct the execution of the program (stored in memory) without the need for human intervention in the program's execution.



Figure I.9. Alan Turing

In 1941, the German engineer *Konrad Zuse* invented a computer that operated using electromechanical relays: the **Z3**. This computer was the first programmable machine to use binary instead of decimal.



Figure I.10. Konrad Zuse

Between 1944 and 1946, the invention of the **ENIAC** (*Electronic Numerical Integrator and Computer*) by *Eckert* and *Mauchly*, considered the first all-electronic computer (no longer containing mechanical parts). It weighed 30 tons, occupied a space of 1500 square meters, and was composed of 18,000 vacuum tubes, with one tube failing every 7 or 8 minutes. Its main drawback was its programming: the ENIAC could only be programmed manually using switches or plugboards. Programs were read directly by the processing unit from the punched tape or cards inserted at the machine's input. Sequence changes in the program couldn't be made without human intervention.

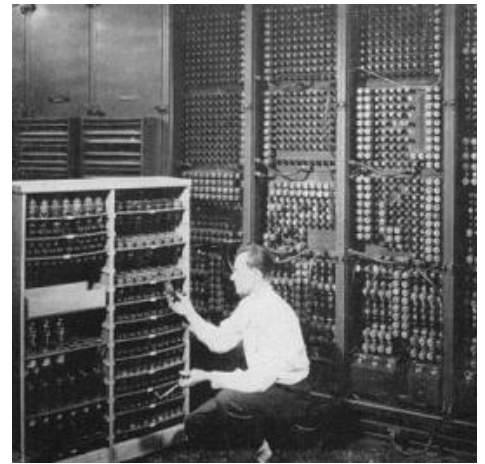


Figure I.11. ENIAC

In 1946, *Von Neumann* introduced the concept of stored program and presented the architecture of the modern programmable and memory-based computer.



Figure I.12. Von Neumann

In 1948, the **transistor** was invented by *John Bardeen*, *Walter Brattain*, and *William Shockley*. In the 1950s, it revolutionized the history of computers by making them less bulky, more energy-efficient, and therefore more cost-effective.



Figure I.13. Transistor

In 1956, the TRADIC (TRAnsistor DIgital Computer) became the first machine to use only transistors and diodes, without any vacuum tubes. It was constructed by Bell Labs.



Figure I.14. The TRADIC

In 1958, the integrated circuit was developed by *Texas Instruments*. It further reduced the size and cost of computers by integrating multiple transistors on a single electronic circuit without the need for electrical wiring.

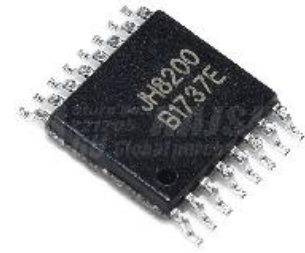


Figure I.15. Integrated Circuit

In 1971, the first microprocessor, the **Intel 4004**, made its debut. This microprocessor integrated logical, arithmetic, and other operations.

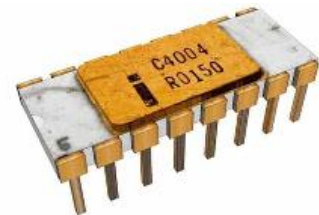


Figure I.16. The Intel 4004 processeur

In 1973, *François Gernelle* invented the **Micral N**, the first microcomputer. The Micral N exhibited all the characteristics of future personal computers of the 1980s.



Figure I.17. The first microcomputer Micral N

In 1981, IBM released the first Personal Computer (PC). This microcomputer featured a proper operating system (Microsoft's MS-DOS) and an « open » architecture, which allowed for the addition of numerous peripherals and the use of various software applications.



Figure I.18. First PC (Personal Computer)

In the 1990s:

- Tim Berners-Lee wrote the first proposal for creating the World Wide Web.
- Windows was released.
- Linux was created.
- Google had its beginnings.
- And many more significant technological developments occurred during this era.

I.3) Introduction to Algorithms

Indeed, the concept of an algorithm is quite general and not limited to the field of computer science. Throughout history, humans have sought to devise sufficiently precise procedures to solve their problems and organize their activities. With the advent of computing, this notion has become more restrictive because an additional objective is to transform an algorithm into a program.

I.3.1) Introductory Examples

There's nothing better than a few introductory examples to introduce the concept of algorithms.

I.3.1.1) Example 1: Fax Machine User Manual

An excerpt from a fax machine user manual on sending a document.

1. Insert the document to be sent into the automatic feeder.
2. Type the recipient's fax number digit by digit from the numeric keypad.
3. Press the send button to initiate the transmission.

This user manual provides instructions on how to send a fax. It consists of an ordered sequence of commands or instructions (insert, type, press) that manipulate data (document, automatic feeder, fax number, numeric keypad, send button) to perform the desired task (sending a document)."

I.3.1.2) Example 2: Calculating the Average of 3 Numbers with a Calculator

Let's say we want to calculate the average of 3 numbers using a calculator. The steps to follow are as follows:

- | | |
|-----------------------------|-------------------------------|
| 1) Press "On" | 7) Press "=" |
| 2) Enter the first number. | 8) Press "/" |
| 3) Press "+" | 9) Press "3" |
| 4) Enter the second number. | 10) Press "=" |
| 5) Press "+" | 11) The average is displayed. |
| 6) Enter the third number. | |

The preceding steps represent the commands (Press, Enter, etc.) that need to be executed. These commands or instructions manipulate data (the 3 numbers, calculator keys) to achieve the desired result (the average of the 3 numbers).

1.3.1.3) Example 3: Giving Directions

A first-year Mathematics student wants to find his way to Amphitheatre No. 4 for the first time to attend an algorithmics class. He asks his colleague, who is already present in the Amphitheatre, to show him the way. The colleague's response is as follows:

« Enter through the main gate of the central campus. Then, go straight until you reach the next intersection. Do not turn right or left; continue straight until the end of the curve, then turn left. Keep going straight, and you will see Amphitheatre 4 right in front of you ».

In this dialogue, the colleague's response represents an ordered sequence of instructions (go straight, turn left, etc.) that manipulate data (intersection, streets) to accomplish the desired task (getting to Amphitheatre 4).

In summary, the steps to follow in each of the three previous examples make up what we call an **algorithm**. It's simple, isn't it?

In this way, we encounter algorithms in our daily lives, and we execute them. Moreover, we unknowingly create algorithms (as in the case of providing directions to a lost student).

1.3.2) Notion of Algorithm

Before delving into the necessary definitions, it's important to note that algorithmics is a term of Arabic origin. It originates from the renowned Muslim mathematician *Muhammad ibn Mūsā al-Khwārizmī* (830 AD), who is one of the pioneers of modern arithmetic.

1.3.2.1) What is an Algorithm?

Here are some important definitions of an algorithm:

- In the Oxford English Dictionary, the algorithm is described as: *« a set of rules that must be followed when solving a particular problem ».*
- according to the Cambridge English Dictionary, an algorithm is: *« a set of mathematical instructions or rules that must be followed in a fixed order, and that, especially if given to a computer, will help to calculate an answer to a mathematical problem ».*
- The world's leading digital dictionary characterizes it as: *« a set of rules for solving a problem in a finite number of steps, such as the Euclidean algorithm for finding the greatest common divisor ».*

- Merriam-Webster's Collegiate Dictionary defines it as: « *a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation* ».
- According to Encyclopedia Britannica, an algorithm is: « *a set of steps that are followed in order to solve a mathematical problem or to complete a computer process*».
- Collins Dictionaries state that an: « *An algorithm is a series of mathematical steps, especially in a computer program, which will give you the answer to a particular kind of problem or question* ».
- The *Universalis Encyclopedia*¹ defines an algorithm as « *An algorithm is a finite sequence of rules to be applied in a determined order to a finite set of data in order to arrive, in a finite number of steps, at a certain result, regardless of the data* ».

It's noteworthy that all of these definitions make reference to mathematics, underscoring the inherent relationship between algorithmics and mathematics.

Other examples of algorithms:

The steps to make coffee, instructions for using a device, starting a vehicle, integer division, and solving a second-degree equation are algorithms.

Remarks:

- 1) If the algorithm is correct, the result is as desired; otherwise, the result will be random.
- 2) To work, an algorithm must contain only instructions that are understandable to the one who will execute it. For example, you cannot give the instruction "solve the problem" and leave the interlocutor to figure it out. It's logical because if the lost student asks their colleague for directions to Amphitheater 4, it's because they don't know it. So, telling them to "figure it out on your own" is not helpful. Therefore, the actions that make up the algorithm should be **primitive** or elementary.
- 3) Proficiency in algorithmics requires two qualities:
 - You need to have a certain intuition.
 - You need to be methodical and rigorous.
- 4) The value of an algorithm lies in its ability to be translated into computer programs that can be executed directly by the computer.

1.3.2.2) Object of Algorithmics

According to Merriam-Webster's Collegiate Dictionary, « *algorithmics is the subdiscipline of informatics or computer science concerned with the study, analysis, and development of algorithms* »

The object of algorithmics is the design, evaluation, and optimization of calculation methods in mathematics and computer science.

Algorithmics focuses on the art of constructing algorithms and characterizing their validity, robustness, reusability, complexity, and efficiency.

1.3.2.3) Qualities of an Algorithm

As discussed in the previous section, an algorithm is essentially the blueprint of a program. Therefore, for a program to be of high quality, the initial algorithm must be correct and of

¹ *Universalis* is the reference encyclopedia in the French-speaking world

good quality as well. To ensure that an algorithm is of good quality, it should possess the following qualities:

- **Readability:** The algorithm should be understandable, even by non-computer scientists.
- **Validity:** It should perform the exact task for which it was designed.
- **Finiteness:** The algorithm must terminate within a finite amount of time.
- **High-Level:** An algorithm should be translatable into any programming language, without relying on technical notions specific to a particular program or operating system.
- **Precision and Non-Ambiguity:** Each element of the algorithm should be unambiguous, leaving no room for confusion.
- **Conciseness:** An algorithm should not exceed one page; if it does, the problem should be decomposed into smaller sub-problems.
- **Structured:** It should be composed of distinct and easily identifiable parts.
- **Robustness:** An algorithm should be capable of protecting itself from abnormal usage conditions.
- **Reusability:** It should be reusable for solving tasks similar to its original purpose.
- **Efficiency:** The algorithm should make optimal use of the hardware resources on which it runs.

I.3.3) Problem-Solving Steps

When using a computer to solve a given problem (by creating a computer program for automated processing), you must go through a series of steps known as problem-solving steps. It's important to note that the most crucial steps in the problem-solving process are independent of the computer and are carried out without it.

Therefore, the steps for solving a problem in computer science are as follows:

I.3.3.1) Problem Definition and Analysis

The purpose of this step is to understand the problem at hand and identify the desired objective by analyzing the problem statement. If the problem is complex, it can be broken down into a set of smaller, less complex sub-problems. Ultimately, solving the initial problem is equivalent to solving all the resulting smaller problems.

In this step, you should:

- Determine the outputs (desired results), their format, and their type.
- Identify the inputs (data) required to obtain these results, including their type, characteristics, and the order in which they need to be input.
- Determine any intermediate variables (if applicable), which are temporary data used to store intermediate calculation results.
- Enumerate various potential computer-based solution methods and evaluate them to select the best one in terms of ease, speed, and required memory.

It's essential to anticipate responses to all foreseeable cases during this step.

Example:

Analyze and outline the steps for calculating the average of 3 integers:

- **Inputs:** the 3 numbers (a , b , and c , integer numbers)
- **Outputs:** the average (moy , real number)
- **The plan:**
 - Enter the values of the 3 numbers a , b , and c
 - Calculate the average moy using the formula $moy = (a + b + c) / 3$
 - Display the result moy .

1.3.3.2) Establishment of the Corresponding Algorithm

After selecting the best method to solve the problem and determining all the calculation formulas to be applied, the next step is to express this method in the form of logical, successive steps that lead to a solution to the problem. These successive steps are known as an « *algorithm* »

An algorithm is often described using a highly simplified, natural pseudo-language that is both readable and formal. The data, results, and intermediate variables are clearly declared, each calculation is fully specified, and so on.

1.3.3.3) Translating the Algorithm into a Computer Program

In this step, the solution steps described in the algorithm are expressed using instructions in a specific high-level programming language, such as Pascal, C, Java, etc. This process results in the creation of a source code program.

The source code program is the outcome of translating the algorithm while adhering to the syntax and rules of the chosen programming language.

1.3.3.4) Compiling the Program

Once the computer program (source code) is written, it needs to be introduced to the computer to ensure it is correctly written and to translate it into machine language (0s and 1s), which is the only language the computer understands. This translation process is called « *compilation* » and is performed by a specialized program called a « *compiler* ».

The resulting program is referred to as an « *executable program* ».

During this translation, the compiler detects any potential errors, both lexical and/or syntactical.

1.3.3.5) Program Execution and Testing

This step involves ensuring that the program produces correct results in all cases and for all possibilities.

Multiple tests need to be conducted, corresponding to different scenarios, to verify the validity of the results.

If all results are valid, the program is accepted as it is. Otherwise, the algorithm may need to be revised and modified.

I.4) Exercises

Exercise I.1 :

To convert degrees Fahrenheit to Kelvins, the following formula is used:

$$K = \frac{F + 459,67}{1,8}$$

Where K is the degree in Kelvin, and F is the degree in Fahrenheit.

Analyze and provide the steps for this conversion.

Exercise I.2 :

Consider the following problem: A merchant wants to determine the profit he can make from selling a given product.

As analysts, you have been asked to analyze and list the various steps that lead to solving this problem, assuming that all entered data is valid.

Recall that profit is equal to the selling price minus the cost price, and the cost price is equal to the sum of the purchase price and expenses.

Exercise I.3 :

Analyze and provide the steps for calculating the volume of a sphere with radius R . The calculation formula is as follows:

$$V = \frac{4\pi}{3} R^3$$

Where V is the volume of the sphere.

Exercise I.4

Given an angle θ , we want to determine the type of this angle. An angle can be one of the following:

- Zero ($\theta = 0^\circ$)
- Acute ($0^\circ < \theta < 90^\circ$)
- Right ($\theta = 90^\circ$)
- Obtuse ($90^\circ < \theta < 180^\circ$)
- Flat ($\theta = 180^\circ$)

Analyze and provide the steps to solve this problem.

Exercise I.5 :

Analyze and provide the steps to solve a second-degree equation.

I.5) Solution of the exercises

Exercise I.1 :

Inputs: Temperature in Fahrenheit (F , real number)

Outputs: Temperature in Kelvins (K , real number)

Plan:

- Enter the temperature in Fahrenheit (F)
- Calculate the temperature in Kelvins using the formula: $K = (F + 459.67) / 1.8$
- Display the temperature in Kelvins (K)

Exercise I.2 :

Inputs: Selling price (SP, real number)
Purchase price (PP, real number)
Expenses (E, real number)

Outputs: Profit amount (PA, real number)

Intermediate data: Cost price (CP, real number)

Plan:

- Enter the selling price (SP), the purchase price (PP), and expenses (E)
- Calculate the cost price using the formula: $CP = PP + E$
- Calculate the profit amount using the formula: $PA = SP - CP$
- Display the profit amount (PA)

Exercise I.3 :

Inputs: Sphere radius (R, real number)

Outputs: Sphere volume (V, real number)

Plan:

- Enter the radius (R)
- Based on the value of R:
 - If $R > 0$:
 - Calculate the volume using the formula: $V = ((4 * \pi) / 3) * R^3$
 - Display the volume (V)
 - If $R \leq 0$:
 - Display the message "Input error"

Exercise I.4:

Inputs: Angle (θ , real number)

Outputs: Type de angle (T, String)

Plan:

- Enter the angle (θ)
- Based on the value of θ :
 - If $\theta = 0$, T = "Zero"
 - If $0 < \theta < 90$, T = "Right"
 - If $\theta = 90$, T = "Droit"
 - If $90 < \theta < 180$, T = "Obtuse"
 - If $\theta = 180$, T = "Straight"
- Display the type of angle (T)

Exercise I.5 :

Inputs: The 3 coefficients (a, b, c, real numbers)

Outputs: The equation's roots (x1, x2, real numbers)

Intermediate data: The discriminant (Δ , real number)

Plan:

- Input the 3 coefficients (a, b, c)
- Based on the value of a:
 - If $a = 0$:
 - Calculate the unique solution x1 using the formula: $x1 = -b/a$
 - Display the unique solution x1
 - If $a \neq 0$:
 - Calculate Δ using the formula: $\Delta = b^2 - 4 * a * c$
 - Based on the value of Δ :
 - If $\Delta < 0$: Display "No solution"
 - If $\Delta = 0$:
 - Calculate the double solution using the formula: $x1 = -b / (2 * a)$
 - Display the double solution x1
 - If $\Delta > 0$:
 - Calculate x1 using the formula $x1 = (-b - \sqrt{\Delta})/(2 * a)$
 - Calculate x2 using the formula $x2 = (-b + \sqrt{\Delta})/(2 * a)$
 - Display both solutions x1 and x2

I.6) Conclusion

This first chapter serves as an introduction to the course. In this chapter, we have attempted to introduce the basic concepts of computer science as a discipline that is ubiquitous in all areas of everyday life, and algorithmics, which constitutes the heart of computer science.

The chapter is divided into two parts. The first part focuses on the field of computer science in general. In this part, we addressed the necessary definitions, provided an overview of the historical evolution of computer science over the years, and presented the computer while describing its basic architecture and components.

The second part delves into algorithmics. It began with the introduction of the concept of an algorithm through concrete examples. It then presented definitions of this concept, its objectives, and qualities. The chapter ended by listing the various steps in problem-solving in computer science, with the establishment of the algorithm being one of the key steps.

Chapter II. Simple sequential algorithm

II.1) Introduction

In the previous chapter, we have seen that solving a computer science problem is not arbitrary but requires extensive preparation, from problem analysis to writing a computer program. The resulting program contains instructions for the computer to execute without any capacity for invention. However, writing the program is preceded by the creation of an algorithm describing the steps to solve the problem. The algorithm itself must be written following specific writing rules defined in what is called an **algorithmic language** or a **pseudocode**.

In this chapter, we will introduce the algorithmic language that will be used throughout this course for describing and creating algorithms. We will describe the general structure of an algorithm and present its basic elements (instructions, data objects, etc.). Additionally, we will explore how to sequence basic instructions to design our first algorithm for solving a specific problem. We will also present flowcharts, a graphical representation of the algorithm, along with the rules for constructing them from an algorithm. Towards the end of the chapter, we will initiate students into the fundamental vocabulary of the C language, enabling them to embark on the journey of programming.

II.2) Concept of Language and Algorithmic Language

In general, language refers to the ability that humans have to express their thoughts and communicate with each other. It is a way of expressing oneself, of communicating, specific to a group.

However, with the advent of computers and information technology, this term has been extended to refer to systems that enable communication with machines. These languages are then called artificial languages.

Subsequently, I will present some definitions related to the term "language".

II.2.1) Language

A language is a structured system of signs (vocal, gestural, graphical, etc.) that allows humans (or other entities) to express thoughts, ideas, information, emotions, and communicate with each other.

A language is composed of:

- **Vocabulary:** a list of symbols (words).
- **Grammar:** rules that define how symbols can be combined.
- **Semantics:** the meaning of symbols.

II.2.2) Computer Language

A programming language, or computer language, is a conventional notation designed to express algorithms and create computer programs to execute them.

Similar to a natural language, a programming language consists of an alphabet, a vocabulary (list of keywords), syntax and grammar rules, and meanings (semantics). These rules specify how keywords can be assembled to create instructions, forming programs that can run smoothly on a machine.

Programming languages serve as an interface between humans and computers, allowing the writing of operations that computers can execute while remaining comprehensible to humans. Since these languages are intended for computers, they must adhere to strict syntax.

II.2.3) Machine Language

Machine language is the language understood by the microprocessor. It comprises extremely basic instructions encoded with precision in the form of binary bit sequences.

To write a program in machine language, one must have a deep understanding of the processor's operation that will be used and be knowledgeable about the binary code for each instruction, making it a challenging skill to acquire at present.

II.2.4) Algorithmic Language

Although an algorithm, which is a description of the steps to solve a problem, can be expressed in natural language through a series of unrestricted sentences, this method or formalism of writing algorithms is not the most commonly used in computer science. Natural languages are inherently ambiguous. Recall that an algorithm must be precise and unambiguous (as discussed in Chapter 1), so it's essential to write algorithms in a formal language with precisely defined semantics to avoid any ambiguity. This specialized language is referred to as **algorithmic language**, **pseudo-language**, or **pseudocode**.

II.2.4.1) What is an Algorithmic Language?

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

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

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

Remark:

It's important not to confuse the algorithm, which is the description of the problem-solving process in a series of steps, with its implementation in a specific computer-interpretable language: the program.

II.2.4.2) Elements of Algorithmic Language

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

a) Keywords

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

The number of keywords in the algorithmic language is limited, and we will introduce them as we progress in this course.

Remarque:

Le langage algorithmique n'est pas standard et il pourra que vous trouverez une notation déférente si vous consultez d'autres documents. Par exemple, la lecture est symbolisé dans notre cours par le mot-clé **LIRE** alors que dans d'autres document elle peut être notée par **SAISIR**.

The algorithmic language is not standard, and you may find different notations when consulting other documents. For instance, in our course, reading is symbolized by the keyword **READ**, whereas in other documents, it might be denoted as **INPUT**.

II.2.4.3) Difference between algorithm and program

Algorithmics expresses the instructions for solving a given problem independently of the specific details of any particular language. Therefore, a program translates the algorithm into a specific language while respecting its syntax. Learning algorithmics is learning to handle the logical structure of a computer program.

To draw an analogy, if a program were a house, algorithm would be the blueprint. It's better to create the blueprint first and then build the house rather than the other way around...

Remark:

The algorithmic language is not standardized, and you may find different notations if you consult other documents. For example, in our course, reading input is symbolized by the keyword **READ**, whereas in other documents, it might be represented as **INPUT**.

II.3) Parts of an algorithm

An algorithm in pseudocode consists of three essential parts: the algorithm header, the declaration section, and the algorithm body (processing section). The following figure illustrates the basic structure of an algorithm in pseudocode:

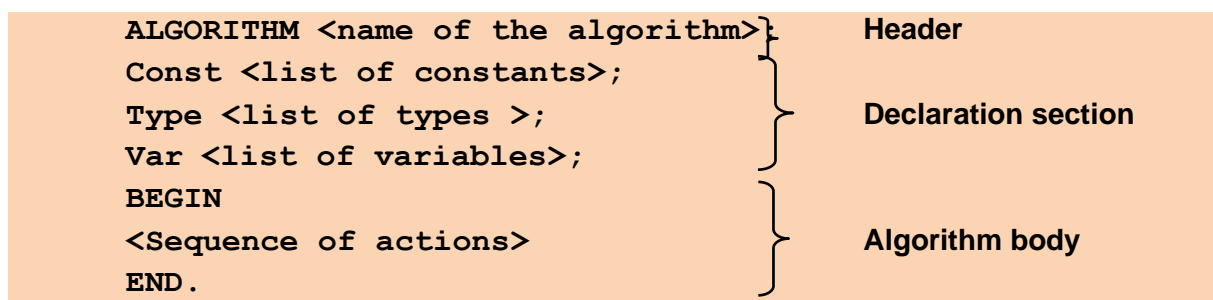


Figure II.1. Basic Structure of a Pseudo-Code Algorithm

II.3.1) Algorithm Header

The algorithm header has the sole purpose of identifying the algorithm by specifying a name for it. The header begins with the keyword **ALGORITHM** followed by the name of the algorithm. However, the name assigned has no influence on the execution and results of the algorithm. Note that the algorithm's name must adhere to certain constraints, which we will discuss shortly.

Examples:

```
ALGORITHM Sum;  
ALGORITHM calculation;
```

II.3.2) The Declaration Section

The declaration section includes declarations for all objects or data elements used in the processing section of the algorithm. Declaration involves naming various objects, specifying their types, dimensions, and so on.

All objects in the processing section must have been declared in the declaration section.

Toutes les instructions doivent se terminer par un point virgule ";" qui serve comme séparateur entre les instructions.

II.3.3) Algorithm Body

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

In this section, you will find basic instructions (assignment, input, output) and control statements that combine basic instructions to perform other tasks.

The processing section begins with the keyword **BEGIN** and ends with the keyword **END**, which indicates the end of the algorithm.

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

II.4) Data: Variables and Constants

In an algorithm, you will frequently need to temporarily store objects on which the entire algorithm's processing relies. These objects can come from the hard drive or be provided by the user (keyboard input). They can also be results obtained by the algorithm, whether intermediate or final.

Whenever you need to store information during an algorithm, you use a variable or a constant. These can come in various forms: textual, numerical, logical, and so on.

Data must be declared in every algorithm before their use.

II.4.1) Variables

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

From the computer's perspective, a variable is simply a memory location at an arbitrary address identified by a name², and able to store data of a predefined type. The name is used to locate this memory location so that the computer can access it directly.

A variable can be schematically represented as a box labeled with a name, with a size (defined by the type), content (value), and a memory address.

Example:

Suppose we have a variable named x ; it can be schematically represented in memory as follows:

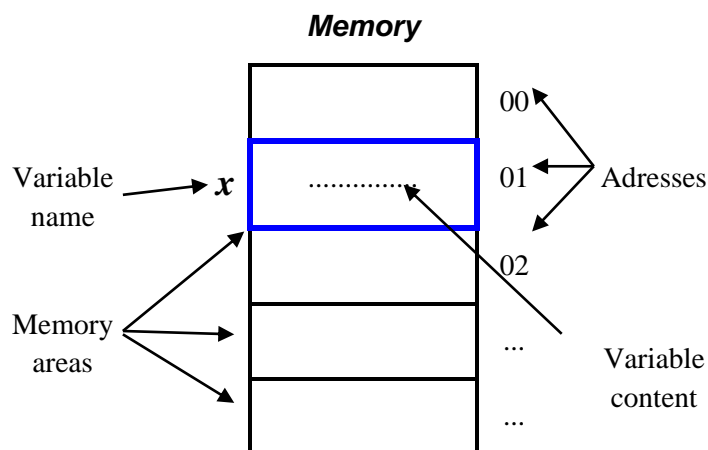


Figure II.2. Example of a Representation of a Variable in Main Memory.

From the previous definitions, we conclude that the concept of a variable in algorithmics is different from that in mathematics, although the name is the same.

To use a variable, we must first define (declare) it in the declaration part of the algorithm to reserve a memory space for it.

