Ministry of Higher Education and Scientific Research

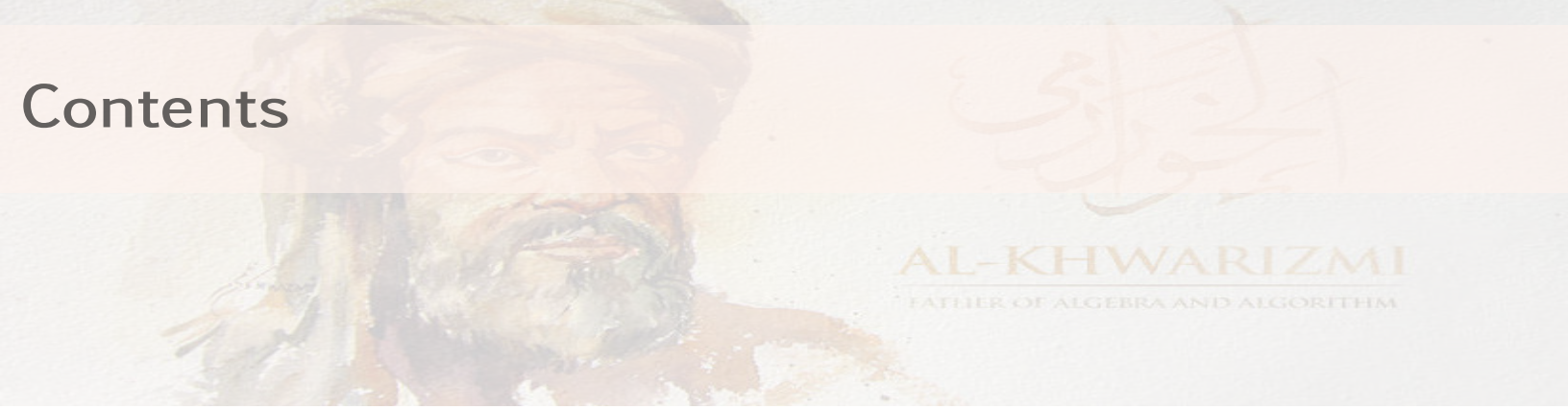
University of Ghardaia
Faculty of Science and Technology
Department of Mathematics and Computer Science