II.4.2) Constants

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

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

Remark:

Constants make it easier to write and maintain the algorithm. Suppose we use the value 5.271 thirty times in the algorithm. Without the use of constants, on the one hand, we have to write this value every time we need it, which is thirty times. On the other hand, when we want to change this value in the algorithm to 5.273, for example, we would be required to go through the entire algorithm and change all occurrences of the old value to the new one, which is very tedious and can lead to errors due to oversight.

² Similar to the algorithm name, the variable name must meet specific criteria, which we will discuss shortly.

It is therefore preferable to declare the value 5.271 as a constant, named ν for example, and use ν throughout the algorithm instead of 5.271. In this case, when we want to change the value 5.271, we only need to modify it once: in the declaration of the constant ν .

II.4.3) Notion of Identifier

Cependant, malgré que le choix est libre du nom de la variable ou la constante, il préférable, pour des raisons de lisibilité et de compréhension de choisir des identificateurs significatifs, c'est à dire en fonctions de ce ils représentent.

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

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

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

Remarque:

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

Examples :

`A, DELTA, X1, VAL, i, MM, B_727, RACINE` are valid identifiers.
`END, 12MOT, VAL*2` are invalid identifiers.

II.5) Data Types

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

Types serve various purposes, including:

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

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

- Elementary types
- Structured or composite types

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

II.5.1) Elementary types

Ce sont de types simples, c'est à dire qu'une variable de ces types contient une seule valeur à la fois. Dans les types élémentaires on distingue les types standards des types non standards.

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

II.5.1.1) Standard types

En algorithmique, il y a cinq types standards, dits aussi types prédéfinis : entier, réel, caractère, chaîne de caractères, et logique.

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

a) Integer

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

b) Real

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

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

Remark:

A variable of **real** type can have a value of **integer** type because the **integer** type is included in the **real** type, but the reverse is not possible.

c) Character

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

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

Examples:

```
'R', '5', '*', ...
```

Remark:

Characters are ordered according to the order of the considered machine codes. Although there are numerous encodings, the most common one is ASCII (American Standard Code for Information Interchange).

d) String

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

Strings of characters are delimited by double quotes (quotation marks).

Examples:

"Algo", "123", "True" are strings.

Remark:

The `string` type can also be viewed as a composite type. It will be studied in more detail in Chapter 5.

e) Boolean (Logical)

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

f) Note or Types

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

Examples :

- The value 3 (`Integer` type)
- The value 3.0 (`Real` type)
- The value '3' (`Character` type)
- The value "3" (`String` type)

II.5.1.2) Non-standard Types (ou non-prédéfinis)

In the non-predefined types, we mainly distinguish between enumerated types, interval types, and set types.

These types will be studied in Chapter 6.

II.5.2) Declaration of variables and constantes

Following our exploration of the data types used in algorithms, let's now delve into the process of declaring variables and constants.

II.5.2.1) Declaration of variables

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

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

```
VAR <name_Variable> : <type_Variable>;
```

Where: `<name_Variable>` is the identifier designating the declared variable, and `<type_Variable>` is the variable's type. The latter can be any elementary or structured data type.

Example:

```
VAR age: Integer;
VAR avg: Real;
VAR name: String;
VAR admitted: Boolean;
VAR famSit: Character;
```

Remarks :

- Il est possible de placer plusieurs déclarations de variables dans la même ligne séparées par des points virgules.
- The declaration of a variable involves reserving memory space corresponding to the type of the declared variable.
- A single **VAR** keyword is sufficient to declare multiple variables, even of different types. The scope of this keyword extends until the end of the declaration section of the algorithm and the beginning of the body.
- If you need to declare multiple variables of the same type, there's no need to create a separate line for each variable. You can declare them all at once on the same line, separating them with commas.
- It is also possible to place multiple variable declarations on the same line, separated by semicolons.

Example:

```
VAR  a : Intger ;
VAR  b : Intger ;
VAR  c : Intger ;
VAR  avg : Real ;
VAR  admitted : Boolean ;
```

These declarations can be refined as follows:

```
VAR  a,b,c : Integer; avg : real; admitted : Boolean;
```

II.5.2.2) Declaration of constants

La déclaration d'une constante commence par le mot clé **CONST** suivi par le nom de la constante suivi par le caractère *égale* " = " suivi par la valeur de la constante. La syntaxe de déclaration est le suivant :

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

```
CONST <name_Constant> = <value_Constant> ;
```

Here: **<name_Constant>** is the identifier designating the declared constant, and **<Value_Constant>** is the value of the constant.

Examples :

```
CONST pi = 3.14 ;
CONST Number = 10 ;
CONST Point = '.' ;
```

Remarks:

- A single keyword **CONST** is sufficient to declare multiple constants.
- Multiple constant declarations can be placed on the same line separated by semicolons.
- Character constants must be enclosed in single quotes " ' ".
- Constant declarations come before variable declarations.

Example:

The previous example is equivalent to this one:

```
CONST pi = 3.14; Number = 10; Point = '.';
```

II.6) Basic Operations

The body of the algorithm encompasses a series of instructions designed to manipulate the data declared in the algorithm's declaration section. These instructions combine the data using various operators to create expressions. Understanding the fundamentals of operators, operands, and expressions is crucial to effectively utilize them in algorithm construction.

We will first address the concepts of operators, operands, and expressions.

II.6.1) Operator and Operand

As previously explained, the type declaration serves to specify the set of operations that can be applied to variables of that type.

An operator is a symbol that denotes an operation, which either acts on variables or performs calculations.

An operand is an entity, such as a variable, constant, or expression, utilized by an operator.

There are various categories of operators. Operators that operate on two operands are called *binary operators*, while those that operate on a single operand are called *unary operators*.

II.6.1.1) Arithmetic Operators

These are the usual arithmetic operations.

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

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

It should be noted that the operators mentioned earlier are binary operators except for the "-" operator, which can also be unary and indicates a change in sign.

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

II.6.1.2) Logical Operators

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

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

II.6.1.3) Comparison Operators

Comparison operators are used to compare two values of the same type and return a boolean result (true or false) based on order relations: natural order for integers and real

numbers, and ASCII lexicographic order for characters and strings. These operators include "<", ">", "=", "≠", "≤", "≥".

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

II.6.2) Expression

An expression is a combination of operands linked by operators, and it evaluates to a single value. The operands can be direct values, constants, variables, or other expressions.

Every expression is associated with a type, which is the type of the resulting value of that expression. However, based on the types of operators and operands within the expression, various types of expressions can be distinguished, including arithmetic expressions, logical expressions, character type expressions, and so on.

Examples:

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

- $12.5 * a + (b/2)$ is an arithmetic expression,
- $a > b \text{ et } b \geq c$ is a logical expression.

II.6.2.1) Validity of an expression

To ensure the validity of an expression and determine its type, it is essential to check the syntax of the expression, the compatibility of the operand types it consists of, and the validity of the operators involved. The type of the expression is determined by the types of its operands. Consequently, it is necessary for the operand types of an operator to be compatible. For instance, adding an **integer** and a **character** doesn't make sense. A particular case is the one involving integers and real numbers. These can appear together in the same expression, and the type of the resulting expression is real.

II.6.2.2) Evaluation of an expression

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

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

Remarks :

- For operators of the same priority, the expression is evaluated from left to right.
- Logical values are also ordered such that **Faux < Vrai**.
- If there are parentheses, start by evaluating the innermost ones.
- To avoid any ambiguity, it is advisable to always use parentheses.

Example:

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

Evaluate and determine the type of the following expressions :

- $12 * 3 + 5$ is correct and has the value of 41 (**Integer** value)
- $12 * (3 + 5.1)$ is correct and has the value of 41 97.2 (**Real** type)
- $-x - k \text{ div } 3$ is correct and is of **Real** type
- $z \text{ mod } j + i$ is incorrect because **MOD** is not valid for **Real** type
- $\text{NON } y \text{ ET } B > 0$ is incorrect : **NOT** should be applied to **Boolean**
- $c='c' \text{ OU } c='t'$ is incorrect : **OR** is not valid for **Character** type
- $(c='c') \text{ OU } (c='t')$ is correct and is of **Boolean** type

II.7) Basic instructions

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

In algorithmics, there are three basic instructions: assignment, and input/output instructions.

II.7.1) Assignment instruction

Assignment is an operation that allows assigning (attributing) a value to a variable. It is denoted by the symbol « \leftarrow ». The syntax of this instruction is as follows:

<Variable> \leftarrow <Value> ;

It is read as: **<Variable>** receives **<Value>** or **<Variable>** gets **<Value>**.

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

Examples:

```
A  $\leftarrow$  3 ;      assign the direct value 3 to variable A.
B  $\leftarrow$  A ;      assign the content of variable A, which is 3, to variable B.
B  $\leftarrow$  B - 2 ;  evaluate the expression B-2 and put the result (equal to 1) into
                  variable B.
```

Remarks:

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

Exercise:

What will be the values of the variables used after the execution of the following instructions?

```

ALGORITHM Test1 ;

VAR A,B: Integer; R:Real;

BEGIN

  A ← 4 ;

  B ← A + 2 ;

  R ← B / A ;

  A ← A div 3 ;

  B ← 3 ;

  R ← (R * 2) / B ;

END.
    
```

Solution

Instruction	A	B	R
1	4	/	/
2	4	6	/
3	4	6	1.5
4	1	6	1.5
5	1	3	1.5
6	1	3	1

II.7.2) Input/Output instructions

Let's imagine that we've created an algorithm to calculate the double of a number, let's say 5. If we've kept it very simple, we might have written something like this:

```

ALGORITHM double ;
VAR A,B :integer ;
BEGIN
  A ← 5 ;
  B ← A * 2 ;
END.
    
```

On one hand, this algorithm gives us the double of 5. That's fine, but if we want the double of a number other than 5, we would need to rewrite the algorithm.

On the other hand, the result is calculated by the machine, but it keeps the result to itself. The user executing this algorithm will never know what the double of 5 is.

Thankfully, there are instructions that allow the machine to interact with the user.

One of these instructions enables the user to input values from the keyboard for the algorithm to use. This operation is called **reading**.

Another instruction allows the algorithm to communicate values to the user by displaying them on the screen. This operation is called **writing**.

II.7.2.1) Reading (input) instruction

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

```

READ (<variable>) ;
    
```

This instruction involves storing the value entered by the user via the keyboard into the memory location allocated for **<variable>**.

During execution, the machine waits for the user to input the value for **<Variable>** using the keyboard. Subsequently, the user must press the Enter key "**↵**" to signal the completion of the input operation.

Example :

```
Var x,y:Integer; a:Real;
READ(x) ;
READ(y) ;
READ(a) ;
```

Remarks :

- The value entered via the keyboard must be compatible with the receiving variable.
- Multiple read instructions can be combined within a single statement by separating the variables to be read with commas. Thus:

```
READ(v1) ; READ(v2) ; ... ; READ(vn) ;
```

Is equivalent to:

```
READ(v1, v2, ..., vn) ;
```

Example :

The reading instructions in the previous example can be grouped into a single statement as follows:

```
READ(x, y, a) ;
```

II.7.2.2) Writing (output) instruction

Writing is an instruction that allows displaying a value on the screen. This value can be a direct value, constant, the content of a variable, a message, the result of an expression, ...

The syntax of this instruction is as follows:

```
WRITE(Value) ;
```

Example :

WRITE("The mark: ");	displays the message "The mark: ".
WRITE(mark);	displays the content of the variable mark.
WRITE(6*2+5) ;	evaluates the expression 6*2+5 and displays its result.

Remarks :

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

Example :

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

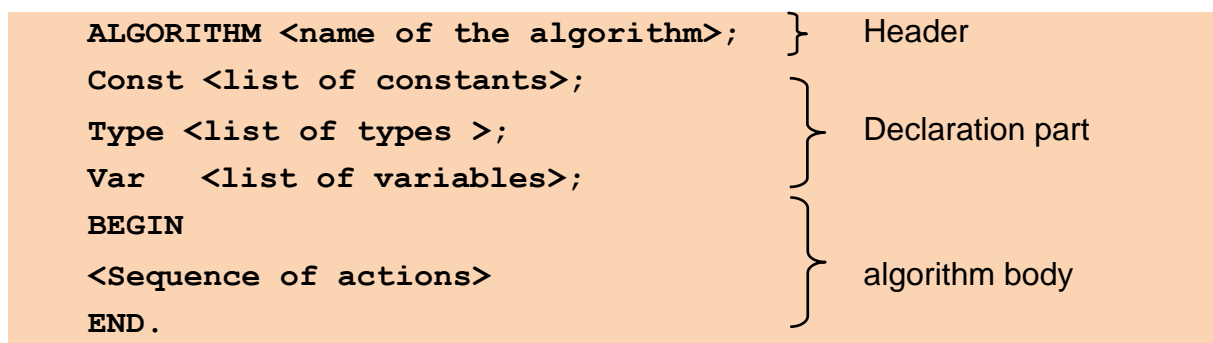
This instruction displays the message "**The mark:**" followed by the content of the variable **mark**, followed by the string **"/20."**

II.8) Building a simple algorithm

To build an algorithm, you need to combine all the concepts presented above. As a result, the algorithm takes the following form:

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

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



Example:

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

First version:

```

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

Second version:

```

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

II.9) Representation of an Algorithm Using a Flowchart

Les opérations dans un organigramme sont représentées par les symboles dont les formes sont normalisées. Ces symboles sont reliés entre eux par des lignes fléchées qui indiquent le chemin. Les principaux symboles sont les suivants:

We have already seen that an algorithm can be formalized either in natural language or in pseudocode. However, there is a third way to represent algorithms: the flowchart. The flowchart is a graphical representation of a problem's solution, which makes it easy to understand but, on the other hand, it tends to take up a lot of space.

Operations in a flowchart are represented by symbols with standardized shapes. These symbols are connected by arrows, indicating the flow of the algorithm. The main symbols are summarized in the following table:




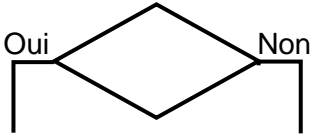

Symbol	Role
	Used to mark the beginning and end of a flowchart.
	Used to mark read and write operations.
	It is used for assignment operations (actions).
	Used to represent tests or conditional branching.
	Symbol of connection between various symbols. It also indicates the sequencing of operations.

Table II.1. Symbols used in a flowchart.

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

Remark:

Flowcharts do not contain declarations.

Example:

The previous algorithm for calculating the sum of two numbers can be represented by the flowchart illustrated in Figure II.3.

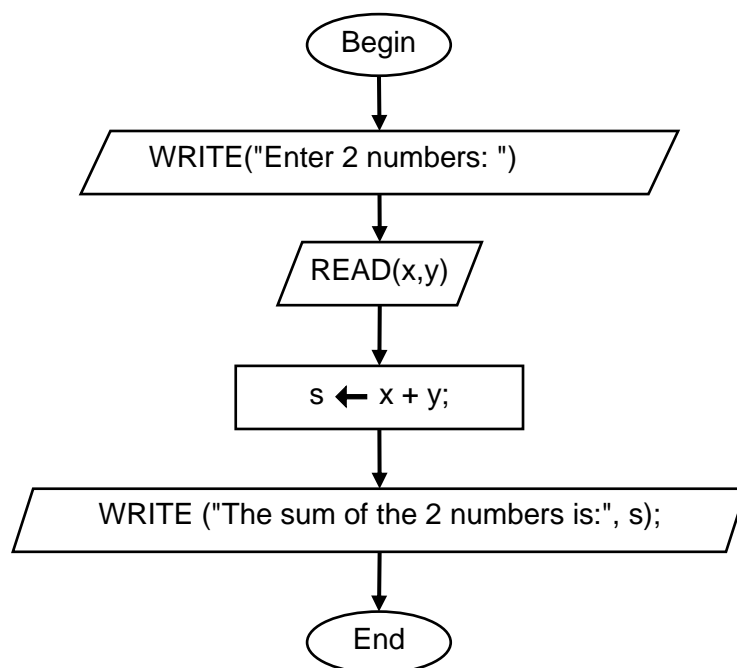


Figure II.3. Organigramme du calcul de la somme de deux nombres

II.10) Translation into C language

We have mentioned that the next step in the problem-solving process, after establishing the algorithm, is to translate it into a computer program by expressing the solution steps described in the algorithm using a programming language.

In fact, a large number of programming languages have been proposed, and this number continues to grow. Each of these languages has its own characteristics, advantages, and limitations. Therefore, the choice of a language to use depends on various factors, such as the type of the intended application, hardware and software architecture, portability, security, and more. In the context of this course, we have opted for one of the most widely used languages: the C programming language.

II.10.1) The C language: Presentation

The C language is an imperative and general-purpose programming language invented in 1972 by Dennis Ritchie and Ken Thompson, researchers at Bell Labs, with the goal of developing the famous UNIX operating system. Ritchie and Thompson drew inspiration from languages like B and BCPL (*Basic Combined Programming Language*) to create the new C language.

C has become one of the most widely used programming languages today. Many more modern languages like C++, Java, C#, and PHP borrow aspects from C. While C was initially created to develop UNIX, it is still widely used for system programming. Therefore, the kernels of major operating systems like Windows and Linux are largely developed in the C language.

II.10.2) Why the C language ?

The C language is one of the most widely used programming languages today. Its main advantages are as follows:

- C is a **general-purpose** (universal) language. It is not oriented towards a specific application domain.
- It is a compact language. It consists of a core set of operators and predefined functions with a simple and efficient formulation. This makes learning the C language less challenging than many other recent languages.
- It is a « **close-to-the-machine** » language. It provides operators that are very close to those of the machine language and functions that allow direct access to the computer's internal functions, especially for memory management. This enables the development of efficient and fast programs.
- It is a **portable** language. This means that a program written in C, following the ANSI-C standard, can run without modification on any operating system after recompilation.
- It is an **extensible** language. In addition to standard functions, many function libraries have been developed.
- It is a **modular** language. A program can consist of multiple modules or « source files », which promotes code structuring, comprehensibility, and code reusability.

II.10.3) Basic Elements of the C language

In this section, we will briefly present the basic elements of the C language.

II.10.3.1) Structure of a C program

The simplest structure of a C program is as follows:

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

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

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

It is important to pay attention to the following elements:

- **main** is a predefined name of the main function that must exist in a C language program. It should be in lowercase.
- The parentheses after the **main** function are mandatory.

- The curly braces ({ and }) mark the beginning and end of a block of instructions or a function. They replace **BEGIN** and **END** in algorithmic notation.

II.10.3.2) Declaration section

a) Identifiers

The identifiers in the C language have the same characteristics as in algorithmics. Furthermore, C has the feature of being case-sensitive, meaning it distinguishes between uppercase and lowercase letters.

Example:

The identifiers: `NAME`, `Name`, and `name` are 3 different identifiers in C language

b) Predefined types in the C language

The basic data types in the C language include characters, integers, and floating-point numbers (real numbers). In C, there are multiple integer and real number types, depending on the number of bytes they are encoded on and their format, i.e., whether they are signed (having a sign - or +) or not. By default, data is signed.

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

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

Table II.2. Basic Data Types in the C Language.

Remarks :

- The amount of memory space occupied by different types depends on the machine where the compiler is implemented. In the previous table, we considered the case of a machine with a 32-bit microprocessor.
- The C language does not have **Boolean** and **String** data types.
- In C, there is no distinction between the character itself (e.g., 'A') and its ASCII code. Therefore, the `char` type can be represented as an `integer` encoded on 1 byte.

c) Declaration of variables

The declaration of a variable begins with the variable's type, followed by an identifier indicating the variable's name, optionally followed by the "=" character and the initial value of the variable. The declaration must be terminated by a semicolon. Therefore, the declaration of a variable can be done in two possible ways:

```
<type_Variable> <name_Variable>;
<type_Variable> <name_Variable>=<initial_Value>;
```

With: <type_variable> being the data type of the variable (one of the types in the following table), <name_Variable> is the variable's name, and <initial_Value> is the initial value you want to assign to the variable.

Examples:

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

d) Declaration of constants

The declaration of constants in the C language can be done in two ways: using the `const` keyword or using the `#define` directive. We limit ourselves here to declaration using `#define` as this method is safer from a data protection standpoint.

Therefore, the declaration begins with the `#define` keyword, followed by a space, the name of the constant, another space, and finally, the value of the constant. It takes the following form:

```
#define <name_constant> <value_Constant>
```

Remarks:

- Each constant must be declared on a separate line, and these lines do not end with a semicolon.
- Character constants should be enclosed in 2 apostrophes.
- String constants should be enclosed in double quotation marks.

Examples:

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

II.10.3.3) Processing Section**a) Operators**

The operators in the C language are as follows:

a.1) Arithmetic Operators

The classic arithmetic operators include the unary operator - (sign change) as well as the binary operators:

```
+ : addition      - : subtraction      % : remainder of the division
* : multiplication / : division (integer and real)
```

Remark:

C only uses the "/" notation for both integer division and floating-point division. If both operands are of integer type, the "/" operator will produce integer division (the quotient of the division). However, it will deliver a floating-point value as soon as one of the operands is a float. So, for example, if `a=9/6;` will return `1` because both operands are integers. In contrast, `a=9.0/6;` will return `1.5` because one of the operands is a real number (`9.0`).

a.2) Logical Operators

They allow the creation of logical or boolean expressions, thus enabling the formation of complex expressions from simple ones.

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

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

a.3) Relational Operators

These operators are used to perform tests between the values of two expressions, and they are primarily utilized in comparison expressions.

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

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

b) Assignment statement

In the C language, assignment is symbolized by the "=" sign. Its syntax is as follows:

```
<name_variable> = <value> ;
```

The right-hand side `<value>` can be a direct value, a constant, another variable, or an expression.

Examples:

```
A = 3 ;           //Assign the direct value 3 to the variable A
B = A ;           //Assign the content of the variable A to the variable B
B = B - 2 ;       //Store the result of the expression B - 2 in the variable B
c = 'K' ;         //Assign to the variable c the character 'K'
```

Remark:

In C, assignment is not just an instruction like in other languages; it's a full-fledged operator. As a result, an expression like `a=b=c=1` is entirely valid, allowing for multiple assignments. You can even perform "chained assignments" and build expressions like `a=(b=10)+2;` which assigns the value 10 to `b` and 12 to `a`.

c) Reading (input)

The reading is done in the C language using the `scanf` function³ from the `stdio.h` library.

The `scanf` function allows to input data from the keyboard and store it at the addresses specified by the function's parameters. The syntax of this function is as follows:

```
scanf("<control string>", &<variable1>, &<variable2>, ...)
```

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

The `<control string>` specifies the format in which the input data is to be converted. Thus, for each variable, a format specifier is specified. Format specifiers are indicated by a character preceded by the "%" sign. The format code and the variable type must match.

The input formats for the `scanf` function are summarized in the following table:

Format	Data type	Data representation
%d	int	Signed decimal
%hd	short int	Signed decimal
%ld	long int	Signed decimal
%u	unsigned int	Unsigned decimal
%hu	unsigned short int	Unsigned decimal
%lu	unsigned long int	Unsigned decimal
%f	float	Floating-point, fixed decimal
%lf	double	Floating-point, fixed decimal
%Lf	long double	Floating-point, fixed decimal
%c	char	Character
%s		String of characters

Table II.3. Input and Output Formats in the C Language.

Example :

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

Remarks :

- Data entered via the keyboard should be separated by spaces or "Enter" unless they are characters.
- It is possible to specify the number of digits or characters to be read. For instance, "%3d" for an integer spanning 3 digits, including the sign.

³ Let's now consider the function as an instruction that performs a specific task. In the second semester, we will study functions and their characteristics in detail.

d) Writing (output)

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

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

```
printf("<control string>",<expression1>,<expression2>,...) ;
```

The `control string` contains the text to be displayed and the format specifiers corresponding to each expression in the parameter list. Format specifications serve the purpose of specifying the format of data to be displayed. They are introduced by the "%" character, followed by a character indicating the print format. The format specifiers are the same as those presented in the table II.3 above.

Example:

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

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

Remark :

We can specify certain parameters of the print format, such as the minimum width of the print field (e.g., "%4d" specifies that at least 4 characters will be reserved to print the integer), and the precision of the fractional part (e.g., "%.3f" means that a floating-point number will be printed with 3 digits after the decimal point). When the precision is not specified, it defaults to 6 digits after the decimal point.

II.10.3.4) Example of a C Program

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

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

II.11) Exercises**Exercise II.1 :**

Write an algorithm that allows entering the marks for the "Algorithms and Data Structures" course of a first-year Mathematics student, and calculates and displays his average, knowing that the latter is calculated using the following formula:

$$Avg = \frac{TW + PW}{2} \times 0.4 + Exam \times 0.6$$

Exercise II.2 :

Let A, B, and C be three logical variables. Write an algorithm that reads the values of these 3 variables from the keyboard and calculates and displays the value of the following expression:

$$R = (A + B). (\bar{A} + C). (B + \bar{C})$$

Exercise II.3 :

The torr (Torr) is a unit of pressure measurement. It is defined as the pressure exerted at 0°C by a column of 1 millimeter of mercury. It was later indexed to atmospheric pressure: 1 standard atmosphere corresponds to 760 torrs and is equal to 101325 Pascals.

Write an algorithm that reads a pressure in torrs and converts it to pascals.

Exercise II.4 :

Consider a regular pyramid with a square base.

Write an algorithm that input the height of the pyramid and the length of the side of its square base, and calculates and displays its surface area.

Recall that the area of a triangle is equal to half of the product of the length of the base of the triangle by its height.

Exercise II.5 :

A magician asks a spectator to think of a number and write it on a slate. He invites the spectator to hide this slate for the duration of the act. He then asks him to add 3 and multiply this sum by the number he initially thought of. He insists: do not forget this result. Then calculates the square of the initial number. Finally, he asks the spectator to subtract this result from the previous one. The magician requests the spectator to say the final result out loud.

Establish the pseudo-code algorithm corresponding to this statement.

II.12) Solution of the exercises**Exercise II.1 :**

```
Algorithm average ;
Var TW,PW,Exam,Avg:Real ;
Begin
Write("Enter the marks for Tw,PW,and exam:");
Read(TW,PW,Exam);
Avg ← ((TW+PW)/2)*0.4 + Exam*0.6;
Write("The average is: ",Avg,"/20");
End.
```

Exercise II.2 :

```

Algorithme expression_logique;
Var A,B,C,R:logique;
Début
Ecrire("Donner la valeur de A, B, et C: ");
Lire(A,B,C);
R ← (A ou B) et (non A ou C) et (B ou non C);
Ecrire("Le résultat est: ",R);
Fin;

```

Exercise II.3 :

1 atmosphere = 760 torr = 101325 Pa which implies that 760 torr = 101325 Pa

So 1 torr = 101325/760 = 133.32 Pa

```

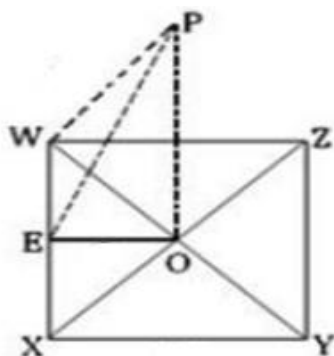
Algorithm conversion;
Var torr,pa:Real;
Begin
Write("Enter the pressure in torrs: ");
Read(torr);
pa ← torr * 101325/760;
Write("The pressure in pascals is: ",pa);
End.

```

Exercise II.4 :

It is worth noting that in order to calculate the area of the triangle, we need to know the height of the triangle (the slope's height). Do not confuse the slope's height with that of the pyramid.

Consider the pyramid illustrated in the following figure:



The height of a regular pyramid is the distance between its apex (Sommet) and the center of its base along an axis perpendicular to the base (PO in the figure above). In contrast, the height of the triangular face in the pyramid is the distance between the apex of the pyramid and the midpoint of one of the sides of the base (PE in the figure above).

The height of the triangular face can be calculated by applying the Pythagorean theorem:

$$EO^2 + OP^2 = PE^2 \Rightarrow PE = \sqrt{EO^2 + OP^2}$$

In this solution, it is assumed that we have a predefined function `sqrt` for calculating the square root of a number.

```

Algorithm pyramide ;
Var height,lenSide,areaPyr,heightTri,areaTri,areaBase:Real ;
Begin
Write("Enter the pyramid height:");
Read(height);
Write("Enter the lenght of its base side:");
Read(lenSide);
heightTri ← sqrt((lenSide/2)^2+ height^2);
areaTri ← (lenSide*heightTri)/2;
areaBase ← lenSide*lenSide;
areaPyr ← areaTri*4+areaBase;
Write("The pyramid's area is: ",areaPyr);
End.

```

Exercise II.5 :

```

Algorithm magician;
Var nb,r,carre:integer;
Début
Write("Think of a number and write it down:");
Read(nb);
r ← nb+3;
r ← r * nb;
carre ← nb * nb;
r ← r - carre;
Write("The final result is: ",r);
End.

```

II.13) Conclusion

In this chapter, we have presented the basic concepts necessary for building an algorithm in pseudocode. We have tried to cover all the concepts and explain them through examples. The chapter also illustrates how to graphically represent an algorithm to obtain a flowchart. Finally, the chapter introduces the C language and explains its basic elements to show how a C program is derived from an algorithm.

Chapter III. Conditional Structures

III.1) Introduction

In the previous chapter, we saw that an algorithm consists of two essential parts: the declaration part, which contains the definition of all data elements to be used in the algorithm, and the processing part, which gathers the set of instructions that manipulate the declared data. The instructions discussed in the previous chapter (assignment, input, and output) are elementary instructions, and the sequence of these instructions allows us to compose a **sequence (block)** that performs more or less complex operations. However, sequential instructions (instructions in a sequence) are executed one after the other, and an instruction can only be executed when all the preceding instructions have already been executed. As a result, each sequential instruction is executed once, and only once, in the order they were written.

Indeed, the problems we deal with often require the examination of various situations that cannot be handled by simple sequences of actions. In such cases, we need to choose between 2 or more processing depending on whether a certain condition is met or not. Since we have multiple situations, and we do not know in advance which case we will have to execute, in the algorithm, we must account for all possible cases. Conditional structures (tests) allow us to achieve this. There are several forms of conditional structures, and they are the subject of the current chapter.

III.2) Notion of Condition

In algorithms, a condition is merely a logical expression that can become a true or false statement depending on the values that make up the expression.

The condition can be simple or compound.

III.2.1) Simple Conditions

A simple condition involves comparing two expressions of the same type. It is composed of three elements: an expression, a comparison operator, and another expression. Comparison operators have already been introduced in Chapter 1.

The combination of these three elements forming the condition, if you will, is a statement that is either **TRUE** or **FALSE** at a given moment.

Examples :

```
val=5      a<b      x+3 ≥ 5*y-4      'c'≠ 'a'
```

Are simple conditions.

III.2.2) Compound Conditions

These are conditions formed by combining multiple simple conditions using logical **AND** and **OR** operators.

Examples:

```
nb ≥ 0 and nb ≤ 20      x=0 or y=0 c ≠ '0' and (c='N' or c='n')
```

Are compound conditions.

Remark:

Ternary comparisons are not allowed in algorithmics. They must be represented using composite conditions.

Example:

The condition $a > b > c$ is not a valid condition,
It must be decomposed into $a > b$ and $b > c$

III.3) Simple Conditional Structures (IF Statements)

Simple conditional structures allow the execution of a sequence of instructions only if a condition is met. The failure to meet the condition corresponds to no action to be taken.

III.3.1) Algorithmic Syntax

The syntax of a simple conditional structure is as follows:

```
IF <Condition> THEN <block of instructions>;
```

The **IF** statement in its simplest form is used to test the validity of a condition. If the condition is true, then the block of instructions following **THEN** is executed. If the condition is false, it is the instruction that follows the conditional structure in the algorithm that is executed.

Example:

Write an algorithm that asks for a number and determines if it is negative.

Solution :

```
ALGORITHM test;
VAR a : Integer;
BEGIN
Write("Enter a number : ");
Read(a);
IF a < 0 THEN Write("Negative number");
END.
```

In this example, we test if the value of the variable **a** is less than 0; if yes, we display the message "Negative number," otherwise, we do nothing.

Remarks:

- The condition can be simple or composite.
- There is no semicolon after the condition or after **THEN**.

- The block of instructions can contain one or more instructions. If the sequence contains multiple operations, they are separated by semicolons and enclosed between **BEGIN** and **END**. Otherwise, the compiler will consider only the first instruction as subordinate to the **IF** structure. If the sequence contains a single instruction, then the words **BEGIN** and **END** are not required.

Example:

Consider the following two algorithms:

```
ALGORITHM test1;
VAR x,y : Integer;
BEGIN
y ← 7;
Write("Enter a number : ");
Read(x);
If x > 10 Then
y ← x - 2;
x ← y;
END.
```

```
ALGORITHM test2;
VAR x,y : Integer;
BEGIN
y ← 7;
Write("Enter a number : ");
Read(x);
If x > 10 Then
    BEGIN
        y ← x - 2;
        x ← y;
    END.
END.
```

Let's assume that the entered value is 5. What will be the value contained in variable **x** at the end of the two algorithms?

Solution:

Algorithm 1: **x = 7** (the initial value of **y**)

Algorithm 2: **x = 5** (remains unchanged)

Since the entered value of **x** is 5, the condition **x < 10** is not satisfied in both algorithms.

In Algorithm 1, only the instruction **y ← x - 2** is subordinate to the **IF** structure, and therefore the instruction **x ← y** is executed. As a result, **x** takes on the initial value of **y**, which is 7.

In Algorithm 2, the two preceding instructions are part of the sequence that executes when the condition is satisfied. Since the condition is false, these instructions are not executed, and we proceed directly to the **END**. The value of **x** remains unchanged (**x = 5**).

III.3.2) Flowchart

The simple conditional structure can be represented in a flowchart as follows:

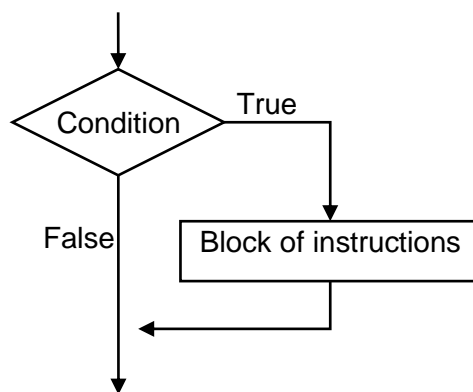


Figure III.1. Flowchart of a simple conditional structure.

Example:

The flowchart corresponding to the algorithm that checks if an entered number is negative is as follows:

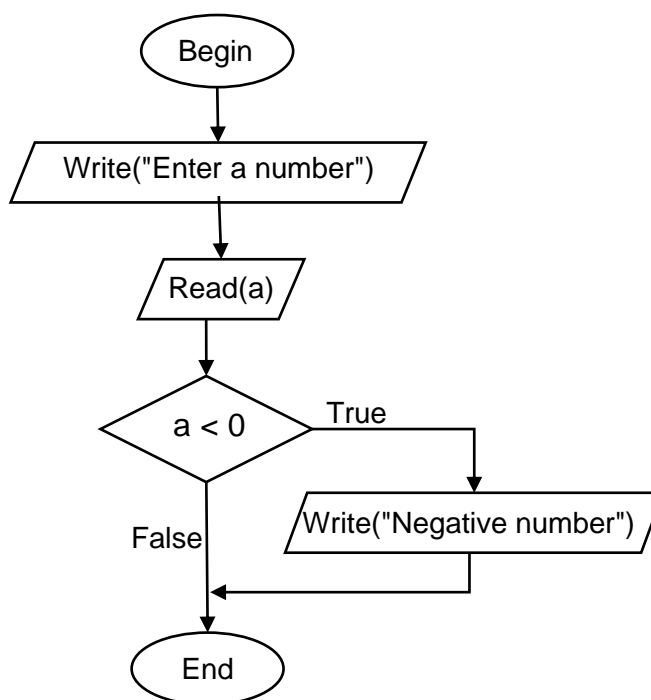


Figure III.2. Flowchart corresponding to the algorithm that tests if a number is negative

III.3.3) C language Syntax

The syntax of a simple conditional structure in the C language is as follows:

```
if(<condition>) <block of instructions>;
```

Remarks:

- The condition must be enclosed in parentheses.
- There is no semicolon after the condition.

- If the block of instructions consists of multiple instructions, it should be enclosed in braces (`{` and `}`), and if it contains a single instruction, braces are not mandatory.

Example:

The C program that determines if an entered number is negative is as follows:

```
main() {
    int a;
    printf("Enter a number : ");
    scanf("%d",&a);
    if(a<0)printf("Negative number");
}
```

III.4) Compound Conditional Structures (IF - ELSE statements)

The compound conditional structure (also called alternating conditional structure) allows to execute a sequence of instructions if a condition is satisfied and to execute another sequence if the condition is not satisfied.

III.4.1) Algorithmic Syntax

The general form of a compound conditional structure is as follows:

```
IF <Condition> THEN <block of instructions1>
ELSE <block of instructions2>;
```

In this form, the condition is evaluated. If it is true, then `<block of instruction1>` will be executed. If it is false, `<block of instruction2>` will be executed.

Remarks:

- The condition can be simple or composite.
- In algorithms, there is never a semicolon before **ELSE**.
- If a block of instructions consists of two or more instructions, it must be enclosed by the keywords **BEGIN** and **END**. If it contains only one instruction, **BEGIN** and **END** are not required.
- The two blocks of instructions cannot be executed simultaneously. It's logical because a condition cannot be both true and false at the same time. These two blocks are executed alternately, which is why they are called alternate structures.

Example:

Write an algorithm to input a number and determine whether it is even or odd.

Solution :

```
ALGORITHM test;
VAR nb : Integer;
BEGIN
Write("Enter a number : ");
Read(nb);
IF nb mod 2 = 0 THEN Write("The number is even")
ELSE Write("The number is odd");
End.
```

In this example, we are testing whether the remainder of dividing the variable `nb` by 2 is equal to 0. If it is, we display the message "Even number"; otherwise, we display another message: "Odd number."

III.4.2) Flowchart

The compound conditional structure can be represented in a flowchart as follows:

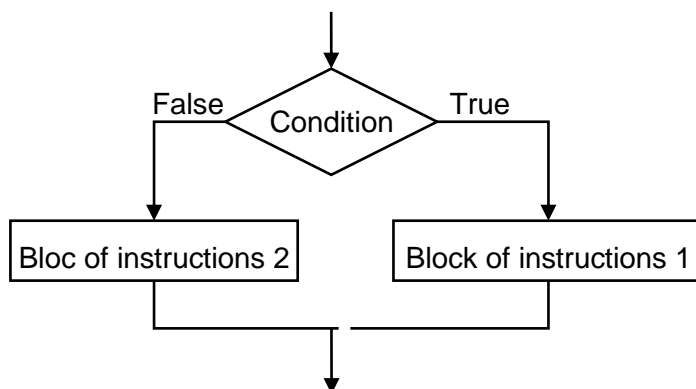


Figure III.3. Flowchart of a compound conditional structure

Example:

The flowchart corresponding to the algorithm that tests the parity of an integer entered by the user is as follows:

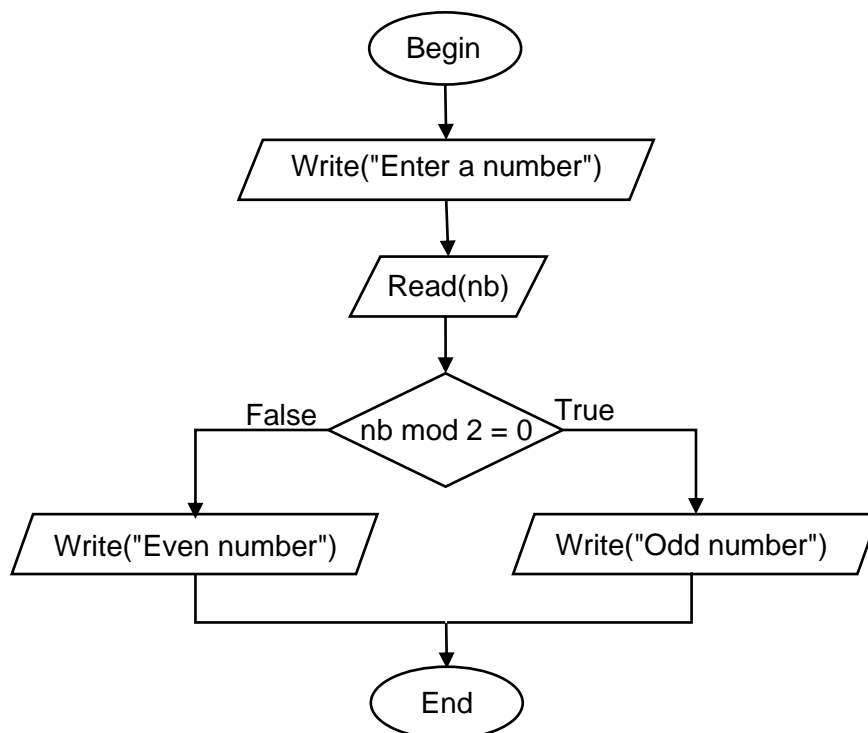


Figure III.4. Flowchart corresponding to the algorithm that tests the parity of a number

III.4.3) C language Syntax

The syntax of a compound conditional structure in the C language is typically written as:

```

if(condition) <block of instructions1>;
else <block of instructions2>;

```

Remarks:

- The condition must be enclosed in parentheses.
- In C language, **else** can be preceded by a semicolon.
- If <block of instructions1> (respectively <block of instructions2>) consists of multiple instructions, it should be enclosed in braces ({ and }). If it contains a single instruction, braces are not mandatory.

Example:

The C program that allows to input a number and determine whether the number is even or odd is as follows:

```

main(){
    int nb;
    printf("Enter a number : ");
    scanf("%d",&a);
    if(a % 2 == 0)printf("The number is even");
    else printf("The number is odd");
}

```

III.5) Nested conditional Structures

It's important to understand that the instruction blocks of **IF** and **ELSE** are sequences of instructions. These blocks can contain reading, writing, assignment instructions, as well as conditional structures. When this occurs, it's referred to as having nested structures.

III.5.1) Algorithmic Syntax

The general form of nested conditional structures is as follows:

```

IF <Condition1> THEN <block of instructions1>
ELSE IF <Condition2> THEN <block of instructions2>
    ELSE IF <Condition3> THEN <block of instructions3>
        ...
        ...
    ELSE <block of instructions_n>;

```

The execution of the nested structure described above proceeds as follows. <Condition1> is evaluated first. If it is satisfied, <block of instruction1> is executed. If not, <condition2> is evaluated. If the latter is true, we execute <block of instruction2>; otherwise, we move on to evaluating <condition3> and so forth.

Remarks:

- Note the absence of semicolons before all the **ELSE** statements in the nested structure.
- An **ELSE** always corresponds to the last **IF** statement encountered to which an **ELSE** has not yet been assigned.

- The evaluation of conditions occurs from top to bottom until one of them is satisfied. The associated block of instructions is then executed, and the processing of the nested structure is completed.

Example:

Write an algorithm that reads a real number and displays whether this number is positive, negative, or zero.

Solution:

```

ALGORITHM test;
VAR x : Real;
BEGIN
Write("Enter a number : ");
Read(x);
IF x > 0 THEN Write("The number is Positive ")
ELSE IF x < 0 THEN Write("The number is Negative")
      ELSE Write("The number is Zero");
END.

```

III.5.2) Flowchart

The flowchart corresponding to the nested conditional structures:

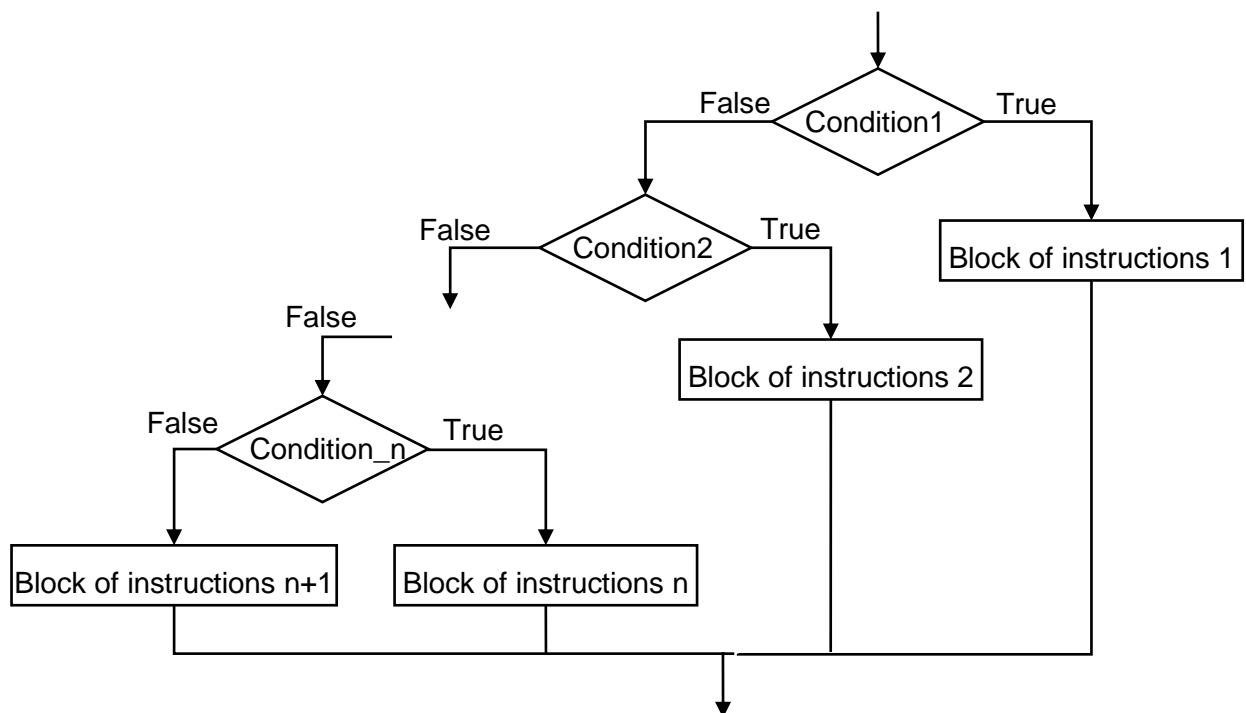


Figure III.5. Flowchart corresponding to nested structures.

III.5.3) C language Syntax

The nesting of multiple tests is done in the C language as follows:

```

if(condition1) <block of instructions1>;
else if(condition2) <block of instructions2>;
    else if(condition3) <block of instructions3>;
        .....
            else <block of instructions_n>;

```

Example:

The C program that allows you to enter a real number and determine if it is positive, negative, or zero is as follows:

```

main(){
    float x;
    printf("Enter a number : ");
    scanf("%f",&x);
    if(a>0)printf("The number is Positive");
    else if(a<0)printf("The number id negative");
    else printf("The number is Zero");
}

```

III.6) Multiple-choice structure (CASE statement)

As we have seen before, nesting conditional structures allows us to handle multiple cases by evaluating various conditions. However, nesting a large number of tests tends to make the algorithm heavier and more challenging to read and manage.

Fortunately, we have a structure that makes the task a bit easier, known as the *multiple-choice statement* or the *selective statement*. This structure allows to select or distinguish several cases (not just two cases like the alternate conditional structure) based on the values of an expression. This expression is called the **selector**, and it must be a scalar type variable or expression (**integer**, **character**, or **boolean**). However, the selective statement compares the value of the selector to a list of values and executes a sequence of instructions among several options based on the actual value of the selector.

III.6.1) Algorithmic Syntax

The syntax of the multiple-choice statement is as follows:

```

CASE <expression> OF
    <Value1> : <block of instructions 1>
    <Value2> : <block of instructions 2>
    .....
    <Value_n> : <block of instructions n>
    OTHERWISE: <block of instructions (n+1)>;
END;

```

The **CASE** statement begins by evaluating and testing the validity of the **<expression>**. The expression's value (if it's valid) is then successively compared to each of the selection values. As soon as there is a match, the comparisons are stopped, and the associated block is executed. If no value matches, then the block associated with **OTHERWISE**, if it exists, is executed.

Remarks:

- In the multiple-choice statement, the order of presentation does not matter.
- The selector (the selection expression) and the values to choose from must be of the same type.
- If a block of instructions consists of more than one instruction, they must be enclosed between **BEGIN** and **END**.
- The default case (**OTHERWISE**) is optional. It is used to perform a task when none of the cases is true.
- The **CASE** statement can be replaced by nesting multiple **IF - ELSE** statements, but the reverse is only possible if the tests concern a single variable and the tests for that variable are equality tests.

Example:

Write an algorithm that allows entering an integer between 1 and 5 and displays it in words.

```
ALGORITHM example;
TYPE Digit = 1..5;
VAR n : Digit;
END
Write("Enter a number between 1 and 5 : ");
Read(n);
CASE n OF
  1: Write("One");
  2: Write("Two");
  3: Write("Three");
  4: Write("Four");
  5: Write("Five");
  OTHERWISE: Write("Input error");
End;
END.
```

III.6.2) Flowchart

The formalism for the multiple choice structure in a flowchart is as follows:

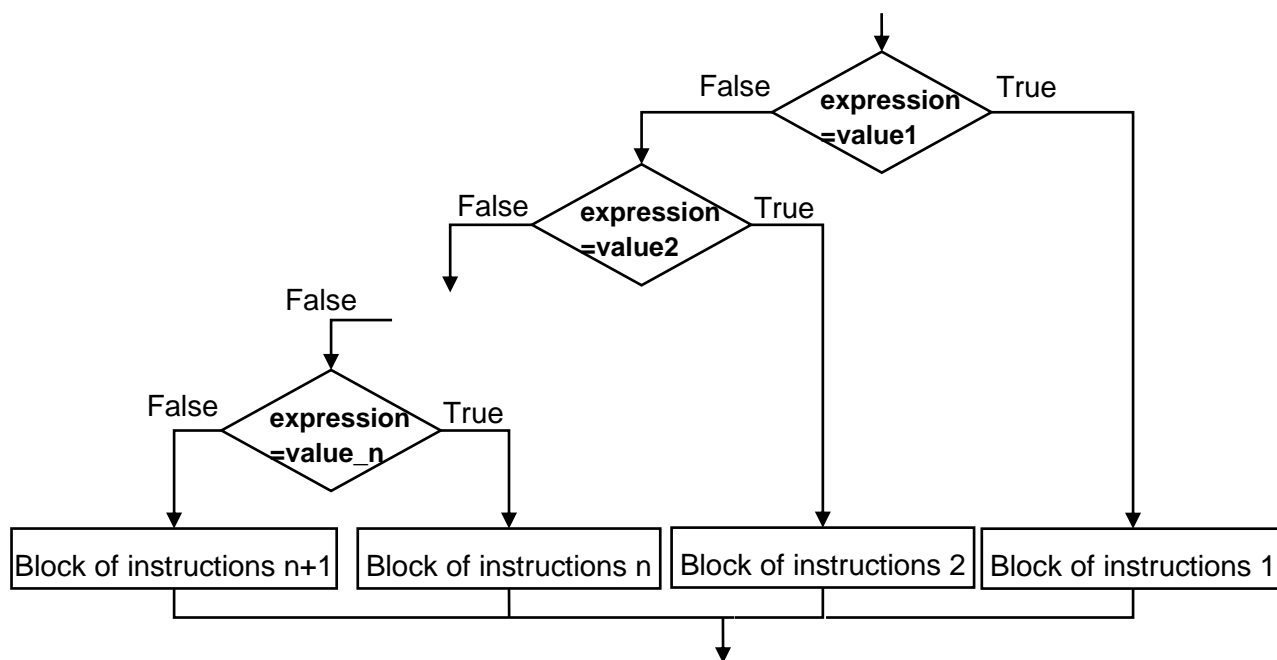


Figure III.6. Flowchart of a multiple-choice structure.

III.6.3) C language Syntax

The multiple-choice conditional structure is expressed in the C language as follows:

```

switch (<expression>) {
    case <value>: <block of instructions1>
                break;
    case <value2>: <block of instructions2>
                break;
    .....
    case <value_n>: <block of instructions n>
                  break;
    default:      <block of instructions n+1>
}
    
```

Remarks:

- The `default` case is optional.
- The `break` keyword is required after each block of instructions to indicate the end of a case. If `break` is omitted, execution continues in the following blocks.
- The instruction blocks should not be enclosed in braces (`{` and `}`).

Example:

The C program that allows to input an integer between 1 and 5 and display it in words is as follows:

```

main() {
    int n;
    printf("Enter a number between 1 and 5: ");
    scanf("%d", &n);
    switch(n) {
        case 1: printf("One");
                break;
        case 1: printf("Two");
                break;
        case 1: printf("Three");
                break;
        case 1: printf("Four");
                break;
        case 1: printf("Five");
                break;
        default: printf("Input error");
    }
}

```

III.7) Branching statement

Sometimes, there is a need to break the sequential flow of an algorithm to bypass certain instructions or to repeat the execution of a set of instructions. This is where the branching instruction comes into play.