Course Handout

Algorithms and Data Structures: Part 1

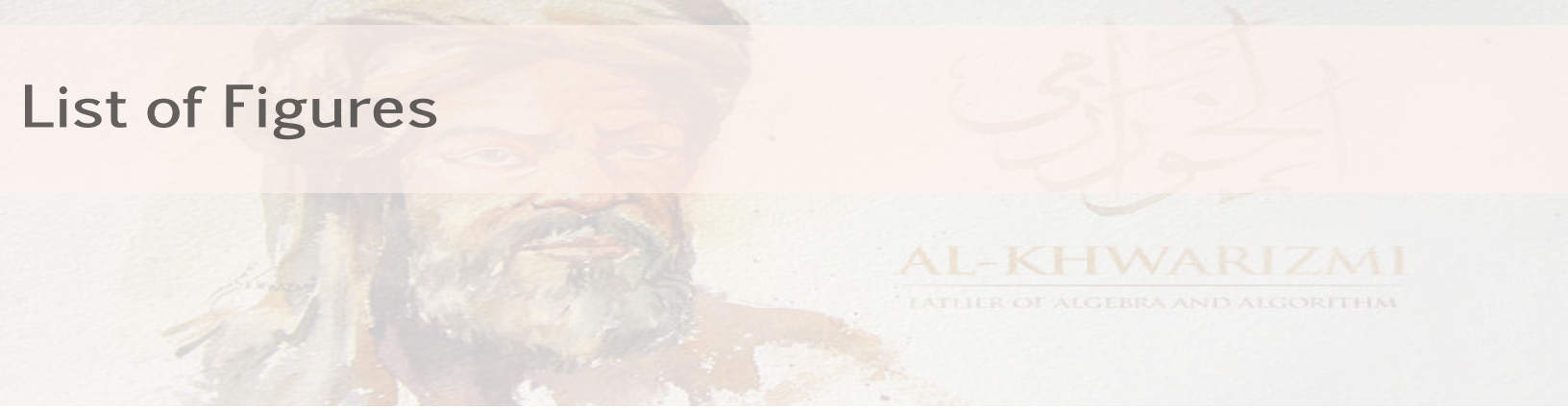
Dr. Slimane BELLAOUAR

2026 Edition

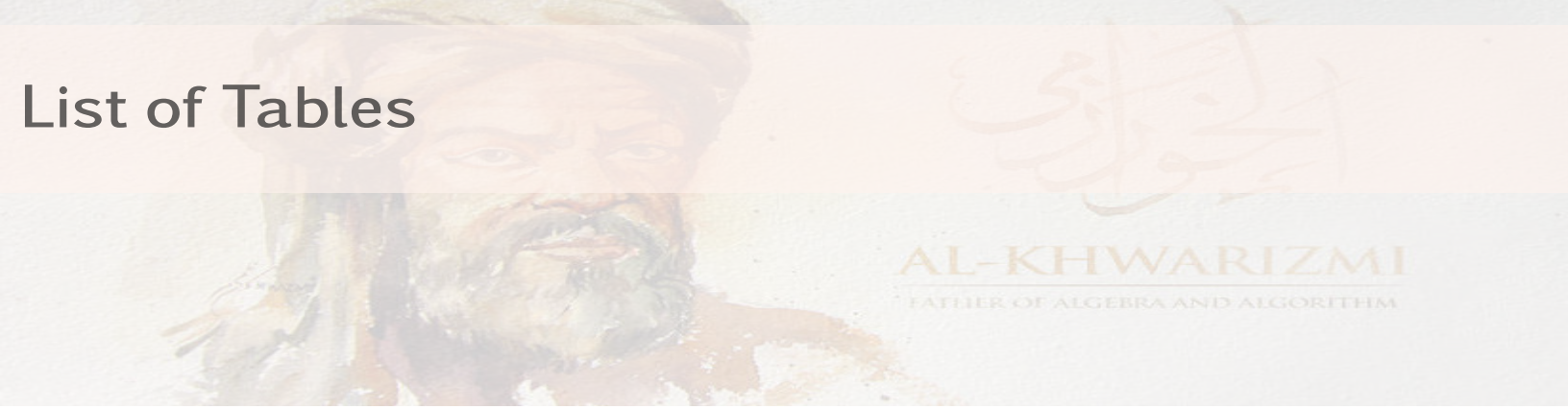


Contents

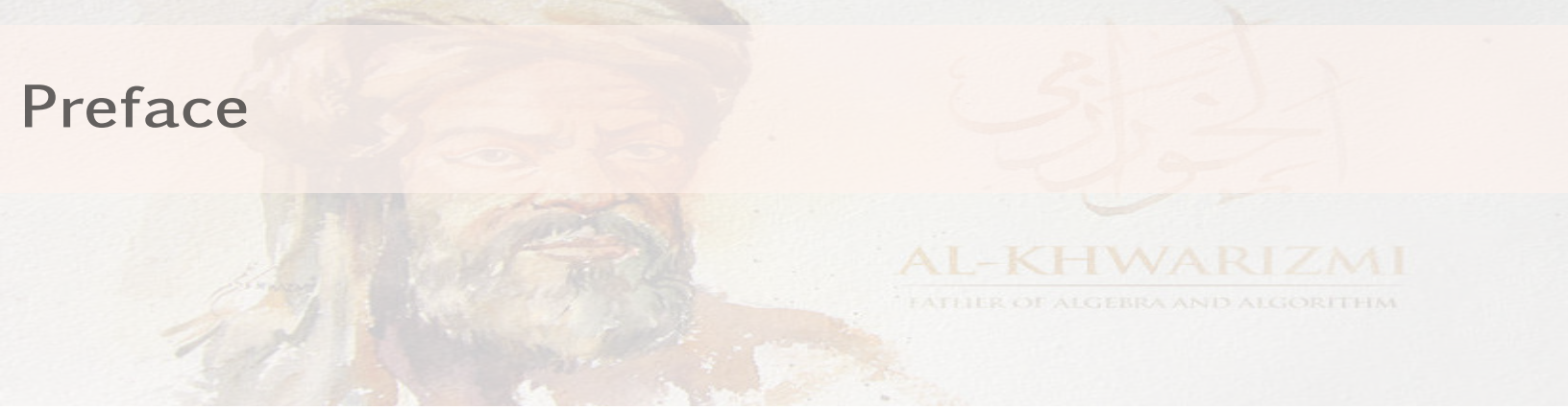
List of Figures



List of Tables



Preface



A widely held, yet mistaken, belief in public, industrial, and even academic circles is that a computer is a machine capable of solving problems because it is endowed with a certain intelligence. In reality, this machine is capable of nothing unless an entity (for example, a programmer) has shown it the steps to follow, without any belief that this machine can provide the slightest interpretation of these steps. In other words, the machine blindly executes the steps. This philosophy of showing the machine the steps to follow to solve a problem is called an *algorithm*.

A survey of graduates from Stanford University asked which courses they relied on most in their professional careers. The introduction to programming module took first place, followed closely by core software courses covering basic data structures and algorithms.

In light of this, this course handout, *Algorithms and Data Structures: Part 1*, is tailored for first-year undergraduate students in Computer Science and Mathematics. As of the 2025–2026 academic year, it fully complies with the most recent official ministry curriculum for the Computer Systems (*Systèmes Informatiques*) specialization. Furthermore, it serves as an excellent foundational resource for other technical disciplines and institutions teaching introductory programming and algorithm modules.

Additionally, in strict alignment with the strategic policy of the Ministry of Higher Education and Scientific Research, this course manual is presented in English. This initiative promotes the widespread use of English in higher education and research, ensuring our students are well-equipped to integrate seamlessly into the global academic and professional landscape.

While algorithm textbooks are abundant in the literature, they can often be daunting or overly theoretical for novice students. Our objective is to provide learners with a manual that is accessible, practical, and easy to comprehend, both conceptually and technically. To achieve this:

- We draw upon all our experience acquired over more than seventeen years of

teaching algorithms at the University of Ghardaia and other institutions.

- We accompany the explanation of new concepts and new techniques with examples and illustrations.
- Each chapter is followed by a certain number of exercises of varying degrees of difficulty, well-designed to implement the theoretical knowledge acquired. These exercises aim to cover professional life situations (mathematical calculations, management, . . .).

For the algorithmic formalism, we mainly used the conventions of the Pascal language. This formalism makes it easier for students to implement algorithms in a high-level language. In order to implement the algorithmic solutions, in this handout, we have chosen to use two programming languages: C and Pascal. The C language is chosen because it is prescribed in the official ministry curriculum. The Pascal language was chosen for its pedagogical value in helping students master fundamental programming concepts.

To be in line with the official ministry curricula, this book is the first part of a series of three parts that cover the teaching of the algorithms and data structures module for the first three semesters of the Computer Science specialty in the Mathematics and Computer Science domain. The first part, "Algorithms and Data Structures 1", covers the following elements:

1. Introduction to Computer Science.
2. Basic Elements of Algorithms.
3. Presentation of the Algorithmic Formalism.
4. Arrays and Strings.
5. Custom Types.

We also plan to include *Python* programming languages in future versions. With the aim of facilitating and widening access to this series of handouts, we plan to produce versions in Arabic.

At the end of this work, I would first like to thank God the Almighty and Merciful, who gave me the strength and patience to accomplish this modest work.

My sincere thanks go to Messrs. Slimane Oulad-Naoui, Abdelkader Bouhani, Chaker Abdelaziz Kerrache, Khaled Kechida, Ahmed Saidi, Messaoud Benguennane and Misses Habiba Benabderrahmane, Wassila Belfardi, and Asma Bouchekkouf teachers at the University of Ghardaia, who accompanied me throughout these years in teaching the *Algorithms and Data Structures* module. Their footprints are clearly visible on the content of this handout.

Finally, my thanks also go to all those who have, in one way or another, directly or indirectly, provided their support.

1. Introduction to Computer Science

This chapter, an introduction to computer science, is at the beginning of this manuscript, even though its content is not actually part of algorithms and data structures. Nevertheless, we have included it in this document for two main reasons. First, to comply with the curriculum framework for the Mathematics and Computer Science (MI) domain. Moreover, we find it very useful for a new student wishing to enter the world of algorithms, data structures, and programming to have a general understanding of the computer science discipline as a whole, and the computer and its components in particular.

This chapter first defines the word "computer science" (informatique) and then outlines the computer science discipline. In addition, the chapter introduces the definition of a computer, its functions, the relationship between software and hardware, the types of computers, and finally, the various components of a computer according to the von Neumann architecture.

In this chapter, we decided not to present the information representation section because, according to the new curriculum framework, it will be covered in detail in the Machine Structure 1 *module, which will be taught in the same semester (Semester 1) as the* ADS1 *module.*

1.1 What is Computer Science?

The French word *informatique* is a contraction of two words: *information* and *automatique* (automatic). The word information refers to information science, while the word automatic is likened to the art of performing actions automatically and rationally. From this, Definition 1.1 is derived:

Définition 1.1 **(Computer Science / Informatique)** Computer science is the study of the automated and rational processing of information using a machine (computer).

Following this definition, we can deduce that the computer science discipline can be applied in several fields, whether scientific, technical, economic, or social. What should also be remembered is that computer science is not the result of synthesizing several disciplines. Rather, it is a discipline in its own right. It essentially comprises a fundamental aspect and an applied aspect. In the fundamental aspect, we can study language theory, computability, logic, ... Whereas in the practical aspect, we can tackle the subjects of programming, software engineering, operating systems, programming languages, compilation, ... In the Anglosphere, the discipline is primarily referred to as *'computer science'*, though *'Informatics'* is occasionally used in specific academic contexts.

1.2 Computer

Définition 1.2 **(Computer)** A computer is a programmable electronic machine capable of performing logical calculations on binary numbers.

The French word "*ordinateur*" was introduced in 1955 by IBM as a French name for the new IBM 650 machine, avoiding a direct translation of the English word *computer* (calculator).

1.2.1 Functions of a computer

A computer must ensure the following four basic functions:

1. Acquire information (acquisition).
2. Retain information (storage).
3. Process information (calculation).
4. Output results (restitution).

Data acquisition is the *input* function. It manifests itself through the reception of data or instructions by the computer, which has storage media to *memorize* information when it is running and when it is turned off. The *calculation* function, or processing in general, translates to executing certain basic operations on the data. Of course, the computer must be able to produce and/or communicate the results. This is what we call the *output* function. Based on these four basic functions, it is possible for a user to perform many tasks: scientific calculations, office automation, email, management applications, computer-aided design, games, ...