The branching instruction allows us to interrupt the normal execution of the algorithm by jumping from one point in the algorithm to another and continuing the execution from that point. This is why they are also called *jump instructions*.

To perform a branch, you first need to identify the instruction in the algorithm to which you want to jump using a *label*. Then, it's possible to jump to that instruction to execute it (and those that follow) by knowing its label.

The label is an identifier assigned to an instruction in the algorithm for the purpose of identifying it. This way, it is possible to go directly to the instruction by knowing its label.

III.7.1) Algorithmic Syntax

To assign a label to an instruction, you simply need to write the label (which is an identifier, as mentioned earlier) followed by a colon ":" before the instruction.

Then, the branching or jump to that label is performed using the **GOTO** instruction while specifying the label.

The syntax is as follows:

```

<Label_name>: instruction i ;
    .....
GOTO A <Label_name>;
    .....

```

Example:

The following algorithm allows entering a student's exam mark and, if applicable, the makeup exam mark, to determine whether the student is admitted or deferred using branching.

```

ALGORITHM example;
VAR exam, makeup : real;
BEGIN
WRITE("Enter the exam mark: ");
READ(exam);
IF exam ≥ 10 THEN GOTO label;
WRITE("Enter the makeup exam mark: ");
READ(makeup);
IF makeup ≥ 10 THEN GOTO label;
WRITE("You are deferred");
GOTO last;
label: WRITE("You are admitted");
last: WRITE("End of the algorithm");
END.

```

Remarks:

- It is possible to jump to:
 - An instruction that precedes the branching instruction, creating a loop effect.
 - An instruction that follows the branching instruction to advance more quickly in the algorithm.
- It is discouraged to use the branching instruction in order to reduce algorithm complexity in terms of time.

III.7.2) C language Syntax

The definition of a label in the C language is done in the same way as in algorithmics, by writing the label followed by a colon ":" before the instruction to be marked.

For branching, it is done using the `goto` statement.

The syntax is as follows:

```

<Label_name>: instruction i ;
                .....
goto <Label_name>;

```

Example:

The program that allows to enter a student's exam mark, and if applicable, the makeup exam mark, and determine whether he is admitted or deferred is as follows:

```

main(){
    float exam,makeup;
    printf("Enter the exam mark : ");
    scanf("%f",&exam);
    if(exam >= 10)goto label;
    printf("Enter the makeup exam mark : ");
    scanf("%f",&makeup);
    if(makeup >= 10)goto label;
    printf("You are deferred");
    goto last;
    label: printf("You are admitted ");
    last: printf("End of the algorithm");
}

```

III.8) Exercises

Exercise III.1 :

Write an algorithm that allows to enter a real number from the keyboard and calculates and displays its absolute value.

Exercise III.2 :

Write an algorithm for entering 3 integer numbers and displaying them in ascending order.

Exercise III.3 :

Write an algorithm that allows a hostess to calculate the price of a ticket based on the passenger's age. Children under or equal to 2 years old do not pay, those under 10 years old (between 3 and 9 years) pay half price, and people between 10 and 27 years old and those at least 70 years old (age equal to or greater than 70) receive a 10% discount. The user should enter the base ticket price and the passenger's age, and the algorithm calculates and displays the price to pay.

Exercise III.4 :

Write an algorithm performing addition, subtraction, multiplication, or division of two numbers based on the user's choice from a menu. The two numbers and the operation to be performed should be entered by the user.

Exercise III.5 :

Write an algorithm that repeatedly prompts for a student's mark (between 0 and 20) until a valid response is given. Use branching statements to achieve this.

III.9) Solution of the exercises

Exercise III.1 :

```

Algorithm absolute_Value ;
Var nb,absV:Real ;
Begin
Write("Enter a real number:");
Read(nb);
absV ← nb;

```

```

If absV<0 Then absV ← absV*(-1);
Write("The absolute value of ",nb," is: ",absV);
End.

```

Exercise III.2 :

```

Algorithm sort;
Var x, y, z: Integer;
Begin
Write("Enter 3 integer numbers:");
Read(x, y, z);
If x ≤ y and y ≤ z Then Write(x, "<", y, "<", z)
Else If x ≤ z and z ≤ y Then Write(x, "<", z, "<", y)
Else If y ≤ x and x ≤ z Then Write(y, "<", x, "<", z)
Else If y ≤ z and z ≤ x Then Write(y, "<", z, "<", x)
Else If z ≤ x and x ≤ y Then Write(z, "<", x, "<", y)
Else Write(z, "<", y, "<", x);
End.

```

Exercise III.3 :

```

Algorithm ticket;
Var age: Integer; basePrice, totalPrice: Real;
Begin
Write("Enter the passenger's age:");
Read(age);
Write("Enter the base ticket price:");
Read(basePrice);
If age ≤ 0 Then Write("Input error")
Else Begin
If age ≤ 2 Then totalPrice → 0
Else If age ≥ 3 and age < 10 Then
totalPrice ← basePrice / 2
Else If (age ≥ 10 and age ≤ 27) or age ≥ 70 Then
totalPrice ← basePrice - basePrice * 10 / 100
Else totalPrice ← basePrice;
Write("The price to pay:", totalPrice);
End;
End.

```

Exercise III.4 :

```

Algorithm menu;
Var a, b, r: Real; op: Character;
Begin
Write("Enter 2 real numbers");
Read(a, b);
Write("Type + for addition");
Write("Type - for subtraction");
Write("Type * for multiplication");
Write("Type / for division");

```



```

Write("Your choice: ");
Read(op);
Case op of
  '+': Begin
        r ← a + b;
        Write(a, "+", b, "=", r);
        End;
  '-': Begin
        r ← a - b;
        Write(a, "-", b, "=", r);
        End;
  '*': Begin
        r ← a * b;
        Write(a, "*", b, "=", r);
        End;
  '/': Begin
        If b ≠ 0 Then Begin
                r ← a / b;
                Write(a, "/", b, "=", r);
            End
        Else Write("Division not possible");
        End;
    Otherwise: Write("Unknown operator");
End;
End.

```

Exercise III.5 :

```

Algorithm enter_mark;
Var mark: Real;
Begin
Write("Enter a mark:");
input: Read(mark);
If mark < 0 or mark > 20 Then
    Begin
        Write("Incorrect mark, enter the mark again:");
        Go to input;
    End;
End.

```

III.10) Conclusion

In this chapter, we have seen that simple sequential structures are not sufficient to solve complex problems that involve multiple cases of processing. Such problems are solved using conditional structures, which allow distinguishing between multiple cases and providing a specific processing for each case. We have introduced various conditional structures and explained their usage: simple conditional structures, compound conditional structures, nested tests, and branching.

Chapter IV. Loops

IV.1) Introduction

We have previously established that, in general, instructions in an algorithm are executed sequentially, one after another, one time only. We have also learned that the sequential flow of instructions is not sufficient to solve everyday problems, and fortunately, it is possible to break the sequential flow using what are called *control structures*.

In the previous chapter, we discussed the first problem that cannot be solved by simple sequential instructions, which is the problem where we have multiple cases, and each case requires separate processing. This type of problem was resolved by using *conditional structures*.

Another problem frequently encountered in everyday life is the need to repeat a task multiple times.

Let's take an example. Suppose we want to enter a person's age. It is known that age must be strictly positive, but nothing prevents the user from entering an incorrect value (negative or zero). However, conditional structures allow us to perform a check on the entered value and only accept values strictly greater than 0. Otherwise, we display an error message to the user. The corresponding code can be as follows:

```
Write("Enter the age: ");
Read(age);
If age ≤ 0 Then write("Input error")
Else write("valid age");
```

But, is the problem solved? No, this piece of code reads the age and tests its value only once. If we are lucky and the entered value is correct, we can continue with the execution of the algorithm. Otherwise, an error message is displayed, and the algorithm terminates without reading a valid age.

If we are sure that the user will provide a valid age on the second attempt, we can add another read instruction within the **ELSE** block, and the problem is solved (see the code below):

```
Write("Enter the age: ");
Read(age);
If age ≤ 0 then
    Begin
        Write("Invalid age, reenter the age:");
        Read(age);
    End
Else write("Valid Age ");
```

Indeed, we cannot guarantee that the value entered the second time is correct. It is necessary to add another test to ensure the validity of the provided age.

```

Write("Enter the age: ");
Read(age);
If age ≤ 0 then
    Begin
        Write("Invalid age, reenter the age:");
        Read(age);
        If age ≤ 0 then
            Begin
                Write("Invalid age, reenter the age:");
                Read(age);
            End
        Else write("Valid Age ");
    End
Else write("Valid Age ");

```

This code works even if the user makes mistakes twice; he will be asked to enter their age for the third time. However, beyond two attempts, it is not functional. We would need to add a read instruction and a test for the value being checked, and so on for each erroneous input. But how many times? The number of times the user will provide a valid age is unknown, so we cannot predict how many **IF - ELSE** blocks should be nested to obtain a valid age.

Therefore, this repetitive processing can be achieved using branching statements with labels, but a program that uses labels can be challenging to maintain. The ideal solution for implementing repetitive processes is the use of special control structures called **loops**. Loops allow to repeat an instruction or a sequence of instructions a certain number of times, which can be known in advance or not.

In this chapter, we will introduce the concept of repeating a sequence of instructions and present the various loop structures in algorithmics.

IV.2) What is a loop ?

IV.2.1) Definition

Loops, also known as **repetitive** or **iterative structures**, are structures that allow to repeat or redo the same sequence of instructions multiple times with different values for a finite number of times.

During each repetition, the instructions within the loop are executed. This is known as a loop **iteration**.

The iteration stops when a termination condition is met, which is expressed either by a logical expression or by a predefined number of iterations.

However, there are three variants of repetition, and for each variant, algorithmics (and most programming languages) offers a specific type of loop:

- Repeating a block of instructions a given number of times (**FOR** loop).
- Repeating a block of instructions as long as a condition is met (**WHILE** loop).
- Repeating a block of instructions until a condition is met (**REPEAT** loop).

IV.2.2) Components of a loop

A loop consists of four essential elements:

- A **block of instructions**, which will be executed a certain number of times.
- A **condition**, similar to conditional instructions. This condition relates to at least one variable, referred to as *the loop variable*. There can be multiple loop variables for a single loop.
- An **initialization**, which concerns the loop variable. This initialization can be directly performed by the loop statement or left to the programmer.
- A **modification**, which also concerns the loop variable. Similar to initialization, it can be integrated into the loop statement or left to the programmer.

IV.3) While loop

The **WHILE** loop allows to repeatedly execute an instruction or sequence of instructions as long as a condition is met. When the condition becomes false, the loop terminates. The condition is expressed in the form of a variable or logical expression.

This loop is particularly useful when the number of iterations is not known in advance.

IV.3.1) Algorithmic Syntax

The syntax of the **WHILE** loop is as follows:

```
WHILE <Condition> DO
    <block of instructions>;
```

The progression of the **WHILE** loop involves successively and repeatedly the following steps. First, the loop's entry condition is evaluated. If it is satisfied, the body of the loop (the block of instructions) is executed, and we return to evaluate the condition again. This process continues until the condition is no longer satisfied. In this latter case, the instructions within the block are not executed, and the algorithm proceeds to the next instruction just after the block.

Example:

Write an algorithm that allows to enter a person's age via the keyboard and to repeat the entry as long as the value entered by the user is incorrect.

Solution :

```
ALGORITHM input_age;
VAR age : Integer;
BEGIN
Write("Enter the age: ");
Read(age);
WHILE age ≤ 0 DO
    Begin
        Write("Invalid age, re-enter the age:");
        Read(age);
    End;
Write("Valide age ");
END.
```

Remarks :

- The condition can be simple or compound just like in conditional structures.

- Note the absence of a semicolon after the condition and after **DO**.
- In this loop, the condition is tested before entering the loop. Therefore, the block of instructions that forms the body of the loop may never be executed; this happens when the condition is false from the beginning.
- The parameters of the condition must be initialized by reading or assignment before the loop, so that, on the first pass, the condition can be evaluated.
- In the block of instructions, it is imperative to have an action that modifies the condition parameters in such a way that the condition becomes false at some point, otherwise, if the condition remains true, you end up in an **infinite loop**.
- If the block of instructions to be repeated contains multiple instructions, it must be enclosed by **BEGIN** and **END**.
- The **WHILE** loop is the most generic loop. It can be used whether the number of repetitions is known in advance or not.

IV.3.2) Flowchart

The formalism of the **WHILE** loop in a flowchart is as follows.

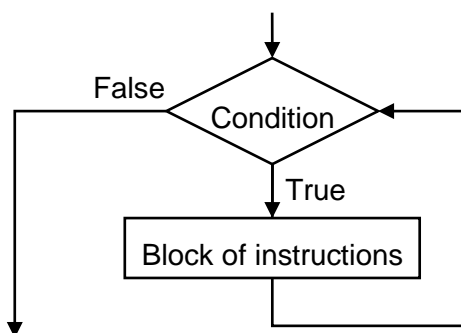


Figure IV.1. Formalism of the **WHILE** loop in a flowchart

Example:

The flowchart corresponding to the algorithm that repeats the age input until a valid age is entered using a **WHILE** loop is as follows:

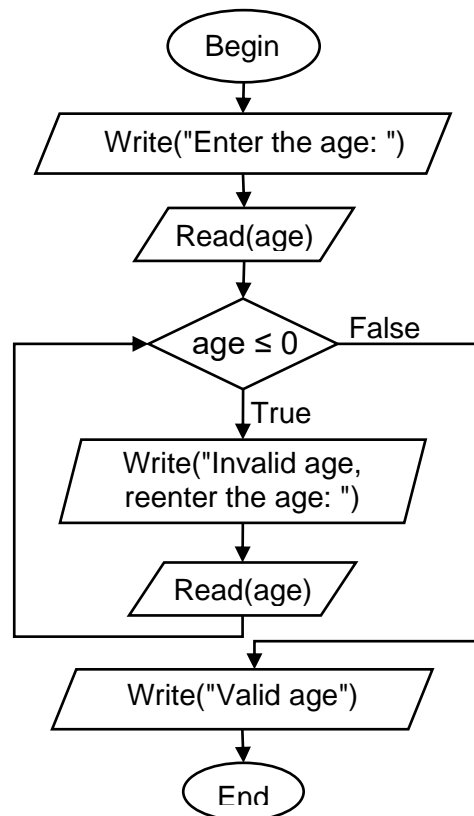


Figure IV.2. Flowchart for repeating age input using **WHILE** loop

IV.3.3) C language syntax

The syntax of the **WHILE** loop in the C language is as follows:

```
while(<condition>) <block of instructions>;
```

Remarks:

- The condition must be enclosed in parentheses.
- There is no semicolon after the condition.
- If the block of instructions consists of multiple statements, it must be enclosed in curly braces ({ and }).

Example:

Here is the C program that repeats reading the age until a valid age is entered:

```
#include<stdio.h>
main(){
    int age;
    printf("Enter the age: ");
    scanf("%d",&age);
    while(age <= 0){
        printf("Invalid age, re-enter the age:");
        scanf("%d",&age);
    }
    printf("Valid age");
}
```

IV.4) REPEAT loop

The **REPEAT** loop allows to repeat the execution of a block of instructions until a condition is met. Like the **WHILE** loop, **REPEAT** is a generic loop that doesn't require knowing the number of iterations in advance.

In contrast to the **WHILE** loop, the **REPEAT** loop unconditionally executes the block of instructions first and then repeats its execution as long as the condition is false. The loop execution stops as soon as the condition becomes true.

IV.4.1) Algorithmic syntax

The syntax of the **REPEAT** loop in algorithmic is as follows:

```
REPEAT  
  <block of instructions>;  
UNTIL <Condition>;
```

The progression of the **REPEAT** loop can be described as follows. First, the block of instructions that makes up the body of the loop is executed for the first time. Then, the condition is evaluated. If it's true, the block of instructions is executed again, and the condition is re-evaluated. This process repeats until the condition is satisfied. In this case, the loop is exited, and the normal execution of the algorithm continues.

Example:

Re-implement the algorithm that repeats the input of a person's age until a valid age is provided, but this time using the **REPEAT** loop.

Solution :

```
ALGORITHM input_age;  
VAR age : Integer;  
BEGIN  
  REPEAT  
    Write("Enter the person's age:");  
    Read(age);  
  UNTIL age > 0;  
  Write("Valid age");  
END.
```

Remarks :

- The condition can be simple or compound.
- Note the absence of a semicolon after **REPEAT** and **UNTIL**.
- In this loop, the condition is only evaluated at the end of the loop. Therefore, the block of instructions that forms the body of the loop is executed at least once, even if the condition is satisfied from the beginning. Thus, the first execution is not subject to any condition.
- The condition in the **REPEAT** loop is the exit or termination condition of the loop, not the repetition condition, as is the case with the **WHILE** loop.
- The variables on which the condition is based must be initialized by reading or assignment before the condition is evaluated (and not before the loop).

- The body of the loop must contain an instruction that modifies the condition parameters to reach the exit condition at some point; otherwise, you would end up in an infinite loop.
- The block of instructions does not need to be enclosed in **BEGIN** and **END**, even if it consists of multiple instructions.

IV.4.2) Flowchart

The **REPEAT** loop can be represented in a flowchart as follows:

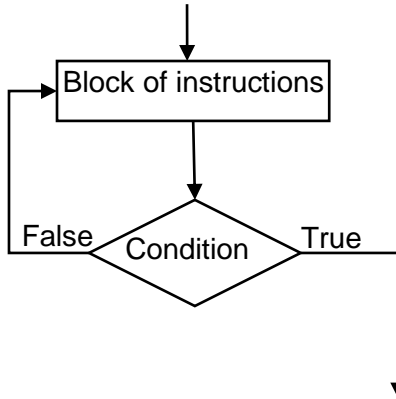


Figure IV.3. Formalism of the **REPEAT** loop in a flowchart.

Example:

The flowchart corresponding to the algorithm from the previous example is as follows:

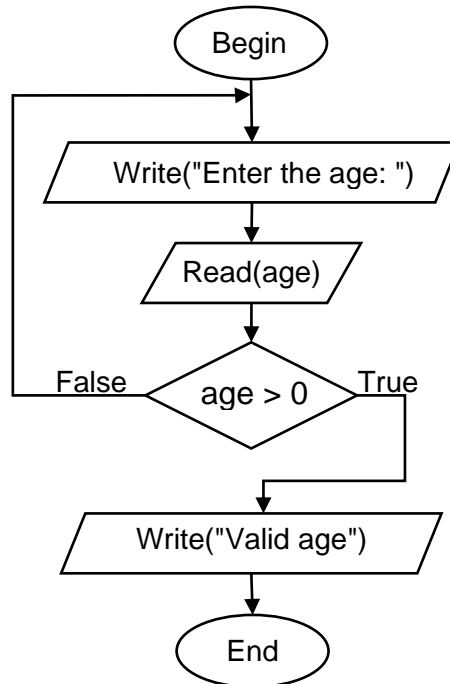


Figure IV.4. Flowchart for Repeating Age Input Using the **REPEAT** Loop

IV.4.3) C language syntax

The C language loop corresponding to **REPEAT** is the **do-while** loop. It is introduced by the **do** statement, followed by the block of instructions, and finally, the condition enclosed in parentheses, placed after a **while**, just like the **while** loop.

The syntax is the follows:

```
do{
    <block of instructions>;
}while(condition) ;
```

In fact, the meaning of the **REPEAT** loop in algorithmics is a bit different from that of the **do-while** loop in the C language. The difference lies in the loop's condition.

However, in algorithmics, it's "repeat until the condition is satisfied," while in C, it's "repeat as long as a condition is satisfied". Therefore, the condition in **REPEAT** is an exit condition, whereas in **do-while**, it's an entry condition to the loop. It's similar to a **while** loop, except the condition is at the end of the loop.

Remarks:

- The condition must be enclosed in parentheses.
- Note the presence of a semicolon after the condition.
- If the block of instructions contains only one instruction, the curly braces (**{** and **}**) are not obligatory.
- The condition in **do-while** is the inverse of the condition in **REPEAT**.

Example:

The C program that repeats reading the age using the do-while loop until a valid age is entered is as follows:

```
#include<stdio.h>
main(){
    int age;
    do{
        printf("Enter the person's age: ");
        scanf("%d",&age);
    }while(age <= 0);
    printf("Valid age");
}
```

IV.4.4) Difference between WHILE and REPEAT

The two loops, **WHILE** and **REPEAT**, share similarities in that both allow the repetition of a sequence of instructions even when the number of repetitions is not known in advance. However, they do have some differences that should be listed in order to use both loops correctly.

The following table summarizes the differences between the **WHILE** and **REPEAT** loops:

While	Repeat
The condition is evaluated before the block of instructions.	The condition is evaluated after the block of instructions.
The condition is an entry condition to the loop.	The condition is a termination or exit condition for the loop.
The block of instructions may never be executed.	The block of actions is executed at least once.
The condition variables must be initialized before the loop.	The condition variables can be initialized after entering the loop, within the block of instructions.
If the block of instructions contains multiple statements, it must be enclosed between BEGIN and END .	No need to enclose the block of instructions with BEGIN and END , as REPEAT - UNTIL serves this purpose.

Table IV.1. Differences between the **WHILE** and **REPEAT** loops

IV.5) For loop

The first two loops, **WHILE** and **REPEAT**, are generic loops that can be used even when the number of repetitions is not known. However, in many cases, we deal with problems where the number of repetitions is known in advance. In such cases, whether using **WHILE** or **REPEAT**, we use a counter and stop when the counter reaches its pre-known final value.

That's why algorithmics and most programming languages offer a more convenient structure for writing this kind of repetitive task more simply. This structure is the **FOR** loop.

The **FOR** loop allows the execution of a block of instructions a certain number of times known in advance. This loop automates the initialization and modification phases of the loop variable, making it a powerful tool for managing repetitive tasks with a known number of iterations.

IV.5.1) Algorithmic Syntax

In this loop, a control variable of integer type, called the **counter**, is used to control the number of iterations of the loop. This variable keeps track of the number of iterations performed to determine when to exit the loop. The loop continues until the desired number of iterations is reached.

The counter takes its values in an interval whose bounds are known. Thus, in the header of the **FOR** statement, you must specify the initial value, the final value, and optionally the step (when it's different from 1).

The syntax of the **FOR** loop in algorithmic is as follows:

```
FOR <counter> ← <initial value> TO <final value> STEP=<step value> DO <Block of instructions>;
```

Such as:

- **<counter>** is the control variable (of **integer** type) that counts the number of loop iterations.
- **<initial value>** is the initial value to which the counter is initialized. It can be a constant or an integer-type variable.
- **<final value>** is the final value at which the counter ends. It can also be a constant or an integer-type variable.
- **<Step Value>** is the increment or decrement value for the counter. The step can be omitted if its value is 1.

The block of instructions is executed each time the counter's value is between the **<initial value>** and the **<final value>**. The progression of the **FOR** loop can be described as follows:

First, the **<counter>** is initialized to the **<initial value>** at the moment of entering the loop. If the **<counter>**'s value does not exceed the **<final value>**, the **<block of instructions>** is executed, and the **<counter>** is automatically increased (incremented) by the increment value **<step value>**. When the increment is not specified, the default increment is 1. This process repeats until reaching the **<final value>**. In this case, the loop terminates, and execution continues normally after the loop.

Example:

Using the **FOR** loop, write an algorithm to display natural numbers from 1 to 5.

Solution :

```
ALGORITHM display_numbers;
VAR i : Integer;
BEGIN
FOR i ← 1 TO 5 STEP=1 DO
    Write(i);
END.
```

Remarks :

- The **FOR** loop can only be used when the number of iterations is known in advance.
- A **FOR** loop can be executed 0 times (when the final value is less than the initial value), 1 time (when the initial value and the final value are the same), or multiple times (the normal case).
- The initial value, the final value, and the increment step can be numeric expressions.
- In the **FOR** loop, the initialization of the loop variable (the counter), its modification (the increment of the counter), and the evaluation of the stopping condition are performed automatically.
- The increment step is optional. If omitted, its default value is 1.
- The increment step can also be negative, and in that case, the counter is decremented by the increment step at each iteration.
- In the body of the loop, the counter can be used for calculations, but it must not be modified either by reading or by assignment.

- The number of iterations in the **FOR** loop is equal to: $\langle \text{final value} \rangle - \langle \text{initial value} \rangle + 1$ (when the increment step is equal to 1).
- The initial and final values, and the increment step are evaluated once and for all before the iteration; the body of the loop cannot modify their value.
- If the block of instructions consists of 2 or more instructions, it must be delimited by the keywords: **BEGIN** and **END**.

IV.5.2) Flowchart

The **FOR** loop can be graphically represented in a flowchart as follows:

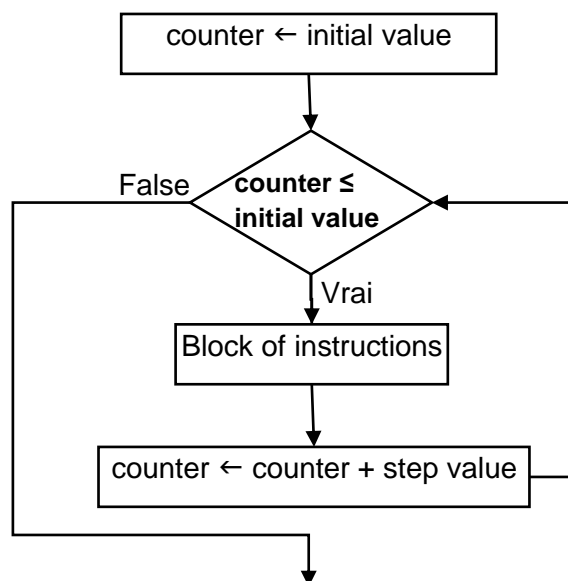


Figure IV.5. Formalism of the **FOR** loop in a flowchart.

Example:

The flowchart corresponding to the previous example is as follows:

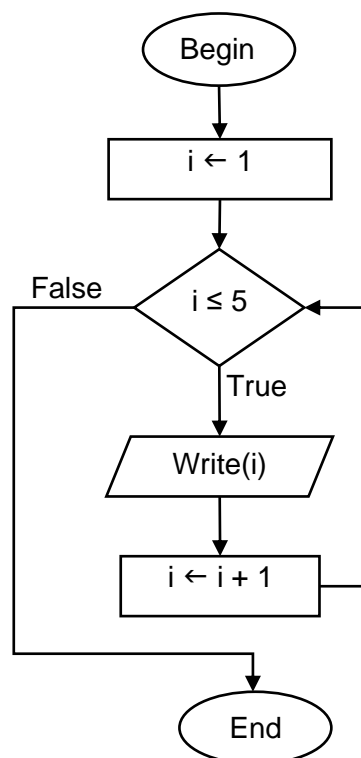


Figure IV.6. Flowchart for Displaying Integers from 1 to 5 Using a **FOR** Loop.

IV.5.3) C language Syntax

The loop corresponding to **FOR** in the C language is also called the **for** loop. However, the syntax of the latter in C language is a bit different from that of **FOR** in algorithmics.

The syntax in C of the **for** loop is as follows:

La syntaxe de la boucle **for** est la suivante:

```

for (<initialization>;<condition>;<modification>)
  <block of instructions>;
  
```

Thus, the header of the **for** loop is composed of three expressions separated by semicolons within parentheses:

- The first expression (<initialization>) is an initialization expression. It is executed only once at the beginning of the loop and is typically in the form: **<counter> = <initial value>**.
- The second (<condition>) is a comparison expression. It is evaluated at the beginning of each iteration, including the first one.
- The last (<modification>) is a progression expression. This expression is used to increment (or decrement) the loop counter and is executed at the end of each iteration.

The execution of the **for** loop proceeds as follows. At the beginning of the for loop, the <initialization> statement is executed. Then, the <condition> is tested. If the condition is true, the instructions within the **for** loop's body are executed, followed by the <modification> statement. The <condition> is re-evaluated with the new <counter>

value before the next iteration, and so on, as long as the `<condition>` remains true. Once the `<condition>` becomes false, the loop terminates.

Remarks:

- It is possible to initialize/modify multiple loop variables simultaneously by using commas in the expressions.
- When the block of instructions consists of more than one statement, it must be enclosed in curly braces (`{` and `}`).

Example:

The C program that displays natural numbers from 1 to 5 is as follows:

```
#include<stdio.h>
main() {
    int i;
    for(i=1;i<=5;i++)
        printf("%d\n",i);
}
```

IV.6) Choice of the appropriate repetitive structure

The choice of the appropriate repetitive structure depends on the problem to be solved.

If the number of repetitions is known in advance, it is advisable to use the **FOR** loop. On the other hand, if the number of iterations is not known in advance, either the **WHILE** loop or the **REPEAT** loop should be used.

However, the choice between these two loops is possible and depends on the minimum number of repetitions desired. If you want to execute the instructions in the block at least once, it is recommended to use the **REPEAT** loop. When the number of iterations can be zero, the **WHILE** loop must be used.

IV.7) Nested loops

As we have seen before, loops execute one or more instructions (block of instructions) a certain number of times. These instruction blocks can, in turn, contain loops. In this case, we refer to them as **nested loops**.

Hence, a **WHILE** loop can contain another **WHILE** loop, another **REPEAT** loop, or another **FOR** loop, and vice versa.

The principle of nested loops can be described as follows:

1. We enter the outer loop (which encloses the inner loop).
2. We enter the inner loop and traverse it until its exit condition is met.
3. We exit the inner loop and return to the level of the outer loop.
4. We iterate over the outer loop, returning to the inner loop.
5. And so on, repeating this process as needed.

Example:

Consider the following algorithm:

```

Algorithm nested_loops;
Var i,j:integer;
Begin
  i ← 1;
  WHILE i ≤ 3 DO
    Begin
      j ← 1;
      WHILE j ≤ 2 DO
        Begin
          Write(i+j);
          j ← j + 1;
        End;
      i ← i + 1;
    End;
  End.

```

The step-by-step execution of the algorithm is summarized in the table below:

Iteration	i	j	Displayed value
1	1	1	2
2		2	3
3	2	1	3
4		2	4
5	3	1	4
6		2	5

Table IV.2. Example of step-by-step execution of an algorithm.

IV.8) Exercises

Exercise IV.1 :

Write an algorithm that takes in a multiple of 3 and determines how many times it is divisible by 3.

Example:

The number 81 is divisible by 3 four times because $81 = 3 * 3 * 3 * 3$.

Exercise IV.2 :

A person has bought a new car at a price P . Knowing that the car's value decreases at a fixed rate each year, the person wants to resell the car before it loses half of its initial value.

Write an algorithm that asks the user to enter the price of the car and the rate of price decrease, and calculates and displays the number of expected years of use.

Exercise IV.3 :

Write an algorithm that calculates the factorial ($n!$) of an integer n read from the keyboard.

Reminder: $n! = 1 \times 2 \times \dots \times n$.

Exercise IV.4 :

Write an algorithm that allows to read a positive non-zero integer x and calculates the following sum up to the n^{th} term (n is entered by the user):

$$S = \sum_{i=1}^n \left(x + \sum_{j=1}^{i-1} j \right)$$

For example, if $x = 3$ and $n = 4$, the sum S is:

$$S = 3 + (3+1) + (3+1+2) + (3+1+2+3) = 3+4+6+9 = 22$$

IV.9) Solution of the exercises**Exercise IV.1 :**

```

Algorithm calculation;
Var nb, nbDiv3, x: Integer;
Begin
Write("Enter an integer multiple of 3:");
Read(nb);
If nb mod 3 ≠ 0 Then Write("Input error")
Else Begin
    nbDiv3 ← 0;
    x ← nb;
    While x mod 3 = 0 Do
        Begin
            x ← x / 3;
            nbDiv3 ← nbDiv3 + 1;
        End;
    Write(nb, " is divisible by 3 ", nbDiv3, " times");
End;
End.

```

Exercise IV.2 :

```

Algorithm car;
Var p, rate, x: Real; numYears: Integer;
Begin
Write("Enter the purchase price of the car: ");
Read(p);
Write("Enter the depreciation rate: ");
Read(rate);
If p≤0 or rate<0 or rate>100 Then Write("Input error")
Else Begin
    numYears ← 0;
    x ← p - p * rate / 100;
    While x > p / 2 Do
        Begin
            numYears ← numYears + 1;
            x ← x - x * rate / 100;
        End;
    End;
End.

```



```

        End;
        Write("The expected number of years is ", numYears);
        End;
    End.

```

Exercise IV.3 :

```

Algorithm factorial;
Var n, i, fact: Integer;
Begin
Write("Enter a positive integer: ");
Read(n);
If n < 0 Then Write("Input error")
Else Begin
    fact ← 1;
    For i ← 1 to n Do
        fact ← fact * i;
    Write(n, "! = ", fact);
    End;
End.

```

Exercise IV.4 :

For example, for $x = 3$ and $n = 4$,

$$i = 1 \quad 2 \quad 3 \quad 4$$

$$S = 3 + (3+1) + (3+1+2) + (3+1+2+3) = 3+4+6+9 = 22$$

The solution involves using nested loops as follows:

```

Algorithm sum;
Var x, n, i, j, s, a: Integer;
Begin
Write("Enter x and n: ");
Read(x, n);
If x ≤ 0 or n ≤ 0 Then Write("Input error")
Else Begin
    s ← 0;
    For i ← 1 to n Do
        Begin
            a ← x;
            For j ← 1 to i - 1 Do
                a ← a + j;
            s ← s + a;
        End;
    Write("S=", s);
    End;
End.

```

IV.10) Conclusion

In this chapter, we have delved into an essential control structure for managing repetitive tasks within an algorithm: the concept of loops. We've introduced the three main types of loops, highlighted their distinctions, and demonstrated their practical application in algorithmic problem-solving, flowchart representations, and the C programming language. Additionally, we've emphasized that the nesting of multiple loops is a powerful approach to address more intricate and multifaceted problems, showcasing the versatility of loops in algorithmic solutions.

Chapter V. Arrays and Strings

V.1) Introduction

Up to this point, we have dealt only with simple data, which we stored in variables of predefined simple types. To recap, a variable of a predefined simple type can store only a single value that matches its data type. For instance, an **Integer** variable can store an integer value, a **Real** variable can store a real number, and so on.

However, simple variables are insufficient for addressing complex problems that involve a large amount of data and require preserving the entered data for subsequent processing. Let's consider an example.

Suppose we want to input the marks of a class of 5 students and calculate the class average. Up to now, using what we have learned in the previous chapters, we would create a separate variable for each mark, input the 5 marks individually, and then calculate the sum and average of the marks in a single instruction. The corresponding algorithm might be as follows:

```
Algorithm calculation;
Var n1,n2,n3,n4,n5,moy:real;
Begin
Read(n1,n2,n3,n4,n5);
moy ← (n1+n2+n3+n4+n5)/5;
Write(moy);
End.
```

It works, but it would become much more complicated if we had to calculate the average for a class of 200 students. It would require declaring 200 variables! One solution to this problem is to use a single variable to store all the marks and employ a loop to read the 200 values one by one, accumulating them in a variable as they are read. At the end of the loop, all that remains is to divide the sum by 200 to find the average. The code for performing this operation is as follows:

```
Algorithm calculation;
Var note,s,moy:real;i:integer;
Begin
  s ← 0;
  For i ← 1 To 200 Do
    Begin
      Read(note);
      s ← s + note;
    End;
  moy ← s/200;
  Write(moy);
End.
```

This is an improvement. The code is functional if the sole objective is to simply calculate the class average. However, is this method of reading the grades sufficient if we want to establish a ranking of students or determine the number of marks that are above the class average?

Unfortunately, it wouldn't be. Here, we need to be able to access the entered marks again, which is not possible with the previous code where all values except the last one were lost due to the use of a single variable during input. Therefore, it's necessary to be able to store these 200 marks. This brings us back to the previous problem, the complexity of writing the algorithm when dealing with a large number of variables.

An alternative to this problem is to consider using a single variable that can hold all the values. Fortunately, in algorithmics, there are other data types besides basic types, known as **compound types** or **structured types**, which are constructed from basic types or other declared types, and represent a collection of multiple values.

In algorithmics, the term **data structure** is often used instead of structured or compound type. A data structure is a specific way of storing and organizing data to process it more easily and efficiently.

However, various types of data structures exist to address very specific problems, such as arrays, records (structures), linked lists, and more.

In this chapter, we will focus on the first structured type that we consider the most important and suitable for the previous problem: namely, the **array** type. **Strings**, which are considered a specific case of arrays, will also be discussed in this chapter. For each of these types, we will cover declaration, manipulation, and provide some usage examples.

Other data structures will be explored in later chapters.

V.2) The Array type

V.2.1) Definitions

V.2.1.1) Array

An array is a homogeneous data structure that gathers, under a single name, a finite set of elements of the same data type. Therefore, we can have an array of integers, an array of real numbers, an array of characters, and so on.

The elements of the array are arranged in sequentially numbered adjacent **cells**. Each element can be identified by its position in the set.

In memory, an array is essentially a memory space dimensioned to accommodate a finite number of contiguous cells or zones, ensuring the storage of multiple elements of the same data type. The elements of the array, therefore, occupy adjacent memory locations. The first element is followed by the second, followed by the third, and so on.

The main advantage of arrays is their ability to represent a set of values using a single identifier.

As such, an array is characterized by:

- A name: an identifier for the array.
- The type of its elements.
- A size or length.
- An index, which is the indicator that allows us to traverse the array's cells one by one.

It's worth noting that an array can be one-dimensional (vector), two-dimensional (matrix), or multi-dimensional. In this initial part of the chapter, we focus on one-dimensional arrays.

V.2.1.2) Array Element

An element refers to one of the values contained within the array.

V.2.1.3) Index

An index is a scalar variable, typically of integer type, that allows access to the elements of an array. It indicates the rank or position of an element within the array.

V.2.1.4) Size

The size, length, or cardinality of an array, sometimes referred to as dimension, is the number of its elements. The size can only be a constant or a constant expression. Consequently, the size cannot be modified during the execution of the algorithm.

Example :

Consider the **Not** array below, which consists of 5 real elements arranged in 5 cells. Each element represents a student's mark.

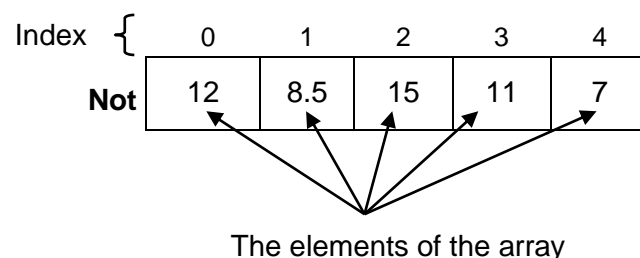


Figure V.1. Diagram of an array with 5 elements.

V.2.2) Declaration

The declaration of an array is done by specifying its name, size, and the type of the elements stored in the array. It's important to note that during the declaration of an array, a contiguous block of memory is allocated for it, which must be defined before its usage and cannot be changed.

We will first present the algorithmic syntax of declaration, followed by the syntax in the C language.

V.2.2.1) Algorithmic Syntax

In algorithmics, we declare an array variable in a similar manner to how we declare simple variables, with the exception that we specify that the declared variable is an array. This is done by adding the keyword **ARRAY**, followed by the size within square brackets, and then specifying the data type.

The declaration syntax is as follows:

```
Var <name_Array> : ARRAY[<size>] OF <type_elements>;
```

With:

- **<name_Array>**: This is the identifier that represents the name of the array.
- **<size>**: The number of elements in the array. This number can be specified as a direct value or as a previously declared constant to increase the flexibility of the algorithm. The size should be placed within brackets after the keyword **ARRAY**.
- **<type_element >**: The type of elements in the array. It can be a standard predefined type (**integer**, **real**, etc.), a non-standard type (interval, enumerated, etc.), or a structured type (array, etc.).

Example:

The previous **Not** array can be declared directly as:

```
Var Not:ARRAY[5] OF Real;
```

Or it can be declared as:

```
Const card=5;
Var Not:ARRAY[card] OF Real;
```

Remarks:

- Declaring an array as **T[N]**, where **N** is a variable number of elements, is incorrect because **N** must be a constant value.
- Like any other types, you can declare multiple arrays with the same characteristics (same size and element type), separating them with commas.
- Indices are not necessarily integers. They can be of any other scalar type. In this case, the size of the array does not need to be explicitly specified.
- The element type of the array should not be confused with the type of the index, which is usually an integer.
- Once an array has been declared, you cannot change its size; it involves static memory allocation. Therefore, you need to define an array that is large enough for the task at hand.

Examples:

1. The following declaration :

```
Var A:Array[5] of Real;
Var B:Array[5] of Real;
Var C:Array[5] of Real;
```

Is equivalent to:

```
Var A,B,C: Array[5] of Real;
```

- The following array is an array with 4 logical elements, and the indices are characters:

```
Var A: Array['e'] of Boolean;
```

This array can be represented graphically as follows:

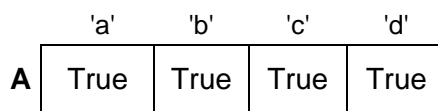


Figure V.2. Diagram of the logical array A

V.2.2.2) C language Syntax

In the C language, the declaration of an array variable follows a similar procedure to declaring regular variables, with the addition of specifying the array size.

Therefore, declaring an array in the C language involves specifying the type of the elements within the array, followed by the array name and its size within square brackets. The declaration syntax is as follows:

```
<type_elements> <name_Array>[<size>];
```

Where:

- <type_elements>** is the type of the elements in the array,
- <name_Array>** is an identifier that represents the name of the array,
- <size>** is the number of elements in the array. This number can be specified directly or as a pre-declared constant.

Examples:

- The declaration in C language for the previous array **Not** can be done by:

```
float Not[5];
```

Or, you can specify the number of elements as a pre-declared constant, like this:

```
#define card 5
float Not[card];
```

- The following declaration defines an array of 4 integer elements with character indices:

```
int L['e'];
```

V.2.3) Manipulation of arrays

The elements of the array are manipulated individually, and therefore, they have the characteristics of any other variables. Consequently, they can be read, displayed, used in assignments, expressions, or comparisons, just as you would with regular variables.

V.2.3.1) Accessing Array Elements

Regardless of the intended purpose of manipulation, we need a way to access the elements of an array individually.

Individual access is achieved by specifying the index of the element (its position in the array). Thus, to access an element of an array, we use the name of the array followed by the index of the element enclosed in square brackets. The syntax for this access is as follows:

<name_Array>[<index>]

With **<name_Array>** being the name of the array and **<index>** as an expression (typically an integer variable) that determines the index or position of the selected element in the array.

This syntax is common in algorithmics and in the C language.

It's important to note that the numbering of array elements starts from 0, not from 1 as in mathematics. Therefore, for an array of size **N**, the cells are numbered from 0 to **N - 1**.

Example:

If **T** is an array of 6 integer elements declared as follows:

```
Var T:ARRAY[6] OF Integer;
```

Suppose **T** is the following:

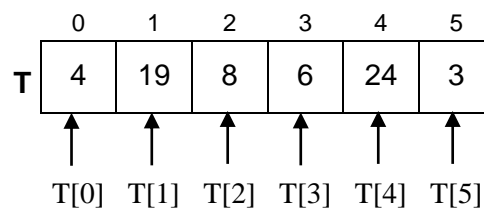


Figure V.3. Example of an array T

- The first element of the array **T** has an index of 0. It is denoted as **T[0]**, and in the example, its value is 4.
- The third element in the array has an index of 2, and is denoted as **T[2]**. It contains the value 8 in the above array.
- The last element (with an index of 5) is denoted as **T[5]**. It contains the value 3.

Remarks:

- The value inside the brackets during the array declaration (the maximum size) should not be confused with the value inside the brackets when used in instructions (the index).
- The index used to refer to the elements of an array can either be a direct, explicit value (e.g., **T[2]**), but it can also be a variable (e.g., **T[i]**, where **i** is a variable, typically an integer) or a computed expression (e.g., **T[i+2]**).
- Accessing an element with an index greater than or equal to the size of the array will consistently result in an incorrect result and often a memory error (attempt to access a memory location outside the array). This is referred to as an **array overflow** (or **index overflow**). However, most compilers do not implement index overflow control. This kind of control is left to the responsibility of the programmers.

V.2.3.2) Filling an array

Le remplissage du tableau peut se faire par affectation, ou par lecture au clavier.

The declaration of an array only reserves space in the computer's memory. Therefore, before any processing, you need to fill the array by assigning values to its elements.

Filling the array can be done through assignment or by reading from the keyboard.

a) Filling by reading

Since each element is manipulated individually, reading an array involves reading all of its elements, one by one. Therefore, you can fill an array with a series of read instructions, like this:

```
Read(<name_Array>[0]);
Read(<name_Array>[1]);
.....
Read(<name_Array>[<size>-1]);
```

Where **<name_Array>** is the name of the array, and **<size>** is the number of elements in the array.

However, since the same **Read** instruction is repeated **<size>** times (only the index value changes), we can avoid this repetition by using loops, specifically the **For** loop (as the number of repetitions is known in advance) for reading the elements of an array. The filling of an array is then accomplished using the following loop:

```
For <indice> ← 0 To <size>-1 Do
    Read(<name_Array>[<indice>]);
```

With **<index>** being a variable (usually of integer type) used as an index to access the elements of the array.

In the C language, this can be achieved with:

```
for(<index>=0; <index> < <size>;<index>++)
    scanf("<format>", &<name_Array>[<index>]);
```

Here, **<format>** is the format for reading the elements of the array ("%d", "%f", ...).

Example:

Filling the array **Not** through reading in the previous example is done by:

```
For i ← 0 To 4 Do
    Read(Not[i]);
```

And in C language by:

```
for(i=0; i<5; i++)
    scanf("%f", &Not[i]);
```

b) Filling by assignment

The array can also be filled by directly assigning values to its elements. This assignment is done using the assignment statement ("**←**" in algorithmics and "**=**" in the C language).

However, when we want to assign a value to a specific element of the array, you use the syntax:

```
<name_Array>[<index>] ← <value>;
```

Here, `<name_Array>` is the name of the array, `<index>` is the index of the targeted element, and `<value>` is the value to be placed in the element.

If we want to assign the same value to all elements of the array, loops are perfectly suited for this type of operation. Filling an array by assignment with the value `<value>` is done as follows:

```
For <index> ← 0 To <size>-1 Do
    <name_Array>[<index>] ← <value>;
```

Here, `<index>` is a variable (usually of `integer` type) used as an index to access the elements of the array, and `<size>` is the number of elements in the array.

In C language:

```
for(<index>=0; <index> < <size>;<index>++)
    <name_Array>[<index>] = <value>;
```

Examples:

- To store the value 15.75 in the second element of the previous **Not** array, we write:

```
Not[1] ← 15.75;
```

- Let **T** be an array of 6 integers. The initialization of all elements of **T** to 0 is done by:

```
For i ← 0 To 5 Do
    T[i] ← 0;
```

- For the previous logical array **A** with character indices:

```
Var A: Array['e'] Of Boolean;
```

The initialization of all elements of **A** to **False** is achieved by:

```
Var i:Character;
For i ← 'a' To 'd' Do
    T[i] ← False;
```

Remarks:

- If the size is expressed by an enumerated type, then the index can only take the values mentioned. Otherwise, the index value must be between 0 and `<size> - 1`.
- In the C language, like other variable types, array elements can be initialized during the declaration of the array. To do this, you place the values of the elements between curly braces (`{` and `}`), separated by commas. In this case, it is not necessary to indicate the size of the array within square brackets, as it is specified by the number of elements present between the curly braces. Additionally, it is possible to only mention the initial values within the curly braces.

Examples:

- We initialize an array with the first 5 odd numbers as follows:

```
int T[] = {1, 3, 5, 7, 9};
```

- It is also possible to initialize only 3 values out of the 5. The remaining two elements will remain uninitialized:

```
int T[5] = {1, 3, 5};
```

V.2.3.3) Displaying the contents of an Array

To write (display) an array, the process is similar. You just need to replace the reading or assignment statement with the writing statement. Thus, displaying the elements of the array can be done through a series of writing instructions:

```
Write(<name_Array>[0]);
Write(<name_Array>[1]);
.....
Write(<name_Array>[<size>-1]);
```

Where **<name_Array>** is the name of the array, and **<size>** is the number of elements in the array.

Or by using the **For** loop as follows:

```
For <indice> ← 0 To <size>-1 Do
    Write(<name_Array>[<index>]);
```

With **<index>** being a variable (usually of integer type) used as an index to access the elements of the array.

In the C language, displaying the elements of an array is done by:

```
for(<index>=0; <index> < <size>;<index>++)
    printf("<format>",<name_Array>[<index>]);
```

Here, **<format>** is the format for writing the elements of the array ("%d", "%f",...).

Example:

Write an algorithm to input from the keyboard the elements of an array of 100 integers and then display the content of this array:

```
Algorithm example;
Const n=100;
Var t:Array[n] Of Integer;i:Integer;
Begin
For i ← 0 To n-1 Do
    Begin
        Write("Enter the element ",i," : ");
        Read(t[i]);
    End;
Write("The elements of the array: ");
For i ← 0 To n-1 Do
    Write(t[i]);
End.
```

V.3) Multidimensional Arrays

Suppose we now want to input the marks of a class of 5 students but in 4 different courses and store them all in a suitable data structure. Now that we are familiar with arrays, a solution to this problem involves using an array to store the students' marks in each course. As a result, we have four arrays, each with five elements. The declaration of the 4 arrays is as follows:

```
Var Not1,Not2,Not3,Not4:Array[5] Of Real;
```

These arrays can be represented graphically as:

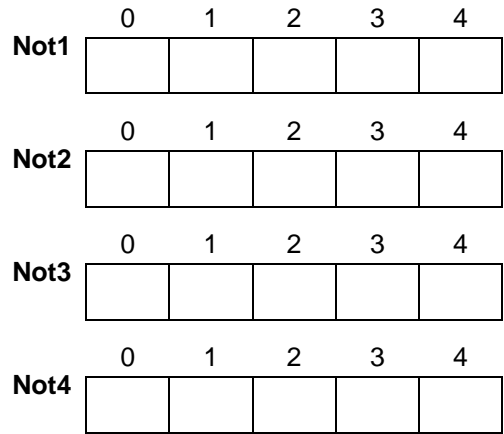


Figure V.4. Diagrams of the 4 mark arrays

Although this solution is correct, it lacks flexibility when manipulating data (input, display, ...) and especially when the number of courses is significant.

It would be better if we could group all this data together in a single data structure.

Fortunately, when we go back to the definition of an array, we find that the elements of an array can be of any type. They can be of **integer**, **real**, **boolean** types, etc., and they can also be of compound types, including the **Array** type.

Therefore, instead of using four arrays, one for each course, it is sufficient to use a single array where each element of this array is, in turn, an array containing the marks of students in a course. The used array has the following structure:

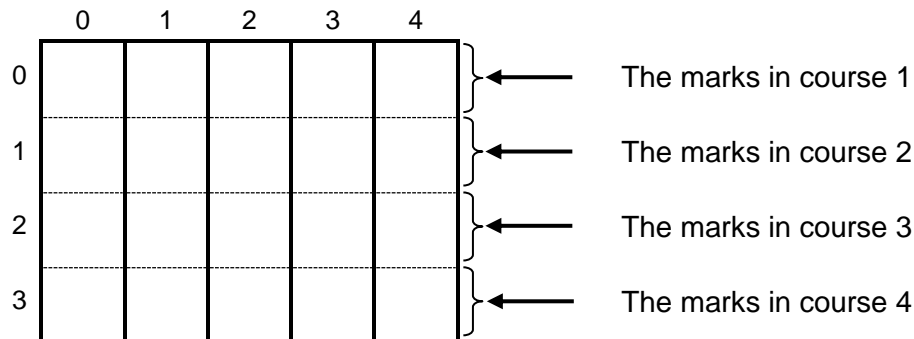


Figure V.5. Example of an array of arrays grouping the marks of all students in all courses

Suppose now that each student has 3 marks in each course (exam mark, tutorial grade, and practical grade) and not just one. In this case, to represent all the marks of all students in all

courses, each element of the previous array must itself be an array of 3 elements. The resulting array can be schematized as follows:

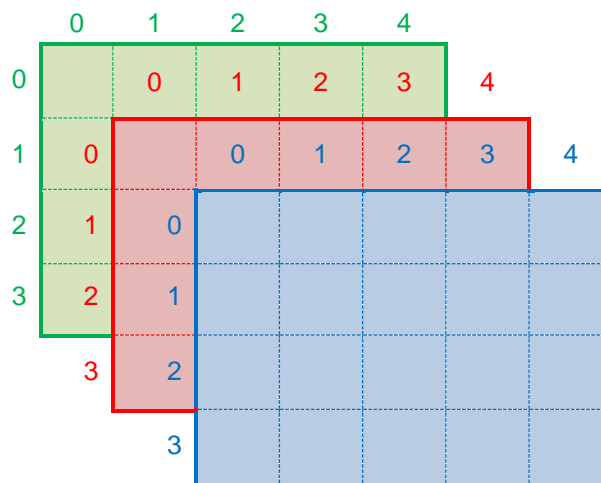


Figure V.6. Diagram of a 3-dimensional array representing all marks of all students in all courses

This type of array is known as a **multidimensional array**.