1.2.2 Hardware and software

If we return to Definition 1.2, we can see that it comprises three parts: an electronic machine, a programmable machine, and logical calculation on binary numbers. The first part, electronic machine, refers to the term *hardware* , which is an assembly of equipment (processor, memory, peripherals, ...) whose operation is governed by the laws of physics. The second part, programmable machine, draws our attention to the fact that this electronic machine is only capable of performing different tasks through the instructions given to it. These instructions, assembled in the form of programs, are called *software*.

The third part implies that the computer can process different types of information (text, numbers, sounds, images, videos, instructions) in binary form.

1.2.3 Types of computers

There are several taxonomies of computer types. Here, we retain the one based on size and power:

1. *Supercomputer*: A supercomputer is a generic word referring to the computer with the highest possible performance, in particular the fastest at the time of its design. Supercomputers are very expensive and are used for specialized applications requiring immense amounts of mathematical calculations. For example, we can cite the following applications: weather forecasting, scientific simulation, computational fluid dynamics, nuclear energy research, electronic design, geological data analysis, ... The discipline dealing with supercomputers is called "High-Performance Computing" (HPC).

   The first supercomputers emerged in 1961. They are named IBM Stretch or IBM 7030. Today, some supercomputers adopt the technique of massively parallel systems and use RISC (Reduced Instruction Set Computer) type microprocessors. Cray Research (founded by Seymour Cray after leaving Control Data Corporation (CDC)) is the best-known supercomputer manufacturer.

2. *Mainframe*: A Mainframe refers to a high-performance central processing unit connected to a network of terminals. These large and expensive types of computers are capable of simultaneously supporting hundreds or even thousands of users. Thus, they are primarily used by governments and large organizations for bulk data processing, critical applications, transaction processing, census statistics, ... Among mainframe manufacturers, we can count the Bull company with the DPS/6 to DPS/8 running the GCOS (General Comprehensive Operating System) system. .

3. *Minicomputer*: Minicomputers are medium-sized computers. A minicomputer is a multiprocessing system capable of simultaneously supporting up to 200 users. As a non-exhaustive example, we can cite the PDP-7, PDP-8, and PDP-11 series from the *Programmed Data Processor* range by the manufacturer *Digital Equipment Corporation* as typical minicomputers.

4. *Personal computer*: A personal computer (PC) is designed to be used by one person at a time. PCs are based on microprocessor technology that allows manufacturers to put the entire central processing unit on a chip. The first consumer personal computers appeared in the late 1970s. In 1977, Apple Computer introduced *Apple II*, one of the most popular PCs. Later, in 1981, IBM built its first personal computer known as the *IBM PC*.

   In recent years, the term PC applies to any computer based on an Intel microprocessor or an Intel-compatible microprocessor. Personal computers are divided into several categories:

   - *A desktop computer* is designed to be used at a desk and is rarely moved. It consists of a box called the system unit that contains most of the essential components (Figure 1.1). The monitor, keyboard, and mouse all connect using cables (or, in some cases, wireless technology). Desktop computers

Figure 1.1: Desktop computer

     are flexible, as we can connect the monitor, keyboard, and mouse of our choice, as well as install additional storage drives, memory, and expansion cards.
   - *A laptop (portable) computer*, sometimes called a laptop or notebook, is a small personal computer, compact and lightweight. Also, it has its own power supply in the form of a battery. Therefore, it is very useful for mobility. It consists of a foldable casing; its cover opens to reveal a screen, a keyboard, and an integrated pointing device that replaces the mouse (Figure 1.2).
   - *Netbook* is short for Internet notebook. It is a smaller and less powerful laptop, designed primarily for accessing the Internet. A netbook is generally cheaper than a laptop, but lighter and more convenient to carry. However, it is not powerful enough (in terms of processor and memory) to run all desktop applications. In addition, the hard drive is replaced by flash memory to reduce power consumption and cost.
   - *A tablet* is a portable computer consisting of a touchscreen and a set of integrated peripherals. It does not have a keyboard or a pointing device; a software keyboard and sliding a finger on the screen act as such. Tablets have limited memory and storage capacities.

Figure 1.2: Laptop computer

- *A smart phone* is a mobile phone capable of running applications and equipped with Internet capabilities. Smart phones generally have touch-screens (Figure 1.3). They have a variety of location-sensitive applications, such as the Global Positioning System (GPS), mapping programs, and local commercial guides.

Figure 1.3: Smart phone

1.2.4 Computer Architecture

We focus here on the so-called von Neumann architecture, named after the mathematician John von Neumann, born in 1903 in Budapest (Hungary). This architecture is also called the von Neumann model or Princeton architecture.

Définition 1.3 (von Neumann Architecture) The von Neumann architecture is based on the concept of a stored-program computer, in which data and instructions are stored in the same memory.

In the von Neumann architecture (Figure 1.4), we can distinguish four parts:

1. the central processing unit,
2. the memory,
3. input devices,

4. output devices.

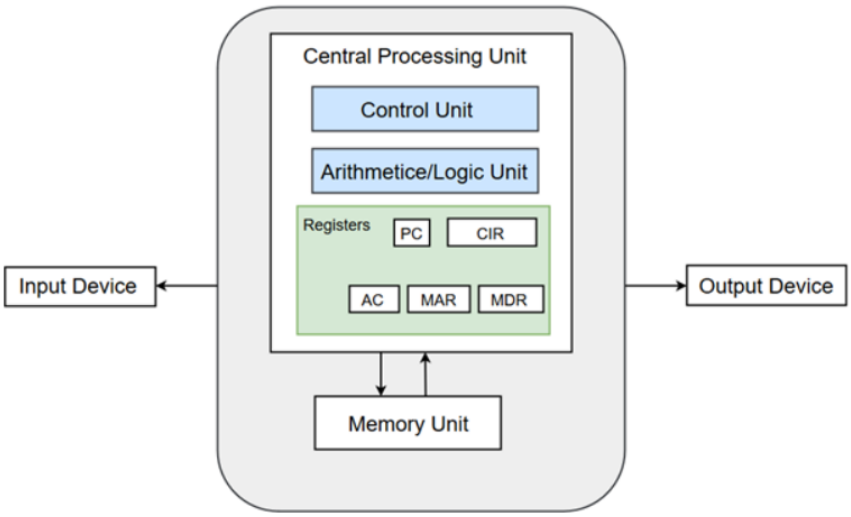


Figure 1.4: von Neumann Architecture

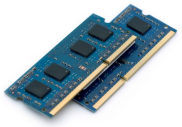
Central Processing Unit

The central processing unit, or Central Processing Unit (CPU), is a device that interprets and executes the commands you give to a computer. The CPU is also known as a processor. It determines the speed of the computer, which is measured in Hz (KHz, MHz, GHz). The CPU (processor) consists of electronic circuits that can execute primitive (hardwired) actions constituting the processor's instruction set, also called "machine language".

In the von Neumann model, instructions are executed one after the other (principle of sequential execution). The CPU mainly contains the control unit (CU) and the arithmetic logic unit (ALU). The CU controls the operation of the ALU, the memory, and the input/output peripherals of the computer, telling them how to respond to the program instructions it has just read and interpreted from memory. The CU also provides the timing and control signals required by other computer components. The ALU allows performing arithmetic (addition, subtraction, ...) and logical (AND, OR, NOT, ...) operations.