V.3.1) Definition

The arrays discussed in the first part of the chapter are one-dimensional arrays (single-dimensional), also known as **vectors**. Each element of these arrays holds a single value of a simple type (**integer**, **real**, **character**, **boolean**), and it is identified by a single index.

However, algorithmics and the majority of programming languages, including the C language, provide us with the possibility to declare and use arrays with several dimensions (2 or more), known as **multidimensional arrays**.

Thus, a multidimensional array is an array whose elements can themselves be arrays, which can, in turn, contain other arrays, and so on. Each element of such an array is not identified by a single index as in one-dimensional arrays but by multiple indices, one for each dimension.

For illustrative purposes, examples will be limited to two dimensions; however, generalization to **N** dimensions is straightforward.

When the array is two-dimensional, it is called a **matrix**. The matrix is considered as a grid consisting of rows and columns. Therefore, two indices are needed to indicate the cells or elements of the matrix: the first index represents the rows, and the second index represents the columns.

Matrices are very practical in programming, with a variety of examples of their use. Matrices are useful for presenting the marks of a class of students in multiple courses, modeling a system of linear equations, describing an image formed by a set of pixels, modeling a chess game, and more.

The diagram below represents a matrix of real numbers with 4 rows and 5 columns.

	0	1	2	3	4
0	7	4.5	7.25	4	15.75
1	11.5	16	10	9	16
2	13	8	15	14.5	20
3	9.75	19	12.25	14.25	5

Figure V.7. A Matrix with 4 Rows and 5 Columns

V.3.2) Declaration

The declaration of a multidimensional array is done in the same way as a one-dimensional array, with the exception that here we need to specify the size for each dimension. For example, in the case of a matrix, we need to specify the number of rows and columns.

V.3.2.1) Algorithmic Syntax

In algorithmics, the sizes must be placed within square brackets and separated by commas.

The declaration syntax is as follows:

```
Var <name_Array> : ARRAY[<S1>, <S2>, ...] OF <type_elements>;
```

With:

- **<name_Array>** : being the identifier designating the name of the array.
- **<Si>** : representing the size along dimension **i**.
- **<type_elements>** : indicating the type of the elements in the array.

Example:

The following declaration declares a matrix of real numbers named **M** with 4 rows and 5 columns:

```
Const nbR=4;nbC=5;
Var M : ARRAY[nbR,nbC] OF Real;
```

The rows of this matrix are numbered from 0 to 3, and the columns from 0 to 4.

V.3.2.2) C language Syntax

A la différence avec l'algorithmique, en langage C, chaque taille doit être placée séparément entre crochets.

Ainsi, une variable de type tableau multidimensionnel est déclarée en langage C par:

Unlike in algorithmics, in the C language, each dimension's size must be specified separately within brackets.

Thus, a variable of multidimensional array type is declared in the C language by:

```
<type_elements> <name_Array>[<S1>][<S2>] ... ;
```

Here:

- **<type_elements>** is the type of the elements in the array.

- `<name_array >` is an identifier for the array.
- `<si>` represents the size along the i^{th} dimension.

Example:

The declaration of a matrix of real numbers M with 4 rows and 5 columns is done in the C language by:

```
#define nbR 4
#define nbC 5
float M[nbR][nbC];
```

V.3.3) Manipulation of Multidimensional Arrays

As with one-dimensional arrays, manipulating multidimensional arrays involves accessing their elements, filling, displaying, etc.

V.3.3.1) Accessing elements of a multidimensional array

In a multidimensional array, each element is identified by multiple indices, one for each dimension. Therefore, accessing an element involves specifying all its indices.

Thus, to access an element of the multidimensional array, we include the array name, followed by the indices of the element, within square brackets and separated by commas. The syntax for this access is as follows:

```
<name_Array> [<index1>,<index2>,...]
```

Where `<name_Array >` is the name of the array, and `<indexi>` is the index of the element in dimension i .

In the C language, the access syntax is similar to that in algorithmics, except that each index must be enclosed in separate brackets:

```
<name_Array> [<index1>][<index2>]....
```

Example:

To access the element at the intersection of row number 2 and column number 3 of matrix M , we would write:

- In algorithmics: $M[2,3]$
- In C language: $M[2][3]$

V.3.3.2) Filling a multidimensional array

Manipulating an array requires it to be previously filled. Like a simple array, a multidimensional array can be filled through reading or assignment.

a) Filling by reading

Reading the elements of a multidimensional array is done in a similar way to a one-dimensional array, except that nested loops are used, each corresponding to the index of a dimension. For example, to read the elements of a matrix, you need two loops; the first loop iterates over the rows, and the second loop iterates over the columns.

Filling a multidimensional array of dimension N through reading is done with the following nested loops:

```

For <ind1> ← 0 To <S1>-1 Do
  For <ind2> ← 0 To <S2>-1 Do
    .....
    For <indN> ← 0 To <SN>-1 Do
      Read(<name_Array>[<ind1>,<ind2>,...,<indN>]);

```

With `<name_Array>` being the name of the array, `<indi>` is the index of dimension i , and `<Si>` is the size along dimension i .

And in the C language:

```

for(<ind1>=0; <ind1> < <S1>;<ind1>++)
  for(<ind2>=0; <ind2> < <S2>;<ind2>++)
    .....
    for(<indN>=0; <indN> < <SN>;<indN>++)
      scanf("<format>", &<name_Array>[<ind1>][<ind2>]... [<indN>]);

```

Where `<format>` is the format for reading the elements of the array ("%d", "%f", ...).

Example:

Filling a matrix of real numbers M with 4 rows and 5 columns through reading is done by:

- In algorithmics:

```

For i ← 0 To 3 Do
  For j ← 0 To 4 Do
    Read(M[i,j]);

```

- In C language:

```

for(i=0;i<4;i++)
  for(j=0;j<5;j++)
    scanf("%f", &M[i][j]);

```

Remark:

In the previous example, the "For i" loop encompasses the "For j" loop, which means that for each row i , all columns must be read. In other words, the elements of the matrix are read row by row. Therefore, if you want to read the elements of the matrix column by column, you simply reverse the loops.

b) Filling by assignment

It is possible to assign a value to an element of the multidimensional array individually or to assign the same value to all elements of the array.

When we want to assign a value to a single element of the array, we use the syntax:

```

<name_Array>[<ind1>,<ind2>,...,<indN>] ← <value>;

```

Where `<name_Array>` is the name of the array, `<indi>` is the index of the targeted element along dimension i , and `<value>` is the value to be placed in the element.

Assigning the same value to all elements of a multidimensional array requires traversing all elements of the array, using nested loops as described for reading. Thus, assigning the value $\langle \text{value} \rangle$ to all elements of a multidimensional array of dimension N is done as follows:

```

For  $\langle \text{ind}_1 \rangle \leftarrow 0$  To  $\langle S_1 \rangle - 1$  Do
  For  $\langle \text{ind}_2 \rangle \leftarrow 0$  To  $\langle S_2 \rangle - 1$  Do
    .....
    For  $\langle \text{ind}_N \rangle \leftarrow 0$  To  $\langle S_N \rangle - 1$  Do
       $\langle \text{name\_Array} \rangle [\langle \text{ind}_1 \rangle, \langle \text{ind}_2 \rangle, \dots, \langle \text{ind}_N \rangle] \leftarrow \langle \text{value} \rangle;$ 

```

With $\langle \text{name_Array} \rangle$ is the name of the array, $\langle \text{ind}_i \rangle$ is the index of dimension i , and $\langle S_i \rangle$ is the size along dimension i .

In C language:

```

for( $\langle \text{ind}_1 \rangle = 0$ ;  $\langle \text{ind}_1 \rangle < \langle T_1 \rangle$ ;  $\langle \text{ind}_1 \rangle ++$ )
  for( $\langle \text{ind}_2 \rangle = 0$ ;  $\langle \text{ind}_2 \rangle < \langle T_2 \rangle$ ;  $\langle \text{ind}_2 \rangle ++$ )
    .....
    for( $\langle \text{ind}_N \rangle = 0$ ;  $\langle \text{ind}_N \rangle < \langle T_N \rangle$ ;  $\langle \text{ind}_N \rangle ++$ )
       $\langle \text{name\_Array} \rangle [\langle \text{ind}_1 \rangle] [\langle \text{ind}_2 \rangle] \dots [\langle \text{ind}_N \rangle] \leftarrow \langle \text{value} \rangle;$ 

```

Examples:

- To store the value 9 in the element at the intersection of the second row and fourth column of the previous matrix M , we write:
 - In algorithmics: $M[1,3] \leftarrow 9;$
 - In C language: $M[1][3] = 9;$
- Let T be a matrix of integers with 10 rows and 7 columns. The initialization of all elements of T to 3 is done as follows:
 - In algorithmics:

```

For  $i \leftarrow 0$  To 9 Do
  For  $j \leftarrow 0$  To 6 Do
     $T[i,j] \leftarrow 3;$ 

```

- In C language:

```

for( $i=0$ ;  $i<10$ ;  $i++$ )
  for( $j=0$ ;  $j<7$ ;  $j++$ )
     $T[i][j] = 3;$ 

```

Remark:

In the C language, just like with a simple array, it is possible to initialize the elements of a multidimensional array during declaration. To do this, we place the values of each dimension between curly braces, separated by commas, and enclose the entire set within global curly braces.

Here, it is not necessary to specify the first dimension. However, the others are required.

As with simple arrays, the final values can be omitted.

Examples:

- The following statement declares an integer matrix with 3 rows and 4 columns and initializes its elements with values from 1 to 12:

```
int T[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}}
```

- In this statement, only a portion of the elements is initialized:

```
int T[3][4]={{1,2},{3,4,5}};
```

V.3.3.3) Displaying the elements of a multidimensional array

The display of elements in a multidimensional array is done in the same way as reading them: using nested loops. Thus, displaying the array **A** of **N** dimensions (with respective sizes: **S₁**, **S₂**, ..., **S_N**) is done through the following nested loops:

```
For <ind1> ← 0 To <S1>-1 Do
  For <ind2> ← 0 To <S2>-1 Do
    .....
    For <indN> ← 0 To <SN>-1 Do
      Write(<name_Array>[<ind1>,<ind2>,...,<indN>]);
```

And in C language by:

```
for(<ind1>=0; <ind1> < <S1>;<ind1>++)
  for(<ind2>=0; <ind2> < <S2>;<ind2>++)
    .....
    for(<indN>=0; <indN> < <SN>;<indN>++)
      printf("<format>",<name_Array>[<ind1>][<ind2>]... [<indN>]);
```

Where **<format>** is the format for displaying the elements of the array ("%d", "%f",...).

Example:

Displaying the elements of a matrix of reals **M** with a size of 4x5 is done as follows:

- In algorithmics:

```
For i ← 0 To 3 Do
  For j ← 0 To 4 Do
    Write(M[i,j]);
```

- In C language:

```
for(i=0;i<4;i++)
  for(j=0;j<5;j++)
    printf("%f",M[i][j]);
```

V.4) Strings of Characters

We mentioned in Chapter 2 that in algorithmics, data can take various types, including the **string** type. However, the latter is not considered a fundamental type by several programming languages but rather a composite type. The motivation behind this is that a variable of this type doesn't hold a single value, as is the case for fundamental types

(**integer**, **real**, etc.). Instead, variables of such type are intended to receive sequences of characters that can evolve, both in content and length, as the algorithm progresses.

Strings of characters are very useful in programming. They are effectively used for storing a large amount of non-numeric information, such as people's names, book titles, company addresses, transmitted messages, etc.

As the character is the basic unit of information in a string of characters, we start this section with a brief overview of characters to pave the way for the study of character strings.

V.4.1) Reminder about Characters

V.4.1.1) Definition

A character is a basic textual data. It is used to represent any symbol that may appear in a text. A character can be:

- An alphabetic character (letter): and a distinction is made between uppercase and lowercase. Thus 'B' is different from 'b'.
- A numeric character (digit): '0', '1', ..., '9'
- A special character: such as '!', '*', '%', '&', '<', ...
- A control character: Space, Escape, Carriage return, ...

V.4.1.2) Presentation of Characters

To be represented and manipulated in the computer's memory, each character is defined by a numerical code corresponding to its unique order number in a character encoding system. This system aims to standardize the representation of characters and communication between computers. There are indeed several encoding standards, with the most famous being the ASCII (*American Standard Code for Information Interchange*).

The ASCII code (or ASCII table) is based on a fairly simple principle where each character has a numerical code to be stored and interpreted by a computer.

In its initial version, the ASCII code represents characters on 7 bits (meaning 128 possible characters, from 0 to 127). Later, it was extended to use 8 bits ($2^8 = 256$ characters) to allow for the encoding of national characters (not only English, such as accented characters like: ù, à, è, é, â, ...etc.) and semi-graphic characters.

The 8-bit ASCII table is depicted in Figure V.8.

Remarks:

- A character constant is represented by a single character enclosed in single quotes (apostrophes): 'A', '*', '9', ' '. This notation allows distinguishing a character constant from both a variable and a numeric constant.
- The apostrophe character is doubled and placed between quotes, resulting in a total of four apostrophes ""'".
- A character is not necessarily printable.

0		24	↑	48	0	72	H	96	`	120	x	144	É	168	¿	192	L	216	±	240	≡
1	⊙	25	↓	49	1	73	I	97	a	121	y	145	æ	169	⌈	193	⌋	217	∓	241	≠
2	⊗	26	→	50	2	74	J	98	b	122	z	146	⦶	170	⌌	194	⌍	218	∖	242	≡
3	♥	27	←	51	3	75	K	99	c	123	{	147	ô	171	½	195	⌎	219	∕	243	≡
4	♦	28	↵	52	4	76	L	100	d	124		148	ö	172	¼	196	⌏	220	∖	244	≡
5	♣	29	↗	53	5	77	M	101	e	125	}	149	ò	173	¾	197	⌐	221	∕	245	≡
6	♠	30	↖	54	6	78	N	102	f	126	~	150	û	174	»	198	⌑	222	∕	246	≡
7		31	↘	55	7	79	O	103	g	127	Δ	151	ù	175	»	199	⌒	223	∕	247	≡
8		32		56	8	80	P	104	h	128	Ç	152	ÿ	176	⌓	200	⌔	224	∕	248	≡
9		33	!	57	9	81	Q	105	i	129	ü	153	ÿ	177	⌔	201	⌕	225	∕	249	≡
10		34	"	58	:	82	R	106	j	130	é	154	Û	178	⌕	202	⌖	226	∕	250	≡
11	♂	35	#	59	:	83	S	107	k	131	â	155	ç	179	⌖	203	⌗	227	∕	251	≡
12	♀	36	\$	60	<	84	T	108	l	132	ä	156	£	180	⌗	204	⌘	228	∕	252	≡
13		37	%	61	=	85	U	109	m	133	à	157	¥	181	⌘	205	⌙	229	∕	253	≡
14]]	38	&	62	>	86	V	110	n	134	ã	158	℞	182	⌙	206	⌚	230	∕	254	≡
15	⌘	39	'	63	?	87	W	111	o	135	ç	159	ƒ	183	⌚	207	⌛	231	∕	255	≡
16	▶	40	(64	@	88	X	112	p	136	ê	160	á	184	⌛	208	⌜	232	∕	255	≡
17	◀	41)	65	A	89	Y	113	q	137	ë	161	í	185	⌜	209	⌝	233	∕	255	≡
18	↑	42	*	66	B	90	Z	114	r	138	è	162	ó	186	⌝	210	⌞	234	∕	255	≡
19	!!	43	+	67	C	91	[115	s	139	ì	163	ú	187	⌞	211	⌟	235	∕	255	≡
20		44	,	68	D	92	\	116	t	140	î	164	ñ	188	⌟	212	⌠	236	∕	255	≡
21	§	45	-	69	E	93]	117	u	141	ï	165	Ñ	189	⌠	213	⌡	237	∕	255	≡
22	■	46	.	70	F	94	^	118	v	142	ÿ	166	ä	190	⌡	214	⌢	238	∕	255	≡
23	↓	47	/	71	G	95	_	119	w	143	ÿ	167	å	191	⌢	215	⌣	239	∕	255	≡

Figure V.8. The 8-bit ASCII table.

V.4.2) Definition of a String of Characters

Formally, a string of characters is a homogeneous data structure used to store, in a single variable, a finite number of character elements.

Thus, a string variable can contain a sequence of zero, one, or more concatenated characters. The characters can be printable or non-printable.

In fact, the `string` type can be considered as an array of characters, but it is enriched by other special operations facilitating its manipulation. These operations depend on the programming language in use.

Remarks:

- String constants must be delimited by two double quotes in algorithmics and in the C language.
- The string that contains no characters is called an *empty string*. It is represented by two concatenated double quotes.
- A character constant can be considered as a string constant of length 1.

Examples:

- "Algorithmic1" is a string containing 12 alphanumeric characters.
- "6453" is a string of characters consisting of 4 numerical characters. It should not be confused with the numerical value 6453.
- "&)+_ \$: %" is a string of characters composed of 8 special characters.
- "" is the empty string.
- " " is a string consisting of a single character: the space character.

V.4.3) Strings in Algorithmics

The string data type is predefined in algorithmics but not in all programming languages. It is designated by the keyword **STRING**.

V.4.3.1) Declaration

The **String** data type defines "strings of characters" variables with a maximum of 255 characters (in the base case). A string may contain fewer characters if specified during its declaration, where the number of characters (ranging from 1 to 255) is placed within square brackets.

The declaration of a string variable follows the following syntax:

```
Var <name_string> : STRING[<length>];
```

With:

- **<name_string>** being the name of the declared variable,
- **<length>** being its length in characters. It's optional.

Remark:

When the length is not specified, the declared string is of maximum length (255 characters).

Examples:

- The following declaration declares a string variable of length 255 characters:

```
Var ch:String;
```

- The following declaration creates a string variable of length 20 characters:

```
Var firstName:String[20];
```

V.4.3.2) Memory Representation

A string of characters occupies a contiguous space in memory. This space is proportional to the declared character string's length. Therefore, each character, encoded in ASCII, occupies one byte in the memory.

In practice, one byte is added to this reserved space, used to store the actual length of the string. This length is the number of characters actually contained in the string and is not the number specified during declaration. If this length is not specified, a default length of 255 characters is applied.

V.4.3.3) Manipulating strings

Strings can be manipulated globally or locally (each character individually).

a) Accessing a character in the string

Accessing a specific character in the string is done in the same way as with arrays. Just specify the name of the string followed by the rank (index, position) of the character in square brackets. The access syntax is as follows:

```
<name_String>[<rank>];
```

Where **<name_String>** is the name of the string, and **<rank>** is the position of the character we want to access.

Remarques:

- It's important to note that in algorithmics, the numbering of characters in a string starts from 1 and not 0, as is the case with arrays. The cell at position 0 is intended to contain the actual length of the string (the number of its effective characters).
- If the effective length is less than the declared length, the remaining space reserved for the string will be filled with **null** characters with ASCII code 0.

Example:

Let the character string **s** declared by:

```
Var S:STRING[5];
```

Suppose this string contains the value "Box". This string can be represented as follows:

	0	1	2	3	4	5
S	3	B	o	x	Null	Null

Figure V.9. Representation of the string "Box" in algorithmics.

As mentioned earlier, character numbering starts from 1 because the zeroth position is reserved to store the actual length of the character string which is 3 in this example. Thus the word "Box" occupies positions numbered 1 through 3. The character string **s** is declared with a size of 5, leaving 2 unused positions. These remaining positions are then filled with the **null** character, indicating the end of the string.

In summary, the character string **s** is represented as follows:

- **s[0]** contains the length (3),
- **s[1]** to **s[3]** contain the characters of "Box",
- **s[4]** and **s[5]** are filled with the null character, marking the end of the string.

b) Reading

Unlike arrays, a string of characters in algorithmics can be read in its entirety with a single reading instruction. It is done as follows:

```
Read(<name_String>);
```

Example:

The keyboard input for the string **firstName** is done using:

```
Read(firstName);
```

c) Writing

Similarly for writing, a string in algorithmics can be displayed using a single writing instruction as follows:

```
Write(<name_String>);
```

It is also possible to display a single character from the string by specifying its rank (position) in brackets in the writing instruction. The syntax is:

```
Write(<name_string[<rank>]>);
```

Where `<rank>` is the position of the character to display.

Example:

The following instruction displays the content of the string variable `firstName`:

```
Write(firstName);
```

To display only the first character of the string variable `firstName`, we write:

```
Write(firstName[1]);
```

d) Assignment

One can assign to any string variable a string expression using the traditional assignment symbol `" ← "`. When the value to be assigned is a string constant, it must be enclosed in double quotation marks. The assignment syntax is as follows:

```
<name_String> ← <expression>;
```

Where `<name_String>` is the name of the string of characters, and `<expression>` is the value being assigned.

We can also assign a value (which must be of character type) to a specific character in a string by specifying its position in square brackets after the name of the string. This assignment follows the following syntax:

```
<name_String>[<rank>] ← <value>;
```

Where `<rank>` is the position of the character to be modified, and `<value>` is the value to be stored, which must be of character type.

Example:

```
firstName ← "Kamel";
firstName[4] ← 'a';
ch ← firstName;
```

The piece of code above assigns the constant `"Kamel"` to the variable `firstName` and then assigns the letter `'a'` to the 4th character of the variable `firstName`. After the assignment, the variable `firstName` will contain the value `"Kama1"`. Finally, it copies the content of the variable `firstName` into the variable `ch`.

Remark:

Assignment is possible between two strings of different lengths, provided that the size of the assigned string is not longer.

e) Operations specific to strings of characters

As we have already mentioned, strings can be considered as arrays of characters on which additional operations and functions can be performed.

However, Algorithmics provides a set of operators and predefined functions specifically designed for manipulating strings of characters.

Here are a few of them:

e.1) Concatenation

Concatenation is the operation used to create a new string by joining together two or more strings. In algorithmics, this is done using the "+" operator.

Example:

Consider the following piece of code:

```
Var s1,s2:String[20];
.....
s1 ← "University";
s2 ← ch1 + " of " + "Guelma";
```

After the execution of the instructions below, the variable `s2` will contain the string **"University of Guelma"**.

e.2) Comparison

It's possible to compare two strings using the standard comparison operators (`=`, `>`, `<`, `≥`, `≤`, `≠`).

Like for characters, the comparison of strings is based on the order of ASCII codes of the characters. The comparison is made between the characters at the same position in the two strings and starts from the first character. If the two characters are equal, we move on to the second character in both strings, and so on.

Examples:

- `"Annaba" < "Guelma"` because the ASCII code of `'A'` is lower than the ASCII code of `'G'`.
- `"Anis" > "Amine"` because the ASCII code of `'n'` is higher than the ASCII code of `'m'`.
- `"497" > "(3p%"` because the ASCII code of `'4'` is higher than the ASCII code of `'('`.

e.3) Calculating effective length:

Algorithmics offers a predefined function to determine the number of characters in a string. This function is called `Length`. The syntax for using this function is as follows:

```
Length(<a_string>);
```

Where `<a_string>` is a string expressed as a literal constant or a variable of type `String`.

Examples:

Consider the following piece of code:

```
Var s:String[20];len1,len2:Integer;
.....
s ← "Algorithmics";
len1 ← Length(s);
len2 ← Length("University of Guelma");
```


After the execution of the previous instructions, the variable `len1` will contain the value 12, which is the length of the string "Algorithmics", and the variable `len2` will contain the value 20, which is the length of the string "University of Guelma".

V.4.4) Strings in C language

In the C language, there is no real data type specifically for strings of characters (like the algorithmic `STRING` type)) since we cannot declare variables of such a type. However, there is a convention for representing strings. This convention involves treating a string as an array of characters and providing specific handling for it.

V.4.4.1) Declaration

The syntax for declaring a string in the C language is the same as declaring a regular array of characters.

Thus, a string variable is declared as follows:

```
char <name_String>[<size>];
```

Here, `<name_string>` is the name of the string, and `<size>` is the maximum number of characters that the string can hold.

Example:

The string `name` of 20 characters can be declared as follows:

```
char name[20];
```

V.4.4.2) Memory Representation

In the C language, a string of characters is represented in memory as a sequence of bytes corresponding to each of its characters, specifically their ASCII codes. This sequence is terminated by an additional byte used to store a special character known as the **null character** or **end of string character**. This null character is a non-printable character denoted `'\0'`, indicating that the string ends at the preceding character.

This means that, in general, a string of `n` characters occupies a memory location of `n+1` bytes.

Example:

For example, the string "Guelma" is represented in an array of size 10 as follows:

0	1	2	3	4	5	6	7	8	9
G	u	e	l	m	a	\0			

Figure V.10. Representation in the C language of the string "Guelma" in an array of size 10.

Remarks:

- To be valid, a string must necessarily end with the end of string character `'\0'`. Otherwise, a runtime error occurs.
- The significance of the end of string character is that it allows storing variable-sized strings in an array of fixed size. However, the memory space allocated for a string (in the form of an array) is static, meaning its size cannot be changed. But its content can be

changed, i.e., modifying the characters it contains. This content always ends with `'\0'`. The remaining cells in the array, if any, remain unused (cells numbered 7, 8, 9 in the previous example).

V.4.4.3) Manipulating strings

a) Accessing a character in the string

Accessing a specific character in a string is done in the same way as accessing an element in an array. Simply specify the string's name followed by the index (position) of the character in square brackets. The access syntax is as follows:

```
<name_string>[<rank>];
```

Where `<name_string>` is the name of the string, and `<rank>` is the position of the character we want to access.

Remark:

Unlike in algorithmics, in the C language, the first character starts at index 0, not 1.

Example:

Given the string of characters:

```
char name[20];
```

`name[0]` refers to the first character in the `name` string.

b) Reading

Reading a string of characters in the C language can be done in several ways.

b.1) Reading using `scanf`

A string of characters can be read like any other variable using the `scanf` function with the format code `"%s"`.

```
scanf("%s", <name_string>);
```

Here, `<name_string >` is the name of the string variable to be read.