Memory

Définition 1.4 (Memory) A memory is a device capable of recording information, storing it, and retrieving it.



In the von Neumann architecture, memory refers to the main or central memory. Unlike a hard drive (secondary memory), this memory is fast and also directly accessible by the CPU. Memory is divided into a certain number of locations (memory words). Each location is uniquely identified by an address. Only the processor can access memory locations, either for writing or reading. When writing, the processor specifies an address and content, and the memory stores the content in the location indicated by the address. When reading, the processor specifies a given address and retrieves its content, which remains unchanged. Two commonly used types of memory are random access memory (RAM) and read-only memory (ROM).

RAM is characterized by the temporary storage of instructions and data. They risk being altered by a power failure; thus, RAM is described as volatile. ROM can be read but not (or rarely) written to. This type of memory is non-volatile; it contains programs necessary for the hardware to operate, particularly during the loading of the operating system. Writing information into ROMs is called programming, which is generally done by the manufacturer according to a certain method depending on the type of ROM:

- ROM.
- PROM (Programmable ROM) : can be written to only once by the user.
- REPROM (REProgrammable ROM) : can be written to a certain number of times by the user:
  - E-PROM (Erasable PROM) : erased by exposure to ultraviolet light.
  - EA-PROM (Electrically Alterable PROM) : modified by electrical voltage.
  - EE-PROM (Electrically Erasable PROM) : erased by electrical voltage.
  - FLASH EPROM : EE-PROM, but erased by block (typically 512 bytes or more) and not byte by byte.

One of the main characteristics of memories is capacity (Definition 1.5).

Définition 1.5 (Capacity) The capacity (size) of a memory is the amount of information that this memory can record.

Capacity can be expressed in:

- bit : The term bit is a contraction of the words "binary" and "digit". A bit can only take two values, most often designated by the digits 0 and 1. It represents the minimum amount of information. Thus, it constitutes the unit of measurement of information in computer science.
- Byte (Octet) : 1 Byte = 8 bits.

- Kilobytes (KB) : 1 Kilobyte (KB) = 1024 bytes = 2^{10} bytes.
- Megabytes (MB) : 1 Megabyte (MB) = 1024 KB = 2^{20} bytes.
- Gigabytes (GB) : 1 Gigabyte (GB) = 1024 MB = 2^{30} bytes.
- Terabytes (TB) : 1 Terabyte (TB) = 1024 GB = 2^{40} bytes.

Furthermore, in the literature, we also speak of *cache* and *registers* as types of memory with considerable access speed that matches the speed of the CPU. In this context, we take the opportunity to introduce secondary memories, which are storage devices.

Storage Devices

In this section, we focus on storage devices or secondary memories. This type of memory keeps information indefinitely, so it is considered non-volatile. In other words, the information persists even after the power is cut. There are many types of secondary memories; we present the most common ones:

Magnetic Disks A magnetic disk is an information storage device supporting high storage density and relatively fast access time. It consists of a rigid surface covered with a magnetic coating.

In the case of a hard drive having several read heads, only one head is used at any given time to read or write; data is therefore stored serially, even though the heads can in principle be used to read or write several bits in parallel. The heads write data by magnetizing the magnetic material as it passes under the heads and read data by detecting magnetic fields. The current capacity of hard drives is on the order of TB. A *floppy disk* contains a flexible plastic platter covered with a magnetic material such as iron oxide. Access time for floppy disks is generally slower than a hard drive. Floppy disk capacities can reach 1.44 MB. However, high-capacity disk drives appeared later. For example, the *Iomega Zip* has a capacity of 100 MB while the capacity of the *Iomega Jaz* is 2 GB.

Optical Disks *Optical disks* use laser beam technology to store and retrieve data.

CD ROM: A CD ROM (Compact Disk Read Only Memory) is an optical disk 12 cm in diameter and 1.2 mm thick that can be written to once and read at will (Write Once Read Many). CD ROMs are used for computer data following the development of CD technology introduced in 1983 and applied to audio. A CD ROM is composed of plastic covered

with aluminum, which reflects light differently for flats (lands) or pits (pits), which are areas created during the burning process. A CD ROM can store about 650 MB of computer data or 74 min of audio data. The data is written on the disk in the form of a spiral track that is almost 5 km long, from the center outwards and has 22188 turns, in the following way:

- the size of a bit on the CD is standardized and corresponds to the distance of 0.278 μm.
- the "1"s are recorded in the form of a transition (edge of a pit)
- the "0"s are recorded in the form of a flat area (bottom of a pit or land)

DVD: The digital versatile disc, or DVD for Digital Versatile Disc, is a newer version of optical disk storage. There are industry standards for data storage on DVD-Audio, DVD-Video, DVD-ROM, and DVD-RAM. When a single side of the DVD is used, its storage capacity can reach 4.7 GB. DVD standards also include the possibility of storing data on both sides in two layers of each side, for a total capacity of 17 GB. DVD technology is an incremental advance over the CD, not an entirely new technology. In fact, the DVD player is backward-compatible. It can be used to play CDs and CD-ROMs as well as DVDs.

Others: A CD R (CD Recordable) is an initially blank CD allowing the user to record their data using a CD burner. But, once burned, the CD R is used like a simple CD ROM. A *DVD-R* has the same format as for a DVD-ROM but it is no longer possible to use multiple layers per side. This gives a capacity of 4.7 GB per side. A *CD RW (CD ReWritable)* is a rewritable CD that can be erased and written to using a CD burner a very large number of times. It can be read by a CD player.

A *Blu-Ray disc* is an optical disk that uses a blue laser diode developed in 1996 by Shuji Nakamura. The use of this type of diode allows reducing the spot size and therefore increasing the data density on the disk. Consequently, Blu-Ray discs can store up to 27GB on a single layer, and can therefore reach 100 GB on 4 layers.

Peripherals

> Définition 1.6 **(Peripheral)** A peripheral is a hardware device that can be attached to a computer and which provides information exchange as input and/or output between the computer and the external environment.

We distinguish three types of peripherals, namely, input, output, and mixed. However, it should be noted that some literature includes storage devices as a type of peripheral.

Input Peripherals An input peripheral is equipment used to enter information into the computer:

- Keyboard: The keyboard is a peripheral that contains several keys to write and enter various characters (letters, numbers, punctuation, ...) as well as special keys and function keys. The distribution of letters of a language on the keyboard is a function of the frequency of use of these letters. The principle consists of placing the most frequent letters on the most accessible keys of the keyboard. There are many key layouts: AZ...

  AZERTY, QWERTY, QWERTZ, Dvorak, ...

- Mouse: The mouse is a pointing device used to move a cursor on the screen and allows selecting, moving, and manipulating objects by clicking a button. There are several families of mice (mechanical, optomechanical, optical, wireless).

- Microphone: A microphone is used to input sound (voice) into the computer.

- Webcam: A webcam is a camera used to film and produce video as input to the computer. It is generally located on the top bezel of the screen. It can be external or internal.

- Scanner: A scanner is an accessory used to digitize a document (text or image) and input it into the computer as a digital image. It is possible to transform this image into text using OCR (Optical Character Recognition) technology. In businesses, a scanner is generally used in conjunction with an EDMS (Electronic Document Management System).

- Others: Barcode label readers, QR (Quick Response) code readers, biometric fingerprint and chip time clocks, credit card readers, ...

Output Peripherals

- Screen: A screen or monitor is a peripheral that displays information entered or requested by the user. Indeed, it displays the images generated by the computer's graphics card. There are two families of screens: Cathode Ray Tube screens (CRT) and flat screens, which are liquid crystal or electroluminescent panels.

- Printer: A printer is a peripheral that allows transferring texts or images (from the computer) onto paper or transparency sheets. On the market, there are essentially three printer models: dot-matrix, inkjet, and laser.

- Plotter: A plotter is a printing peripheral for vector graphics. This peripheral is used in CAD (Computer-Aided Design) and drafting. Since the 1980s, plotters have been replaced by large-format laser and inkjet printers.

- Speaker/Headphones: A speaker is a peripheral that emits sounds from the computer. Some monitors have built-in speakers.

Mixed Peripherals

- Modem: A modem (modulator-demodulator) is a small box that connects to the Internet. Also, a modem is used between two computers to exchange data over the telephone network. Modulation is the output function; it converts a digital signal into an analog signal. Demodulation constitutes the input function.
- Multifunction Unit: A multifunction unit is a peripheral capable of performing several input/output tasks: printing, scanning, sending faxes, receiving faxes, ...
- Touchscreen: A touchscreen is a display screen that is also pressure-sensitive; it interacts with the user as soon as an object (usually the user's finger or a stylus) touches the screen. Thus, it serves as both an input and output device.

2. Basic Elements of Algorithms

Before studying a new discipline, one usually starts with its basic elements. This chapter takes on this task and proposes to introduce the fundamental elements of an algorithm. Indeed, we begin with the definition of an algorithm and the study of its properties. Next, and in an algorithmic approach logic, we move on to the analysis and conception phase, up to the use of an algorithmic formalism.

In the second part of this chapter, we discuss the essential components required to formulate algorithmic solutions, such as variables, constants, expressions, data reading and writing, sequential processing, conditional processing, and finally, iterative processing (loops).

2.1 What is an algorithm?

In order to answer the question "what is an algorithm?", we start with a few everyday life questions:

- Do you know how to start a car and get it moving?
- Do you know how to calculate the roots of a second-degree polynomial: $ax^2 + bx + c = 0$, with $a \neq 0$?
- Have you ever installed a printer?

If the answer is yes, without knowing it, you have already executed an algorithm!

A few more questions:

- Have you ever given directions to someone who is lost?

- Have you ever shown your neighbor a cooking recipe?

If the answer is yes, without knowing it, you have already (built) and had an algorithm executed!

Through this brief interrogation, we can deduce that algorithms are part of our daily lives, whether we know it or not.

Still on our path towards a definition of the word algorithm, let's give some illustrative examples (Examples 2.1 and 2.2) that attempt to answer some of the previously seen questions: :

Exemple 2.1 (Starting a car and getting it moving)

1. Open the car door,
2. sit down (where the steering wheel is),
3. put on the seatbelt,
4. adjust the seat,
5. adjust the rearview mirrors,
6. check that it is in neutral,
7. put the key in the ignition,
8. turn the key,
9. press the clutch pedal (the one on the left),
10. shift into 1st gear,
11. slowly release the clutch pedal,
12. don't forget to release the handbrake.

Exemple 2.2 (Solving the equation $ax^2 + bx + c = 0$, with $a \neq 0$)

1. Input the values of (a, b, c),
2. calculate $\Delta = b^2 - 4ac$,
3. if $\Delta < 0$ then no roots in $\mathbb{R}$
   if $\Delta = 0$ then a double root: $x = -\frac{b}{2a}$
   if $\Delta > 0$ then two roots:
   $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $\quad x_2 = \frac{-b+\sqrt{\Delta}}{2a}$.

Now, following the interrogation and Examples 2.1 and 2.2, we can arrive at a first definition (Definition 2.1) of the word algorithm:

Définition 2.1 (Algorithm) An algorithm is a *sequence of actions* that, once correctly *executed*, leads to a given *result*.

We will try to discuss a few points in Definition 2.1 to establish some ideas:

- An *action* is a step in the algorithm. It can be composed of several *primitive actions*. A *primitive action* can be executed without any additional information.
- An action is *executed* by a *processor* , which can be defined as an entity capable of understanding and executing the actions that make up an algorithm by manipulating a set of objects (elements) called an *environment*. A processor can be:
  - A person,
  - an electronic device,
  - a mechanical device,
  - a *computer*.

In light of the discussion above, we can extend Definition 2.1 as follows:

Définition 2.2 (Algorithm) An algorithm is a sequence (series) of primitive actions that, when executed by a well-defined processor, will perform a very specific task.

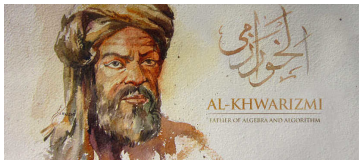
2.2 Properties of an Algorithm

After introducing the definition of the term algorithm (Definition 2.2), it is recommended to present the properties that an algorithm must have:

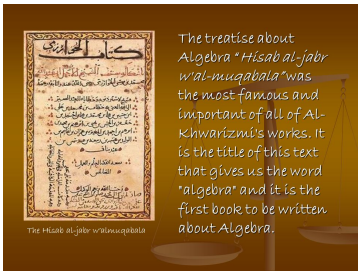
- *Generality*: It must take into account all possible cases. It handles the general case as well as special cases.
- *Finiteness*: It must always terminate after a finite amount of time.
- *Repetitiveness*: It is generally repetitive (it contains a repeating process).
- It is independent of programming languages and computer hardware.
- Under similar execution conditions (with identical data), it always yields the same result.

2.3 Origin of the word algorithm

We often think of an algorithm as something modern, but the term actually dates back about 900 years. The word algorithm comes from the name of the Persian Arabic-speaking mathematical genius **Muhammad ibn Musa al-Khwarizmi**.

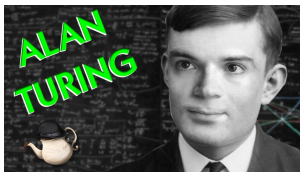


He was born around 780 AD in the region that gave him his nickname, Khwarazm, now in Uzbekistan. During the era of the Abbasid Caliph Abdallah al-Mamun, he was a member of the "House of Wisdom" in Baghdad, an institute where scholarly minds conducted scientific research. He made groundbreaking contributions to mathematics, astronomy, geography, and cartography. He wrote an influential book titled "On the Hindu Art of Reckoning." Three hundred years later, the book was rediscovered and translated into Latin. It introduced Hindu-Arabic numerals to the West, eventually replacing the cumbersome Roman numerals. The Hindu-Arabic numeral system, as well as the decimal point described in Al-Khwarizmi's book, form the basis of the numbers we use today worldwide. The name Al-Khwarizmi, once Latinized in the book's title, became *algorithmi*, and this is the origin of the word *algorithm*.



We must also thank Al-Khwarizmi for the word *algebra*, which comes from another treatise titled *Hisab al-jabr w'al-muqabala*. His works revolutionized mathematics in the West, showing how complex problems can be broken down into simpler parts and solved.

The word algorithm was coined in the Middle Ages to simply designate the calculation techniques related to the positional numeral system—in other words, calculating with digits—introduced to Europe by the Arabs. By the 13th century, it had become an English word. But it was not until the end of the 19th century that the word algorithm came to mean step-by-step actions performed to solve a problem.