Note the absence of the `'&'` symbol in `scanf` when reading a string. This is because a string in C is already an address, so there's no need to obtain its address using `&`.

Example:

Consider the following string:

```
char ch[20];
```

Reading this string is done with:

```
scanf("%s", ch);
```

b.2) Reading using `gets`

By default, `scanf` stops input as soon as it encounters a delimiter character: newline, space, tab, etc. So, if the user enters the text "Hello world" only "Hello" will be placed in the variable `ch`.

Fortunately, the C language has another function specifically for reading a string: the `gets` function. The latter is better suited for reading strings containing spaces or tabs. Its syntax is as follows:

```
gets (<name_String>);
```

Example:

The reading of the previous string variable `ch` can be done using:

```
gets (ch);
```

c) Writing

There are different ways to display a string of characters in the C language.

c.1) Writing using `printf`

For displaying a string in the C language, we can use the `printf` function with the format code `"%s"`.

```
printf ("%s", <name_string>);
```

Here, `<name_string >` is the name of the string variable to be displayed.

Example:

Displaying the string of characters `ch` is done as follows:

```
printf ("%s", ch);
```

c.2) Writing using `puts`

Just like reading, C also has another function specifically designed for writing a string: the `puts` function. Its syntax is as follows:

```
puts (<name_String>);
```

Example:

The display of the previous string of characters `ch` can be done using the following instruction:

```
puts (ch);
```

d) Assignment

In the C language, it's not possible to directly assign a value to a string variable, except during its declaration. The syntax for such an assignment is as follows:

```
char <name_String> [<length>] = <value>;
```

Here, `<value>` is the initial value assigned to the `<name_String >` variable. It's a string of characters constant enclosed in double quotation marks.

Note that you do not need to include a final `'\0'`. This operation is done automatically because the quotes indicate unambiguously that it is a string.

On the other hand, we can assign a value to a specific character in a string by specifying its position in brackets after the name of the string. This assignment follows the following syntax:

```
<name_String> [<rank>] ← <value>;
```

With `<rank>` representing the position of the character to be modified, and `<value>` representing the value to be stored, which must be of character type.

Example:

- We can declare the 30-character string `course` and initialize it with the constant `"Algebra"` in a single line as follows:

```
char course[30] = "Algebra";
```

- To assign the letter `'v'` to the 5th character of the `course` variable, we write:

```
course[4] ← 'v';
```

e) Operations specific to strings of characters

The C language provides a variety of functions for manipulating strings of characters. These functions are defined in the `<string.h>` library (header file). Here are some of these functions:

e.1) The `strcpy` function

This function copies one string into another. It is equivalent to an assignment for strings. The prototype of the `strcpy` function is as follows:

```
strcpy(<string1>, <string2>);
```

It copies the content of `<string2>` into `<string1>` and returns the latter as the result.

Example:

```
char s[10]="Hello";
char t[10];
strcpy(t,s);
```

The piece of code above copies the content of the string `s`, which is the value `"Bonjour"` into the string `t`.

e.2) The `strlen` function

This function returns the effective length (number of characters) of a string. The syntax for using this function is:

```
strlen(<a_string>);
```

Example:

```
int nb=strlen("Hello");
```

This instruction assigns the length of the string `"Hello"`, which is 5, to the integer variable `nb`.

e.3) The `strcat` function

The `strcat` function concatenates two strings, meaning it appends the characters of the second string to the end of the first string. The prototype is:

```
strcat(<string1>, <string2>);
```

The function copies the contents of `<string2>` to the end of `<string1>`, and the result is stored in `<string1>`.

Example:

```
char s[30]="Hello";
char t[10]=" world";
strcat(s,t);
```

A la fin de ces instructions, la variable `s` va contenir la chaîne "Hello world", qui est le résultat de concaténation des deux chaînes contenues dans `s` et `t`.

At the end of these instructions, the variable `s` will contain the string "Hello world", which is the result of concatenating the two strings stored in `s` and `t`.

e.4) The strcmp function

The `strcmp` function is used to compare two strings for alphabetical order. The prototype is:

```
strcmp(<string1>,<string2>);
```

Thus, this function returns a negative value when `<string1>` is less than `<string2>`, a positive value if `<string1>` is greater than `<string2>`, and returns 0 when the two strings are the same.

Example:

```
char s[30]="Guelma";
char t[10]="Annaba";
int v=strcmp(s,t);
```

The value contained in the variable `v` will be a positive value indicating that the string "Guelma" is greater than the string "Annaba" in alphabetical order.

V.5) Exercises**Exercise V.1 :**

Write an algorithm to input an array of 7 real elements from the keyboard and calculate the average of the strictly positive elements in the array.

Exercise V.2 :

Write an algorithm allows to fill an array of N integer elements from the keyboard, reverse it, and display the reversed array.

Exercise V.3 :

Write an algorithm for entering the elements of an array of n integer elements and searches for the existence of a value v provided by the user in this array. If the value exists, the algorithm should display the index of its first occurrence and the index of its last occurrence in the array.

Exercise V.4 :

The Pythagorean table is a two-dimensional array in which each cell contains the result of an operation. In the Pythagorean addition table, for example, each intersection (i, j) contains the result of adding i and j . In the Pythagorean multiplication table, each intersection (i, j) contains the result of multiplying i by j , and so on.

We want to create a modified version of the Pythagorean table. In this table, each cell (i, j) will contain the result of adding i and j when the sum of the indices is even and the result of multiplying i by j when the sum of the indices is odd.

Write the algorithm for creating this modified Pythagorean table.

Exercise V.5 :

Write an algorithm to determine the number of lowercase vowels present in a text (assumed not to exceed 132 characters) provided by the user.

V.6) Solution of the exercises

Exercise V.1 :

```

Algorithm calculation;
Const n = 7;
Var T: Array[n] of Real; i, nb: Integer; s, avg: Real;
Begin
  For i ← 0 to n-1 Do
    Read(T[i]);
  s ← 0; nb ← 0;
  For i ← 0 to n-1 Do
    If T[i] > 0 Then
      Begin
        s ← s + T[i];
        nb ← nb + 1;
      End;
  If nb = 0 Then avg ← 0
  Else avg ← s / nb;
  Write("The average is ", avg);
End.

```

Exercise V.2 :

```

Algorithm reversal;
Const n = 7;
Var T: Array[n] of Integer; i, x: Integer;
Begin
  For i ← 0 to n-1 Do
    Read(T[i]);
  For i ← 0 to (n-1) div 2 Do
    Begin
      x ← T[i];
      T[i] ← T[n-1-i];
      T[n-1-i] ← x;
    End;
  Write("The array after reversal:");
  For i ← 0 to n-1 Do
    Write(T[i]);
  End.

```

Exercise V.3 :

```

Algorithm search;
Const n = 10;
Var T: Array[n] of Integer;
i, first, last, v: Integer; found: Boolean;
Begin
  For i ← 0 to n-1 Do
    Read(T[i]);
  Write("What value are you looking for: ");
  Read(v);
  found ← False;
  For i ← 0 to n-1 Do
    Begin
      If T[i] = v Then
        Begin
          last ← i;
          If found = False Then
            Begin
              first ← i;
              found ← True;
            End;
          End;
        End;
      End;
    End;
  If found = False Then Write(v, " does not exist in the array")
  Else Begin
    Write("The first occurrence is at index ", first);
    Write("The last occurrence is at index ", last);
  End;
End.

```

Exercise V.4 :

```

Algorithm Pythagorean_Table;
Const n = 10;
Var M: Array[n,n] of Integer; i, j: Integer;
Begin
  For i ← 0 to n-1 Do
    For j ← 0 to n-1 Do
      If (i + j) mod 2 = 0 Then
        M[i, j] ← (i + 1) + (j + 1)
      Else
        M[i, j] ← (i + 1) * (j + 1);
  Write("The modified Pythagorean table:");
  For i ← 0 to n-1 Do
    For j ← 0 to n-1 Do
      Write(M[i, j]);
  End.

```

Exercise V.5 :

```
Algorithm vowels;
Var text:String[132];nbVowels,i:Integer;
Begin
Write("Please enter the text:");
Read(text);
nbVowels ← 0;
For i ← 1 To Length(text) Do
  Begin
    If text[i]='a' or text[i]='e' or text[i]='i' or
      text[i]='o' or text[i]='u' or text[i]='y' Then
      nbVowels ← nbVowels +1;
  End;
Write("The number of vowels is ", nbVowels);
End.
```

V.7) Conclusion

The challenges encountered in real-world problem-solving often involve managing extensive datasets. To address this, an efficient method for representing information in main memory becomes crucial. The conventional approach, assigning a primitive variable to each piece of information, reveals its limitations when dealing with large datasets. Consequently, the concept of aggregating multiple pieces of information under a unified identifier emerges as an ideal solution. It is from this motivation that different data structures have been proposed.

In this chapter, we introduced the primary data structure enabling the consolidation of values of the same type into a singular entity : the array. Within an array, values are systematically organized in cells, each uniquely identified by a numerical index. This allows for individual manipulation of each element, fostering adaptability in application. Throughout the chapter, we explored various aspects of the array type, including multidimensional arrays and character strings. We delved into the intricacies of declaration and manipulation for each of these data structures, providing comprehensive insights in both algorithmic and C language contexts.

Chapter VI. Custom Types

VI.1) Introduction

The algorithms written so far manipulate data of different types. These data types can be simple, such as `integer`, `real`, `character`, and `boolean` types, or structured, as in the case of arrays and strings.

The main common characteristic of all the mentioned data types is that they are predefined in algorithmics and in most programming languages. In other words, they are known directly by compilers without needing explicit definitions. Another characteristic of predefined types is that they are homogeneous, meaning the data contained in a variable of such a type is of the same nature.

However, predefined types do not allow us to describe all kinds of real-world information. They do not, for example, enable us to group information of different types related to the same object into a single structure. For instance, can we describe a student, a product, or a vehicle using a simple predefined type such as an `integer`, a `string`, or an `array`? The answer is no.

Nevertheless, algorithmics and most programming languages offer algorithm and program designers the possibility to define new data types, known as ***user-defined types*** or ***custom types***. These types allow the representation of data structures composed of several elements of standard types.

The definition and manipulation of user-defined types are the subject of this chapter. Specifically, in this chapter, we focus primarily on two types: ***records*** and ***enumerations***. Other types will be briefly described with fewer details.

VI.2) Concept of Data Type

VI.2.1) Definition

A data type defines the nature of the values a piece of data can take, as well as the operators that can be applied to it.

Examples:

- The `Integer` type consists of the set of positive and negative integer values, along with the operators applicable to such values, namely the arithmetic operators (`+`, `-`, `*`, `/`, `div`, `mod`), and the relational operators (`<`, `>`, `=`, `≠`, `≤`, `≥`).
- The `Boolean` type is defined by the set of logical values (`True` and `False`) and the operators applicable to these values, namely the logical operators (`And`, `Or`, `Not`).

VI.2.2) Type declaration

In algorithmics and in the C programming language, it is possible to use new types in addition to the predefined types. However, to be able to use a new type, it is necessary to declare it first in the declaration section. Thus, the declaration of a new type defined by the user in algorithmics (respectively in the C language) is done using the keyword **TYPE** (respectively **typedef**). By convention, the declaration of types will be done after the declaration of constants and before the declaration of variables.

Indeed, each category of types has a special declaration style. We will present in the following of the chapter the details of the declaration of each custom type.

Examples:

1. The following declaration defines a new custom type called **number** whose values are those of the predefined **integer** type. The new type **number** can then be used in declarations in the same way the **integer** (**int**) type can be used.

In algorithmics:

```
TYPE number = integer;
```

In the C language:

```
typedef int number;
```

2. Declaration of a custom type called **Tab**, which is an array of 10 real numbers:

In algorithmics:

```
Const n=10;
TYPE Tab = Array[n]of real;
```

In the C language:

```
#define n 10
typedef float Tab[n];
```

VI.3) Enumerations

VI.3.1) Definition

Enumerations, or **enumerated types**, are the first custom type discussed in this chapter. Enumerations are defined by algorithm designers and programmers and are not directly recognized by compilers.

Thus, an **enumerated type** is a type for which the designer explicitly lists an ordered set of possible values that a variable of this type can take in an enumeration. These values are explicitly defined and specified by identifiers (constants). The order of the values is the order in which the identifiers were enumerated.

VI.3.2) Enumerations in algorithmics

VI.3.2.1) Declaration

The enumerated type to be defined must be declared in the declaration part of the algorithm using the keyword **TYPE**. The declaration of an enumerated type with **n** values follows the following syntax:

```
TYPE <name_Type> = (<val0>, <val1>, . . . . , <valn-1>);
```

Where <name_Type> represents the name of the defined enumerated type, and <val₀> ... <val_{n-1}> are the values of this type.

After defining the enumerated type, we proceed to declare variables of this type as follows:

```
VAR <name_Variable> : <name_Type>;
```

Here <name_Variable> is the name of the variable, and <name_Type> is the name of the defined enumerated type.

Examples:

The following declarations allow to define three enumerated types (**Season**, **Color**, and **Day**), specifying their respective values, and declare variables of these types:

```
TYPE Season = (Spring, Winter, Autumn, Summer);
   Color = (Red, Green, Blue, Yellow, White, Black);
   Day = (Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday);
VAR s: Season; c1, c2: Color; d: Day;
```

Thus:

- The variable **s** can only take one of the four values: **Spring**, **Winter**, **Autumn**, **Summer**.
- The variables **c1** and **c2** can only take one of the values: **Red**, ..., **Black**.
- The variable **j** can only have one of the values: **Saturday**, ..., **Friday**.

Remarks:

- The constants of an enumeration are related by an order defined by the position of the values in the enumeration. Therefore, the order in which the identifiers are listed is significant. For example, **Summer** > **Winter**, and **Saturday** < **Sunday**.
- The rank of an enumerated constant is determined by its position in the list of identifiers. In this case, the first value has a rank of 0.
- The names assigned to the different constants (values) of an enumeration cannot be reused. For example, the following declaration is not allowed: **VAR green: Integer;**

VI.3.2.2) Manipulation

A variable of an enumerated type can only have values specified during the declaration of the type. Additionally, such a variable cannot be read or written using the **Read** or **Write** instructions, but it can be manipulated in various other ways.

a) Assignment

We can assign a value to a variable of an enumerated type using the usual assignment operator (**←**). The assignment follows the following syntax:

```
<name_Variable> ← <value>;
```

Where: <name_Variable> is the name of the enumerated variable, and <value> is the value to be placed in the variable.

The value to be assigned must be one of the constants listed during the declaration of the type. This constant can be indicated directly or be contained in another variable of the same type.

Example:

```
TYPE Season = (Spring, Winter, Autumn, Summer);
      Color = (Red, Green, Blue, Yellow, White, Black);
VAR s:Season;c1,c2:Color;
.....
s ← Winter;
c1 ← Yellow;
c2 ← c1;
```

b) Predefined Functions

Algorithmics provides certain predefined functions for the manipulation of enumerated types. We list below the three main functions:

- The **Ord** function: returns the rank (order) of a value, i.e., its position in the list of values specified in the declaration.
- The **Pred** function: returns the value that immediately precedes the current value in the enumeration. The predecessor of the first value is not defined.
- The **Succ** function: provides the value that immediately follows the current value in the enumeration. The successor of the last value is not defined.

Examples:

```
Ord(Autumn)      returns 2.
Pred(White)     returns the value Yellow.
Succ(White)     returns the value Black.
```

c) Using

An enumerated type variable can be used in a test, in a loop, in a **Case** statement, etc.

Examples:

```
TYPE Day = (Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday);
VAR d:Day;
```

The following instructions are allowed:

```
1) If d = Friday Then Write("Rest");
2) For d ← Saturday To Friday Do
   Write(d);
3) Case d of
   Saturday: Write("Rest");
   Sunday:   Write("Work");
   .....
End;
```

VI.3.3) Enumerations in the C language

In the C language, an enumeration type, although it constitutes a user-defined type, is considered a specific case of an integer type and, therefore, a scalar (or simple) type.

VI.3.3.1) Declaration

An enumeration (enumerated type) is defined in the C language using the `enum` keyword. In this section, two methods of declaring an enumerated type are presented.

a) Declaration of an Enumeration

The syntax for declaring an enumeration (or a template for the enumerated type) in the C language is as follows:

```
enum <name_Enumeration> {<val0>, <val1>, ..., <valn-1>};
```

This declaration defines an enumeration named `<name_Enumeration>` and specifies that it has n possible values designated by the identifiers `<val0>, ..., <valn-1>`. These values constitute the constants of the enumeration.

Subsequently, the declaration of variables of the defined enumerated template is done as follows:

```
enum <name_Enumeration> <name_Variable>;
```

Example:

Consider the following declarations:

Enumeration Declarations:

```
enum Season {Spring, Winter, Autumn, Summer};
enum Color {Red, Green, Blue, Yellow, White, Black};
enum Natural {Four, Five, Six, Seven};
```

Variable Declarations:

```
enum Season s;
enum Color c1, c2;
enum Natural n;
```

The lines above define three enumerations (**Season**, **Color**, and **Natural**), specifying their respective values, and declare variables of these models.

Remarks:

- Note the presence of the `enum` keyword in each declaration of enumeration variable.
- Enumerated constants: `<val0>, ..., <valn-1>` are encoded by integers $0, 1, \dots, n-1$. Thus, the declaration: `enum Natural {Four, Five, Six, Seven}` simply associates an `int` value with each of the four mentioned identifiers. Specifically, it assigns the value 0 to the first identifier **Four**, the value 1 to the identifier **Five**, and so on.
- It is possible to modify the integer value (code) associated with an enumerated constant during model declaration by adding the equal sign (`=`) and the new value after the constant identifier.

Example:

```
enum Natural {Four=4, Five, Six, Seven};
```

Here, the values of the enumerated constants start from the value 4. Thus, **Four** = 4, **Five** = 5, **Six** = 6, and **Seven** = 7.

b) Declaration using the `typedef` keyword

In fact, the previous declaration lacks flexibility. As mentioned earlier, with the previous declaration, we are obliged to write the `enum` keyword every time we want to declare a variable of the enumerated type.

To simplify the declaration of enumerated variables, there is another, more convenient way of declaring. This involves using the `typedef` keyword to define what is called in the C language a *shortcut* or a *type synonym*. Thus, using this keyword at the time of defining the enumeration will give a new name to this type.

In this way, it is sufficient to use the name of the defined type without the need to precede it with `enum`. Thus, the syntax of this declaration is as follows:

```
typedef enum {<val0>, <val1>, ..., <valn-1>} <name_Type>;
```

Where `<Name_Type>` is the name of the enumerated type and `<val0>` ... `<valn-1>` are the constants of this type.

After that, to declare a variable of the previous type, the syntax used is:

```
<name_Type> <name_Variable>;
```

Where `<name_Type>` is the name of the defined enumerated type, and `<name_Variable>` is the name of the variable.

Example:

The following two lines allow defining an enumerated type `Color` using the `typedef` keyword and then declaring two variables, `c1` and `c2`, of this type:

```
typedef enum {Red, Green, Blue, Yellow, White, Black} Color;
Color c1, c2;
```

VI.3.3.2) Manipulation**a) Reading and writing**

A variable of an enumerated type is read and written like a variable of `integer` type, using the format code `%d`. This is logical since the values of an enumeration are encoded as integer values. Noting that it is important to validate the read values to avoid erroneous results.

Example:

The following program reads the value of a variable of the enumerated type `Color`, ensures that the entered value is valid, and then displays it. Reading and writing are performed in the form of integer numbers.

```
int main() {
    typedef enum {Red,Green,Blue,Yellow,White,Black} Color;
    Color c;
    scanf("%d",&c);
    if(c<Red || c>Black)printf("Input error");
    else printf("%d",c);
    return 0;
}
```

b) Assignment

Assigning a value to an enumerated variable is done using the usual assignment operator (=). The value to be assigned must be one of the constants specified during declaration. This constant can be written explicitly, expressed by the associated integer value, or contained in another variable of the same type.

The assignment follows the syntax:

```
<name_variable> = <value>;
```

Where : <name_variable> is the name of the enumerated variable, and <value> is the value to be placed in the variable.

Example:

Consider the following declarations:

```
enum Season {Spring, Winter, Autumn, Summer};
enum Season s1,s2,s3;
```

The instructions below assign the constant **Autumn** to the three enumerated variables **s1**, **s2**, et **s3**:

```
s1 = Autumn; //assigns the constant Autumn to the variable s1
s2 = 2;      //assigns the constant with the value 2, which is Autumn to variable s2
s3 = s1;     //assigns the value contained in s1, which is Autumn the to variable s3
```

Remark:

In the C language, there is no control of the value assigned to an enumerated variable. It is possible to assign any integer value to a variable of an enumerated type. The control is left to the responsibility of the programmers.

Example:

Consider the variable **c1** of the previous **Color** type. The following instruction:

```
c1 = 20;
```

Is accepted although the value 20 does not belong to the **Color** type.

c) Using

As in algorithmics, an enumerated-type variable in the C language can be used in a condition, in a loop, in a **Case** statement, etc.

Examples:

```
typedef enum {Saturday,Sunday,Monday,Tuesday,Wednesday,Thursday,Friday} Day;
```

```
Day d;
```

The following instructions are accepted:

```
1)   if(d == Friday)printf("Rest");
2)   for(d=Saturday;d<=Friday;d++)
      printf("%d ",d);
3)   switch(d) {
      case Saturday: printf("Rest");break;
      case Sunday:   printf("Work");break;
      .....
    }
```

VI.4) Records (structures)

We have seen in the previous chapter that arrays allow us to group several elements of the same type under the same name, each of them being identified by its index in the array. However, in practice, we may also want to group within the same structure information that relates to the same entity but does not necessarily have the same type.

Take an example. Suppose we want to store in memory all the information related to a student (student ID, last name, first name, date of birth, address, marks, etc.). Designating a variable for each piece of information doesn't seem like the ideal solution, and the resulting algorithm would be difficult to manage. Similarly, grouping all this information in an array is not possible because the information has different types (student ID: **Integer**; first name, last name, and address: **String**; marks: **Real**, etc.).

To overcome this problem, algorithmics and programming languages have introduced new data structures called **Records**, which are better suited for the representation of this type of information.

Records are indeed very useful in programming for representing real-world entities. They allow, among other things, the representation of:

- Dates formed by three values: day, month, and year.
- Complex numbers composed of a real part and an imaginary part.
- Company employees defined by: a social security number, last name, first name, address, position, grade, etc.
- Vehicles described by: a license plate, brand, color, power, category, number of seats, etc.
- Bibliographic records (ISBN code, book title, author, publisher, publication date, etc.)
- Products in a store (code, product name, unit price, available quantity, etc.)
- Bank clients (number, name, account number, phone number, etc.).

VI.4.1) Definition

A *record* also known as a *structure*, is a data structure that allows gathering within a single entity a set of data of the same type or different types associated with a single object.

The record is composed of a set of elements called **fields**, where each field corresponds to a piece of data. Similar to the cells of an array, the fields of a record can be accessed individually for reading, writing, or manipulation.

VI.4.2) Records in algorithmics

VI.4.2.1) Declaration

Before declaring a record variable, its type must first be defined in the declaration section of the algorithm using the keyword **TYPE**.

To define a record type, you must first choose a name for it, then list all the fields you want to store inside this type. Each field is identified by a name, which allows direct access, and a type. This type can be any simple (**integer**, **character**, etc.) or structured (**array**, **string**, **enumeration**, etc.) type.

The general form of declaring a record type is as follows:

```

TYPE <name_Type > = RECORD
    Begin
        <name_Field1>: <type_Field1>;
        <name_Field2>: <type_Field2>;
        .....
        <name_Fieldn>: <type_Fieldn>;
    End;

```

Where:

- **<name_Type>**: is the identifier that designates the name of the defined record type.
- **<name_Field_i>**: is the name of the i^{th} field of the record.
- **<type_Field_i>**: is the type associated with the i^{th} field.

Once the type is defined, we can declare variables of this type as we normally would. The syntax for this declaration is as follows:

```

VAR <name_Variable>: <name_Type>;

```

Where **<name_Variable>** is the name of the variable, and **<name_Type>** is the name of the defined record type.

Examples:

The following declarations allow the definition of two record types and the declaration of variables of these types:

- 1) The **Date** type is used to describe a real-world date, and it consists of three fields: **day**, **month**, and **year**, each of type **Integer**.

```

TYPE Date = RECORD
    Begin
        Day: integer;
        Month: integer;
        Year: integer;
    End;
VAR D: Date;

```

- 2) The **Etudiant** type represents a university student. It is defined by his identification number (ID), last name, first name, age, and his marks in 9 courses. All this information constitutes the fields of the record. Here, the marks are grouped in an array of real numbers.

```

TYPE Student = RECORD
    Begin
        ID, Age: integer;
        LastName, FirstName: String[20];
        Marks: Array[1..9] of real;
    End;
VAR Stud: Student;

```

Remarks:

- It is not possible to declare a constant of record type.
- A record can be schematized as a set of cells of different sizes because the types of the fields in a record are not necessarily the same, unlike an array.

Example:

The record **D** of type **Date** mentioned earlier can be schematized as follows:

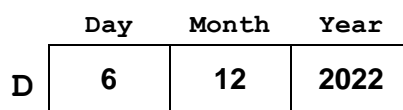


Figure VI.1. Diagram of a record of type **Date**.

VI.4.2.2) Manipulating a record

In fact, the only possible instruction to manipulate a variable of record type (in its entirety) without accessing its fields is assignment.

However, the fields of a record can be manipulated individually like any other variable of a similar type. They can be read, written, assigned values, and used in conditions, loops, etc.

a) Accessing a Field of a Record

While the elements of an array are accessible through their index (their number in the array), the fields of a record are accessible by their names.

Thus, we can access a field of the record by specifying the record's name followed by the field's name, both separated by the dot operator (**.**). The syntax for access is as follows:

```
<name_Record>.<name_Field>
```

Where **<name_Record>** is the name of the record, and **<name_Field>** is the name of the field we want to access.

Examples:

- To access the **Day** field of the record **D**, we write: **D.Day**

- To access the **Age** field of the record **Stud**, we write: **Stud.Age**

Remark:

Since access to fields is done by their name, the order of declaring these fields is not important.

b) Reading and writing

Just like with arrays, it's not possible to read or write an entire record globally. Therefore, to read a record, it's necessary to read each of its fields one by one. The same goes for displaying; all the fields of the record must be displayed one by one.

The syntax for reading a field is as follows:

```
Read(<name_Record>.<name_Field>);
```

The syntax for writing a field is as follows:

```
Write(<name_Record>.<name_Field>);
```

Example:

Let **D** be a variable of the previous type **Date**:

```
Var D: Date;
```

Reading all the fields of the record **D** is done by the following 3 read instructions:

```
Read(D.Day);
Read(D.Month);
Read(D.Year);
```

Or using one single reading instruction:

```
Read(D.Day, D.Month, D.Year);
```

Similarly, displaying the record **D** is done by displaying all its fields:

```
Write(D.Day);
Write(D.Month);
Write(D.Year);
```

Or by:

```
Write(D.Day, D.Month, D.Year);
```

Remark:

It should be noted that, unlike arrays, it is not possible to use a loop to read or write all the fields of a record. This is because the fields are labeled with names and are not numbered like the elements in arrays.

However, this is not a problem, as in practice, the number of fields is very limited (5 to 20 fields).

c) Assignment

As mentioned earlier, assignment is possible between two record variables of the same type. Thus, the following form of instruction is accepted:

```
<name_Record1> ← <name_Record2>;
```

This instruction implies that all the fields of the record `<name_Record2>` are copied to the corresponding fields of the record `<name_record1>`.

We can also assign values to individual fields. To assign a value to a specific field of the record, we use the syntax:

```
<name_Record>.<name_Field> ← <value>;
```

With `<name_Record>` being the name of the record, `<name_Field>` being the name of the field to which we want to assign the value, and `<value>` being the value to put in the field.

The value and the field must be of the same type.

Example:

Let `D1` and `D2` be two variables of type `Date`:

```
Var D1,D2: Date;
```

And let the following instructions be given:

```
D1.Day ← 6;
D1.Month ← 12;
D1.Year ← 2022;
D2 ← D1;
```

These instructions assign a value to each field of the record `D1` individually and then copy each of these values to the corresponding fields in the record `D2`.

d) The *WITH...DO* statement

The repetition of the record variable name to manipulate each of the record's fields is tedious. To simplify access to the fields of a record, we can use a special instruction: the **WITH** statement.

Inside the **WITH** statement, we can directly manipulate the fields of the record without adding the record name and the dot.

Thus, instructions of the following form:

```
Read(<name_Record>.<name_Field1>);
<name_Record>.<name_Field2> ← <value>;
Write(<name_Record>.<name_Field3>);
```

Can be replaced, using the **WITH** structure, by:

```
WITH <name_Record> DO
  Begin
    Read(<name_Field1>);
    <name_Field2> ← <value>;
    Write(<name_Field3>);
  End;
```

Where `<name_Record>` is the name of the record, and `<name_Fieldi>` is the name of the i^{th} manipulated field.

Example:

Let the record `D` of type `Date` be as follows:

```
Var D: Date;
```

An example of accessing the fields of the record `D`, both without and with the `WITH` structure, is as follows:

Without the `WITH` structure

```
Read(D.Day);
D.Month ← 7;
D.Year ← 1992;
Write(D.Day, D.Month, D.Year);
```



With the `WITH` structure

```
WITH D DO
  Begin
    Read(Day);
    Month ← 7;
    Year ← 1992;
  Write(Day, Month, Year);
  End;
```

VI.4.2.3) Nesting of Records

Suppose that in the previously defined `Student` type, we no longer want the student's age but their date of birth. The date of birth consists of three inseparable values (day, month, year). Therefore, a date corresponds to a real-world entity that must be represented by a record type with 3 fields. Since we have already defined the `Date` type, we can use it in the declaration of the `Student` type for the date of birth.

In this case, we have a field in a record that is itself a record. This is referred to as record nesting.

The notation used to access fields remains the same (using the dot) whenever we want to delve deeper into the structure.

Example:

```
TYPE Date = RECORD
  Begin
    Day, Month, Year: integer;
  End;
Student = RECORD
  Begin
    ID: integer;
    LastName, FirstName: String[20];
    Date_Birth: Date;
    Marks: Array[1..9] of real;
  End;
VAR Stud: Student;
```

To access the birth month of the student `Stud`, you need to use the dot operator `"."` twice:

```
Stud.Date_Birth.Month
```

VI.4.2.4) Arrays of records

Records are used to represent within an algorithm or computer program a real or abstract entity. However, it often happens that we want to deal with not just one entity but several. For example, we might want to represent a group of students, a set of dates, a list of products in a store, etc.

Let's take the example of a group of students; instead of creating several variables of the `Student` record type, we can create an array that groups all the students in the group. It becomes an array of records.

a) Declaration

When declaring an array type, we must specify the type of its elements. This type must be known before the declaration of the array. Therefore, the declaration of an array of records must be preceded by the definition of the type of records that constitute it.

Finally, we declare a variable of the defined array type.

Example:

The declaration of an array of 30 students, each defined by his ID number, last name, first name, age, and marks, is performed as follows:

```
CONST n = 30;
TYPE Student = RECORD
    Begin
        ID, Age: integer;
        LastName, FirstName: String[20];
        Marks: Array[1..9] of real;
    End;
    Tab = ARRAY[n] Of Student;
VAR T: Tab;
```

b) Accessing Fields of a Record in an Array

Each record corresponds to a cell in the array. Accessing a field of a record in the array is done by specifying the array name, followed by the record number in square brackets, followed by a dot, and then the field name. The access syntax is therefore as follows:

```
<name_Array>[<index>].<name_Field>
```

Where `<name_Array>` is the name of the array of records, `<index>` is the record number in the array, and `<name_Field>` is the name of the field we want to access.

Examples :

- `T[4]` represents the fifth record in the array of students `T` from the previous example.
- `T[4].FirstName` refers to the `FirstName` field of the fifth record in the array `T`.

c) Manipulating an Array of Records

The fields of records in an array are manipulated separately in the same way described previously.

However, traversing the records that make up the array's elements can be done using a loop.

Example:

Consider the following declaration :

```

CONST n = 30;
TYPE Student = RECORD
    Begin
    ID, Age: integer;
    LastName, FirstName: String[20];
    Marks: Array[1..9] of real;
    End;
    Tab = ARRAY[n] Of Student;
VAR T: Tab; i :Integer ;

```

The following instructions are accepted:

- 1) Read(T[0].LastName);
- 2) T[2].Marks[0] ← 18;
- 3) Write(T[5].FirstName);
- 4) T[10] ← T[9];
- 5) For i ← 0 To n-1 Do
 Read(T[i].age);

VI.4.3) Records in the C language

In the C language, the term "structure" is often used instead of "record." Each element of the structure is called a *field* or *member*.

Unlike arrays, the various elements of a structure do not necessarily occupy contiguous memory locations.

VI.4.3.1) Declaration of a Structure

The declaration of a structure variable can be done in various ways. However, in the C language, a structure is defined using the reserved keyword **struct**.

a) Declaration of a structure model

The first declaration involves defining a structure template and listing the fields it contains, and then declaring a variable of the defined template.

The declaration of a new structure template follows the following syntax:

```

struct <name_Template>{
    <type_Fields1> <name_Fields1>;
    <type_Fields2> <name_Fields2>;
    ...
    <type_Fieldsn> <name_Fieldsn>;
};

```

Where: <name_Template> is the name of the defined template, <type_Fields_i> is the type of the ith field of the structure, and <name_Fields_i> is the name of the ith field of the structure.

Please note the mandatory semicolon at the end of a structure definition.

Next, to declare a variable of the structure type corresponding to the previous template, we use the following syntax:

```
struct <name_Template> <name_Variable>;
```

Where: **<name_Variable>** is the name of the declared variable, and **<name_Template>** is the name of the defined template.

Examples:

- 1) This example illustrates the definition of a structure template named **Date**, composed of 3 integer fields: **Day**, **Month**, **Year**, and the declaration of associated variables.

```
struct Date{
    int Day,Month,Year;
};
struct Date d1,d2;
```

- 2) The following example defines a structure template labeled **Student** describing a university student known by their ID number, last name, first name, age, and marks. Then, declaring a structure variable following this template.

```
struct Student{
    int ID,Age;
    char LastName[20],FirstName[20];
    float Marks[9];
};
struct Student Stud;
```

Remarks :

1. Do not confuse the definition of the model, which is only a syntactic information for the compiler and does not reserve any memory space, with the declaration of variables of this model which effectively creates data.
2. Note the need to repeat the keyword **struct** in the declaration of a structure variable following a model.

b)Declaration by Defining Type Synonyms

To avoid the repetition of the **struct** keyword with every declaration of a structure type variable, we can proceed with the second method of declaration.

This method involves the use of the **typedef** keyword to define what is called in the C language a **shortcut** or a **type synonym**.

Declaration by defining a type synonym follows the following syntax:

```
typedef struct {
    <type_Field1> <name_Field1>;
    <type_Field2> <name_Field2>;
    ...
    <type_Fieldn> <name_Fieldn>;
}<name_Type>;
```


Where: `<name_Type>` is the name of the defined type synonym, `<type_Fieldi>` is the type of the i^{th} field of the structure, and `<name_Fieldi>` is the name of the i^{th} field of the structure.

In this way, the declaration of a structure variable is done as for a variable of another type by first putting the type and then the identifier of the variable, without needing to precede it with `struct`. The syntax for this declaration is as follows:

```
<name_Type> <name_Variable>;
```

Here, `<name_Variable>` is the name of the declared variable, and `<name_Type>` is the name of the defined structure type.

Example:

The definition using `typedef` of the previous `Date` structure type is as follows:

```
typedef struct {
    int Day,Month,Year;
} Date;
```

The declaration of variables `D1` of `D2` of type `Date` is simply done as follows:

```
Date d1,d2;
```

VI.4.3.2) Manipulating a structure

Similarly to algorithmics, manipulating a structure is done through its fields. However, if a field of the structure is of a given type, then all operations that can be performed on objects of that type can be carried out on that field.

The only allowed operation on structures as a whole is assignment.

a) Accessing a Field of a Structure

Accessing the different fields of a structure is done using the `"."` operator in the same way as in algorithmics. Thus, field access follows the following syntax:

```
<name_Structure>.<name_Field>
```

Where `<name_Structure>` is the name of the structure, and `<name_Field>` is the name of the field we want to access.

Example:

Consider the following declaration:

```
struct Date d;struct Student s;
```

Thus:

```
d.Day refers to the Day field of the structure d.  
s.Marks[0] refers to the first element of the array Marks of the structure s.
```

b) Reading and writing

Reading and writing a structure is done field by field.

The syntax for reading a field is as follows:

```
scanf("<format>",&<name_Structure>.<name_Field>);
```

The syntax for writing a field is as follows:

```
printf("<format>", <name_Structure>.<name_Field>);
```

Here, <format> represents the format for reading and writing the field ("%d", "%f", "%s",...), <name_Structure> is the name of the structure, and <name_Field> is the name of the field we want to read or display.

Example:

Reading the fields of the structure `D` of type `Date` is done as follows:

```
scanf("%d%d%d", &D.Day, &D.Month, &D.Year);
```

Displaying a date stored in the variable `D` can be done as follows:

```
printf("%d/%d/%d", D.Day, D.Month, D.Year);
```

c) Assignment

It is possible to assign a value to each field of the structure individually, just as it is possible to assign one structure to another structure of the same type.

The assignment between two structures copies all the fields from the source structure to their corresponding field in the target structure. This assignment follows the following syntax:

```
<name_Structure1> = <name_Structure2>;
```

Here, <name_Structure1> is the name of the target structure, and <name_Structure2> is the name of the source structure.

To assign a value to a specific field of the structure, we use the syntax:

```
<name_Structure>.<name_Field> = <value>;
```

With <name_Structure> being the name of the structure, <name_Field> being the name of the field we want to assign the value to, and <value> being the value to be placed in the field.

Example:

Consider two variables `D1` and `D2` of type `Date`. The following instructions are accepted:

```
D1.Day = 6;
D1.Month = 12;
D1.Year = 2022;
D2 = D1;
```

Remark:

In the C language, you can initialize a structure variable during its declaration, similar to arrays, using curly braces and commas. This initialization can be done in a sequential manner (the first value in the first field, the second value in the second field, and so on), selectively (specifying the fields along with their values), or in a mixed way (a combination of both).

Furthermore, it is possible to initialize only certain fields of the structure within the braces and leave the others empty.

Example:

Let's return to the definition of the `Date` type as defined earlier:

```
typedef struct {
    int Day,Month,Year;
} Date;
```

The following variables are initialized during their declaration:

- `Date birth_Date={7,2,2001};`
Sequential initialization (**Day**, then **Month**, then **Year**).
- `Date entry_Date={.Month=10,.Year=2015,.Day=17};`
Selective initialization.
- `Date exit_Date={20,.Year=2015};`
Mixed initialization. The month is not initialized in this case.

VI.4.3.3) Nesting of structures

Similarly to algorithmics, structures in the C language can be nested. Thus, a structure field can itself be of structure type, provided that this structure is defined earlier.

Example:

```
typedef struct {
    int Day,Month,Year;
} Date;
typedef struct{
    int ID;
    char LastName[20],FirstName[20];
    Date Date_Birth;
    float Marks[9];
} Student;
Student Stud;
```

To access the birth month of the student `stud`, you need to use the dot operator `"."` twice:

```
Stud.Date_Birth.Month
```

VI.4.3.4) Arrays of structures

Like in algorithmics, several structures of the same type can be grouped together in an array. This array is then an array of structures.

a) Declaration

The declaration of an array of structures is done in the same way as the declaration of an array whose elements are of a simple type: the type of the elements of the array (which is a structure type), followed by the array name and the number of elements in square brackets. The declaration of an array of structure requires that the type of structures has already been declared before.

Note that it is preferable to create a synonym for the array type (using `typedef`) and to declare array variables of that type.

Example:

Declaring an array of 100 persons, each defined by his last name, first name, and age, is done as follows:

```
#define n 100
typedef struct{
    char LastName[20], FirstName[20];
    int age;
} Person;
typedef Person Tab_Pers[n];
Tab_Pers T;
```

b) Manipulation

Each structure in the array of structures is identified by its index in the array, and each field in this structure is accessible through its name.

Thus, to access a field named `<name_Field>` of a structure in the `<index>` position of an array of structures `<name_Array>`, we use the following syntax:

```
<name_Array>[<index>].<name_Field>
```

Moreover, the fields of the structures in the array are manipulated separately in the same way described earlier. They can be read, written to, assigned values, etc.

However, the traversal of the records constituting the array's elements can be done using a loop.

Example:

Considering the array of structures `T` from the previous example, the following instructions are allowed:

```
1) gets(T[0].LastName);
2) T[2].age = 18;
3) puts(T[5].FirstName);
4) T[10] = T[9];
5) for(i=0; i<n; i++)
    scanf("%d", &T[i].age);
```

VI.5) Other possibilities for type definition

In addition to records and enumerations, there are other custom data types.

In this section, we will briefly describe two of them, namely: the interval type and the set type.

VI.5.1) Interval type

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

VI.5.1.1) Declaration

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

```
TYPE <Name_Type> = <lower_Bound>..<upper_Bound>;
```

Here, `<Name_Type>` represents the name of the type, and `<lower_Bound>` and `<upper_Bound>` are two constants indicating the lower and upper bounds of the interval, respectively. These two constants must be of the same scalar type.

Note that this type does not have an equivalent in the C programming language.

Examples:

The types `Month` et `Alphabet` are declared as follows :

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

Remark:

It is possible to declare an interval type, where the lower and upper bounds take their values from a previously defined enumeration type.

Example:

```
TYPE Day = (Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday);
TYPE Working_Days = Sunday..Thursday;
```

VI.5.1.2) Manipulation

A variable of the interval type can be manipulated like any other simple scalar variable. Thus, it can be read, written to, assigned a value, used in an expression, condition, loop, etc.

It is important to ensure, with each manipulation, that the value assigned to the interval-type variable does not exceed the bounds of the interval.

VI.5.2) Set type

The set type defines an unordered collection of elements of the same type, and its size is finite. It allows standard mathematical operations and relations such as union, intersection, complement, equality, inclusion, and membership.

VI.5.2.1) Declaration

A set type can be defined based on a base type for its elements in the following manner:

```
TYPE <name_Type> = SET OF <basic_Type>;
```

Where `<name_Type>` is the name of the defined set type, and `<base_Type>` is the base type of the elements in the set.

Thus, a variable of the set type can take as values all subsets of the base type.

Example:

```

TYPE   Color = (Red, Green, Blue, Yellow, White, Black);
        Numbers = 0..100;
        set_Colors = SET OF Color;
        Digits = SET OF Numbers;
VAR    c : set_Colors; d: digits;

```

In this example, the set type `set_Colors` is a set of the enumerated type `Color`, and the type `Digits` is a set of the interval type `Numbers`.

VI.5.2.2) Manipulation

Les variables de type ensemble ne peuvent être ni lues, ni écrites. Par contre, on peut affecter des valeurs à des variables de type ensemble. Pour ce faire on mets les valeurs entre crochets et séparées par des virgules.

The variables of the set type cannot be read or written. However, values can be assigned to variables of the set type. To do this, values are placed in brackets and separated by commas.

```
<name_variable> ← [<value1>,<value2>,...];
```

Examples:

- 1) Consider the following two sets:

```
VAR   flag_Color, trafficLight_Color : set_Colors;
```

They can be assigned values as follows:

```
flag_Color ← [Red, Green, White];
trafficLight_Color ← [Red, Green, Yellow];
```

- 2) Consider the following two sets:

```
VAR   A, B : Digits;
```

They can be assigned values as follows:

```
A ← [0, 3, 6, 9];
B ← [0, 6, 10];
```

VI.5.2.3) Set Operations

Consider the two sets **A** and **B** from the previous example.

The possible operations on sets are:

a) L'union

The union of two sets **A** and **B** (denoted as **A + B**) is the set composed of elements that belong to **A** or **B**.

Example:

```
A + B = [0, 3, 6, 9, 10]
```

b) Intersection

The intersection of two sets **A** and **B** (denoted as **A*B**) is the set composed of elements that belong to both **A** and **B**.

Example:

$$A * B = [0, 6]$$

c) Difference

The difference of two sets **A** and **B** (denoted as **A-B**) is the set composed of elements that belong exclusively to **A**.

Example:

$$A - B = [3, 9]$$

d) Comparison

There are only two comparison operators: equality (=) and inequality (\neq).

Example:

$$[7, 4] = [4, 7]$$

e) Inclusion

It is said that a set **A** is included in another set **B** (denoted as **A ≤ B**) if all elements of **A** belong to **B**.

Example:

$$[0, 6] \leq B$$

f) Containment

It is said that a set **A** contains another set **B** (denoted as **A ≥ B**) if all elements of **B** belong to **A**.

Example:

$$A \geq [3, 9]$$

g) Membership

To test if an element belongs to a set, the **in** operator is used.

Example:

```
3 in A returns True.
```

VI.6) Exercises**Exercise VI.1 :**

Write an algorithm allows to define an enumerated type, having the colors of a traffic light as values, intended for traffic control at an intersection. Subsequently, it reads a variable of integer type and displays which color corresponds to the entered value.

Exercise VI.2 :

Declare types that allow storing:

1. A day of the month (possible values between 1 and 31).
2. A month number (possible values between 1 and 12).
3. A year (possible values between 1 and 2010).
4. A date, composed of three values (day, month, year).
5. A football player characterized by his name, nationality, and date of birth.
6. A set of 11 football players.
7. A football team with its name, players, as well as the points earned, goals scored, and conceded in the current season.
8. A list of N football teams.
9. A sports tournament described by: a name, a list of participating teams, a start date, and an end date.

Exercise VI.3 :

Let $A(x_A, y_A)$, $B(x_B, y_B)$ and $C(x_C, y_C)$ be three points in the plane, with $x_A \neq x_B$ and $x_A \neq x_C$. These three points are said to be aligned (belong to the same line) if and only if the leading coefficient (slope) of line (AB) is equal to the leading coefficient of line (AC) .

The slope of line (AB) , denoted as m_{AB} , is given by:

$$m_{AB} = \frac{y_B - y_A}{x_B - x_A}$$

Write an algorithm to:

- Propose the most suitable data structure to represent a point in the plane.
- Read the coordinates of three points A , B , and C and determine if these three points are aligned. If the points are not aligned, the algorithm should calculate the distance between each pair of points.
- Remember that the distance between two points $A(x_A, y_A)$ and $B(x_B, y_B)$ is calculated as:

$$d(A, B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

It is assumed that there is a predefined algorithmic function called `sqrt` that calculates the square root of a number.

Exercise VI.4 :

Write an algorithm to determine the number of vowels in a sentence entered via the keyboard. Use the set $VL = \{a, e, u, o, i\}$.

VI.7) Solution of the exercises

Exercise VI.1 :

```

Algorithm TrafficLight_management;
TYPE trafficLight = (Green,Orange,Red);
Var num:Integer;
Début
Write("Provide a color number (between 0 and 2): ");
Read(num);
Case num Of
    ord(Green): Write("Green light");
    ord(Orange): Write("Orange light");
    ord(Red): Write("Red light");
    Otherwise: Write("Input error");
End;
End.

```

Exercise VI.2 :

```

Const N = 16;
1) Type Day = 1..31;
2) Type Month = 1..12;
3) Type Year = 1..2010;
4) Type Date = Record
    Begin
        d:Day;
        m:Month;
        y:Year;
    End;
5) Type Player = Record
    Begin
        Name, Nationality: String[20]
        Date_Birth:Date;
    End;
6) Type Set_Players = Array[11] Of Player;
7) Type Team = Record
    Begin
        Name:String[50];
        Players:Set_Players;
        Pts_Gained,Goals_Scored,Goals_Conceded:Integer;
    End;
8) Type List_Teams = Array[N] Of Team;
9) Type Tournament = Record
    Begin
        Name:String[50];
        Teams:List_Teams;
    End;

```

```
Date_Start, Date_End: Date;
End;
```

Exercise VI.3 :

```
Algorithm ex2;
Type Point = Record
    Begin
        x, y: Real;
    End;
Var A, B, C: Point; m_AB, m_AC, d_AB, d_AC, d_BC: Real;
Begin
    Read(A.x, A.y);
    Read(B.x, B.y);
    Read(C.x, C.y);
    m_AB = (B.y - A.y) / (B.x - A.x);
    m_AC = (C.y - A.y) / (C.x - A.x);
    If m_AB = m_AC Then Write("The 3 points are aligned")
    Else Begin
        d_AB = sqrt((B.x - A.x) ^ 2 + (B.y - A.y) ^ 2);
        d_AC = sqrt((C.x - A.x) ^ 2 + (C.y - A.y) ^ 2);
        d_BC = sqrt((C.x - B.x) ^ 2 + (C.y - B.y) ^ 2);
        Write("The distance between A and B is ", d_AB);
        Write("The distance between A and C is ", d_AC);
        Write("The distance between B and C is ", d_BC);
    End;
End.
```

Exercise VI.4 :

```
Algorithm VowelsCount;
TYPE Vowel_Set = Set of Characters;
Var VL: Vowel_Set; nb, i: Integer; Sentence: String[50];
Begin
    Write("Please enter a sentence: ");
    Read(Sentence);
    VL ← ['a', 'e', 'i', 'o', 'u', 'y'];
    nb ← 0;
    For i ← 1 to Length(Sentence) Do
        Begin
            If Sentence[i] in VL Then
                nb ← nb + 1;
            End;
        End;
    Write("The number of vowels is: ", nb);
End.
```

VI.8) Conclusion

In this chapter, we explored the limitations of predefined data types in handling the diverse array of data required to represent real-world objects effectively. To overcome this challenge, algorithmics and programming languages empower algorithm designers and programmers to introduce their own data types, commonly known as **custom types**.

While delving into various data types, our focus centered on two prominent categories: **enumerations** and **records**. Enumerations enable the systematic listing of an ordered set of possible values that a variable of this type can assume. Conversely, records offer a sophisticated data structure capable of amalgamating diverse data types. Consequently, the record type emerges as the most versatile choice for faithfully representing real-world entities. Through the utilization of records, one can articulate the characteristics of individuals, organizations, vehicles, and more, providing a comprehensive and flexible approach to data modeling.

References

- [1] M. Amad, "Algorithmique et Structures de Données", Course Material and Directed Exercises, 1st and 2nd year License, Abderrahmane Mira University of Bejaia, 2016.
- [2] L. Baba Hamed, S, Hocine, "Algorithmique et structure de données statiques: cours et exercices avec solutions", University Publications Office, Algiers, 2006.
- [3] M. Belaid, "Algorithmique & Programmation en Pascal, Cours, Exercices, Travaux Pratiques, Corrigés", Eurl Pages Bleues Internationales, Bouira - Algérie, 2008.
- [4] S. Bellaouar, "Algorithmique et Structure de Données : Partie 1", Course Handout, University of Ghardaia, 2020.
- [5] B. Bessaa, "Exercices corrigés d'Algorithmique", The LMD Booklets, Pages Bleues, Algiers, 2018.
- [6] A. Boucherit, " Algorithmique et Structures de données I", Course Material, 1st year MI, Hama Lakhdar University - El-Oued, 2020/2021.
- [7] D. Bouchicha, "Initiation à l'algorithmique et à la programmation en Pascal", Rached Edition, Sidi Bel Abbes - Algeria, 1st edition, 2019.
- [8] T. H. Cormen, "Algorithmes Notions de base", DUNOD, 2013.
- [9] V. Felea, V. Felea, "Introduction à l'informatique: Apprendre à concevoir des algorithmes - Cours et problèmes corrigés", Vuibert; 1st edition, 2013.
- [10] C. Haro, "Algorithmique raisonner pour concevoir", ENI Edition, 2015.
- [11] A. Lalouci, "Algorithmique et structures de données 1", Support de cours, 1^{ère} année MI, Centre universitaire de Mila, 2021/2022.
- [12] R. Malgouyres, R. Zrour, F. Feschet, "Initiation à l'algorithmique et à la programmation en C - Cours avec 129 exercices corrigés", DUNOD, 2nd edition, 2015.
- [13] K. Messaoud, "Algorithmique et structures de données 1", Course Material, 1st year MI, Mohamed Seddik Benyahia University - Jijel, 2021.
- [14] T Slimani, "Programmation et structures de données avancées en langage C: cours et exercices corrigés", Lulu.com Edition, 2014.
- [15] J. Tisseau, "Initiation à l'algorithmique", University Press - National School of Engineers Brest, 2009.
- [16] D. Zegour, "Apprendre et enseigner l'algorithmique (Tome 1): Cours et annexes", European University Editions, 2013.
- [17] Joyce Farrell, "Programming logic and design: comprehensive version", 8th Edition. Cengage Learning, 2015.