At the beginning of the 20th century, Alan Turing, a British mathematician and computer scientist, explained how, in theory, a machine could follow algorithmic instructions and solve complex mathematical problems. This was the birth of the computer age.

During World War II, Alan Turing built a machine called the *Bombe* which used algorithms to crack the German *Enigma* codes (a portable electromechanical machine used for the encryption and decryption of information).

Algorithms are now everywhere, helping us get from point *A* to point *B*, searching the Internet, making recommendations on what we can buy, watch, or share, and making predictions.

This little word, which has its origins in the Arabic Middle Ages, is gradually transforming our lives.

2.4 From the Problem to the Result

In order to provide a general overview of a programmer's work, in this section we will take a look at the problem-solving steps, from the initial contact with the problem to obtaining the results.

To help the student follow along easily, we adopt a reverse-engineering approach that starts with something more concrete for the student: computer applications. Then, we work our way backward, introducing the necessary ingredients to arrive at the starting point—which is, of course, the problem to be solved.

2.4.1 Computer Applications

Since the 1950s, computer applications have continued to develop and diversify. They have become an indispensable tool in our professional and private lives. A non-exhaustive list of applications includes:

- Internet access,
- sending emails,
- website creation,
- photo archiving and editing,
- video games,
- office automation: word processing, spreadsheets, database management, . . . ,
- management and accounting: invoicing, payroll, inventory, . . . ,
- scientific computing,
- weather forecasting,
- Computer-Aided Design (CAD) or drafting,
- controlling satellites, experiments, ...

But the question arises: why can such a machine, the computer, perform so many varied tasks?

The answer is quite simple: because the computer can be programmed. By programming, we mean that we must provide the computer with the set of operations to execute.

2.4.2 Program

We begin with Definition 2.3.

Définition 2.3 (Program) A *program* consists of a set of directives called *instructions*.

Instructions specify (to a computer):

- the elementary operations to be executed,
- the sequence in which they are chained together.

However, it should be noted that a computer has a limited repertoire of *elementary operations* that it can execute very quickly. Similarly, it knows how to make choices as well as repetitions.

Figure 2.1 illustrates the relationship between an algorithm and a program.

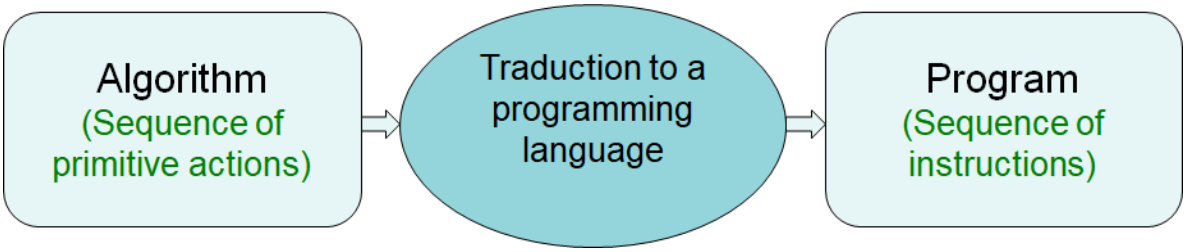


Figure 2.1: Translation of an algorithm into a program.

Indeed, an algorithm is written on paper, formally describing a method for solving a problem. This description must be in a precise language understandable by humans.

In contrast, a program is a representation of a sequence of instructions in a machine's memory. It is a representation in a programming language that can be interpreted by the machine.

Therefore, to go from an algorithm to a program, we need a translation, generally performed manually by the application developer.

Following this illustration, Definition 2.4 emerges.

Définition 2.4 (Program) A program is a *sequence of instructions* written in a *programming language* translating an algorithm.

2.4.3 Programming Languages

In Section 2.4.2, we introduced the concept of a programming language, perhaps without giving a reason. In this section, we discuss the reason why we use a programming language. Subsequently, we conduct a survey of some commonly used programming languages.

Translation

A computer only knows the binary numeral system. The only language the computer can understand is a long sequence of bits (0 and 1) often processed in words (groups of 8, 16, 32, or 64 bits). Thus, any information (data or instruction) processed by the computer must be encoded in binary.

For example, suppose the machine instruction 0101010011011010 means to add (operation code 0101) the content of the accumulator register *AX* (010011) with a value located at address 011010 and place the result back into the accumulator.

It is clear that it is difficult, if not impossible, for a human being to write and understand programs in machine language.

One solution is to use:

1. a language in an accessible format to express the program. The language used is called a *programming language* and the written program is called a *source program* or *source code*.

2. a *translator* for the language used. The translator comes in several forms, namely, *assembler*, *interpreter*, or *compiler*, depending on the type of programming language.

This scenario is illustrated using Figure 2.2.

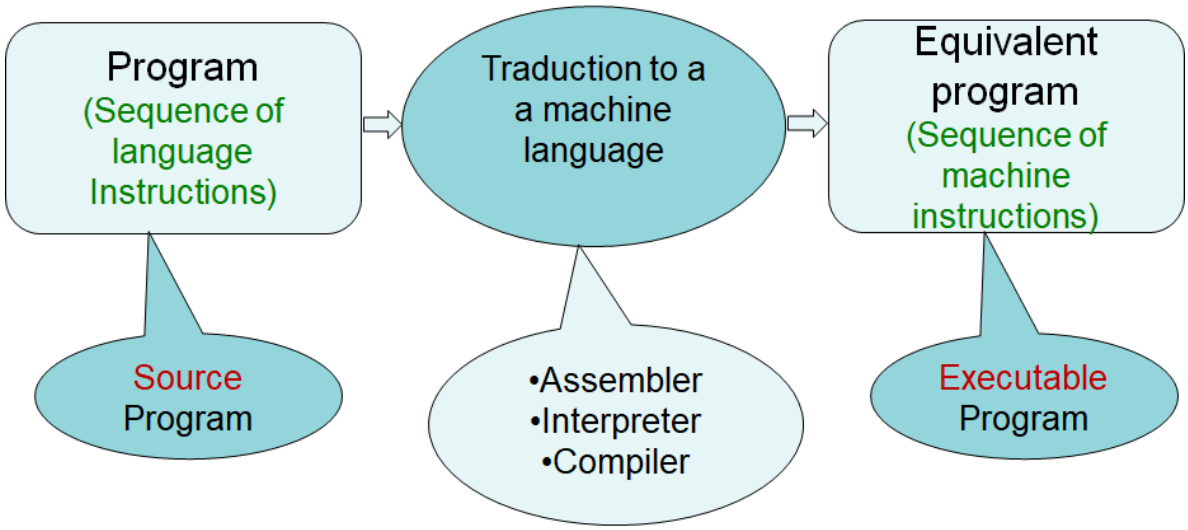


Figure 2.2: Translation of a source program into an executable program.

Which Language?

If we do a quick internet search on programming languages, we can discover hundreds of them! So, which language should you learn?

To make a decision, we need to answer two questions:

1. Which language is in demand on the job market?

2. Which language is the easiest to learn?

To get an idea of what the job market demands, simply browse job search websites (usajobs, careerbuilder, indeed, monster, ...). Here is a selection of the Top 4 best programming languages of 2019 to land a job:

4. *Swift*, for iOS apps.

3. *Java*, for Android apps, back-end web development, ...

2. *Python*, mainly for Data Science and back-end web development.

1. *JavaScript* for almost everything, especially for web development.

The second question focuses on ease of learning. Indeed, the conviction to learn a simple language first is based on the fact that such learning allows you to thoroughly understand and familiarize yourself with the fundamental concepts of programming, without worrying about unnecessary complex elements right from the start.

In our context, the objective of the ADS1 module is algorithm design. Among other things, learning programming is only a secondary objective. Furthermore, the Computer Science bachelor's degree curriculum includes other modules (Programming Tools for Mathematics, Algorithms and Data Structures 3, Object-Oriented Programming, Web Application Development, Software Engineering, Mobile Applications, ...) allowing you to learn other languages later on (C, Matlab, Java, Python, JavaScript, ...).

For these reasons, our choice was primarily focused on ease of learning. Hence the choice of a language that is not only simple but also pedagogical. This is indeed the high-level language Pascal.

Before closing this section, we will try to provide an overview of programming languages through a small example used to display the message "Hello World!".

A- Assembly Language (1950)

Assembly language is a low-level language invented starting in 1950. The binary sequences of machine language are replaced by symbolic notation (mnemonics). The translation of these instructions (mnemonics) is done using an assembler program. The program in Listing 2.1 displays the message "Hello World!" in assembly.

```
1 Cseg segment
2 assume cs:cseg, ds:cseg
3 org 100h
4 main proc
5 jmp debut
6 mess db 'Hello world!$'
7 debut:
```

```
mov dx, offset mess
mov ah, 9
int 21h
ret
main endp
cseg ends
end main
```

Listing 2.1: example of *x*86 assembly language under DOS

B- Procedural Programming Languages

Procedural programming is a recommended programming paradigm for a new developer. This paradigm uses a top-down linear approach and treats data and procedures as two different entities. Based on the concept of procedure calls, procedural programming divides the program into procedures, also called routines or functions, which simply contain a sequence of steps to execute.

In other words, procedural programming involves writing a list of instructions telling the computer what it must do step by step to complete the task.

As an example, we mention a few languages from the procedural family below.

COBOL Language

The *COBOL* language, which stands for **CO**mmon **B**usiness **O**riented **L**anguage, created in 1959, is a business-oriented language.

Listing 2.2 is COBOL code to display the message "Hello World!".

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
DISPLAY "Hello world!".
STOP RUN.
```

Listing 2.2: example of COBOL language

Basic Language

BASIC is an acronym for **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode, created in 1964 to be used by non-computer scientists.

The Basic code in Listing 2.3 displays the message "Hello World!".

```
10 PRINT "Hello world!"
20 END
```

Listing 2.3: example of Basic language

Pascal Language

Pascal is a programming language created in 1969 at the ETH Zurich by N. Wirth. It is designed to teach programming as a science. It is characterized by clear, rigorous syntax that facilitates program structuring. Furthermore, it can be used in industry.

Listing 2.4 illustrates a Pascal program that displays the message "Hello World!".

```
program Bonjour;
begin
  Writeln('Hello world!');
end.
```

Listing 2.4: example of Pascal language

C Language

The *C* language is a low-level language created in 1972 at Bell Labs by *Dennis Ritchie* and *Ken Thompson*.

The C program in Listing 2.5 shows an example of displaying a "Hello World!" message.

```
#include <stdio.h>
int main(int argc, char **argv)
{
printf("Hello world!\n");
return 0;
}
```

Listing 2.5: example of C language

C- Object-Oriented Languages

Object-oriented languages use the Object-Oriented Programming (OOP) paradigm. The OOP paradigm is based on writing, interacting, and reusing software building blocks called objects.

The *Simula*-67 language, in the 1960s, heralded the beginnings of OOP. However, the true definition of OOP's core concepts (object, message, encapsulation, polymorphism, inheritance, overriding, ...) was established with Smalltalk-72 and later Smalltalk-80.

C++ Language

The *C++* language (C with classes) was created in 1983. It allows the use of all existing C libraries. It is widely used in industry.

Listing 2.6 shows a program written in C++ that displays the message "Hello World!".

```cpp
#include <iostream>
int main()
{
std::cout << "Hello world!" << std::endl;
return 0;
}
```

Listing 2.6: example of C++ language

Ada Language

The origins of the *Ada* language date back to the early 1980s. Ada is used to develop scientific and management applications. One of the unique features of the Ada language is the real-time feature built into the language.

Listing 2.7 is Ada code to display the message "Hello World!".

```ada
with Text_IO; use Text_IO;
procedure hello is
begin
   Put_Line("Hello world!");
end hello;
```

Listing 2.7: example of Ada language

Java Language

The *Java* language was created in 1995 by *Sun Microsystems*, which was acquired in 2009 by the *Oracle* corporation that now maintains it.

Java is designed to develop applications that run on any type of processor and operating system. It is well-suited for web applications.

The example of displaying a "Hello World!" message in Listing 2.8 shows Java syntax.

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Listing 2.8: example of Java language

D- Event-Driven Programming Languages

Event-driven programming languages follow a programming paradigm based on the concept of events. The application's components communicate with each other via events (clicking a button, typing in a text box, selecting a radio button, loading a form, closing a window, ...).

Below is a non-exhaustive list of event-driven programming languages: :

- Visual Basic
- Visual C++
- Visual J++
- C# (C Sharp)
- Delphi
- C++ Builder
- J++ Builder

E- Other Languages

Scripting Languages

A scripting language allows for quickly writing small interpreted programs whose main use is the automation of certain small tasks and which have relatively few features.

For example, the following languages are scripting languages:

- JavaScript
- AppleScript
- VBScript

Markup Languages

A markup language is a language used for enriching textual information. It uses syntax units to delimit a sequence of characters or mark a specific position within a character stream. These syntax delimiters are called *tags*.

The following languages are examples of markup languages:

- PS (PostScript)
- LaTeX
- HTML (HyperText Markup Language)
- XML (eXtensible Markup Language)

Web Programming Languages

Web programming languages are used to edit websites that can be either static or dynamic.

In addition to the programming languages already mentioned as web programming languages, we can add the following:

- PHP (Hypertext Preprocessor) is a server-side interpreted scripting language (derived from C and Perl). It is possible to embed PHP code within HTML.
- ASP (Active Server Pages) is an interpreted language defined by Microsoft in 1996 to create dynamic web pages for Windows. It works on the server side. ASP uses ADO technology (ActiveX Data Object) to connect to a database.

2.4.4 An Algorithm for a Problem

So far, we have described the path that takes us from the algorithm to the application while unveiling the concepts of application, source program, programming language, and the translation from a source program to an executable program. To complete the picture, we still need to discuss the approach that allows us to obtain an algorithm for a given problem. This is the purpose of this section.

Problem Definition

The problem is posed by a client called the problem's *instigator* . Generally, the problem is stated without details and without precision. The algorithm designer's role, at this stage, is to specify the problem's details to obtain a precise statement: :

1. Carefully read the problem statement to understand it.
2. Specify all available information (the data).
3. Specify the expected results and their formats, possibly in a formal manner.

For example, if the problem stated by the instigator is "Find the list of divisors of a number", the algorithm designer must redefine the problem by specifying the data (inputs) and the results (outputs) of the problem to obtain a precise statement: "Given an integer N, construct a computerized solution that allows us to obtain the list of its divisors".

Problem Analysis

Problem analysis involves finding the means (algorithm) to go from the data to the results. In some cases, it may be necessary to conduct a theoretical study.

In the literature, there are several approaches to analyzing a problem. Bottom-up analysis and top-down analysis are among them.

If we return to the problem of calculating the divisors of an integer, we can propose the following analysis result (algorithm):

- Successively divide N by $i = 1, 2, 3, \ldots, N/2$
  - each time the remainder of the division of N by i is equal to 0 (then i is a divisor)
    - print i.

In a more sophisticated manner, we can propose Algorithm *divisors*.

Algorithm divisors

```
Variable N, i : Integer
begin
    Read(N)
    for i from 1 to (N div 2) do
    begin
        if (N Mod i) = 0 then
        begin
            Write (i)
        end
    end

end
```

2.4.5 Steps (from problem to results)

This section aims to recapitulate the process of moving from the problem to the results. Figure 2.3 illustrates such an approach.

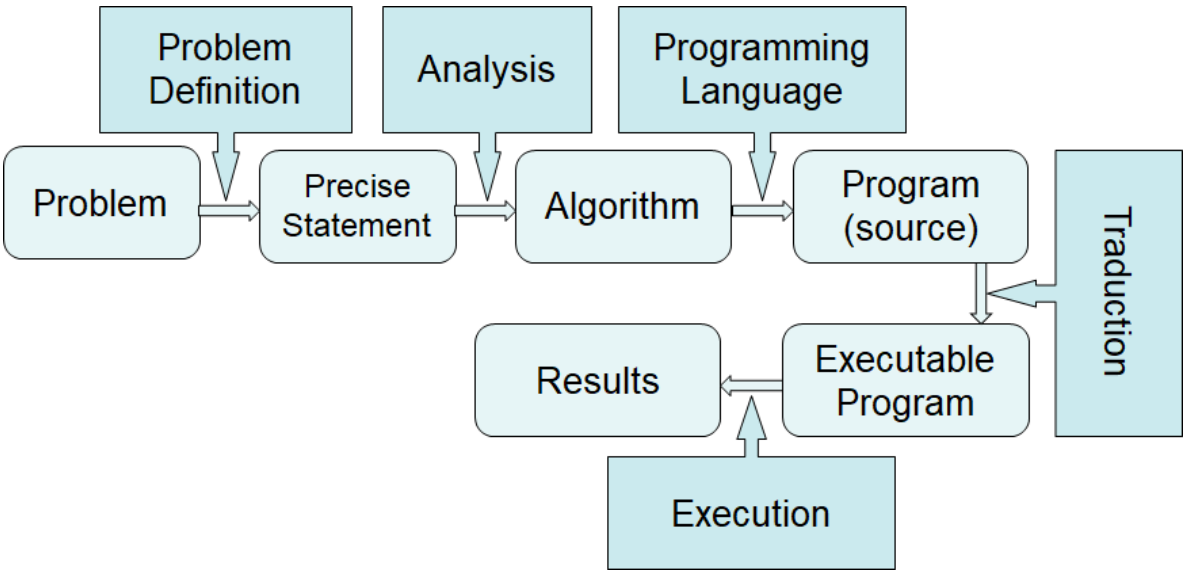


Figure 2.3: Steps for implementing an application.

The first steps, problem definition and its analysis, constitute the design phase, which is done without using a machine.

Then, to materialize the designed algorithm, we must translate it into a programming language. For example, Algorithm *divisors* is translated into the Pascal language to give a source program (div.pas) as shown in Figure 2.4.

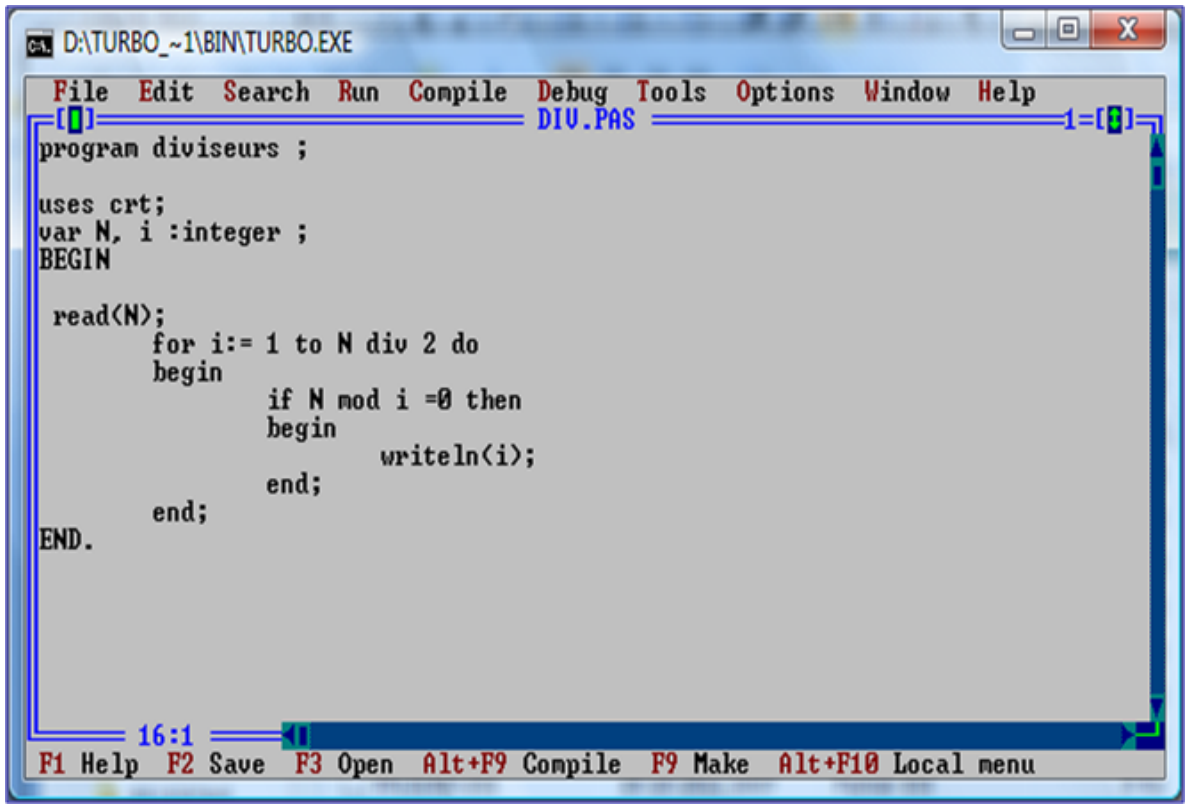


Figure 2.4: Source program (Pascal language) associated with Algorithm divisors.

A second translation is necessary; this time, the translator is a Pascal compiler used primarily to read and process the source program to convert it into machine language directly understandable by the machine. The resulting file (executable program) can be launched like any other machine language program.

After obtaining the executable program, one must complete the application implementation process with the debugging phase, which initially consists of executing the obtained code and viewing the results. Figure 2.5 shows the result of executing the machine code for the case of calculating the divisors of an integer.

Once the results are obtained, they must be compared with the expected ones. If there is a match, then the process is complete; otherwise, one must proceed to the error correction stage, which spans from correcting the algorithm's translation back to the problem analysis.

The correction phase can be expressed using the fragment of Algorithm *correction*.

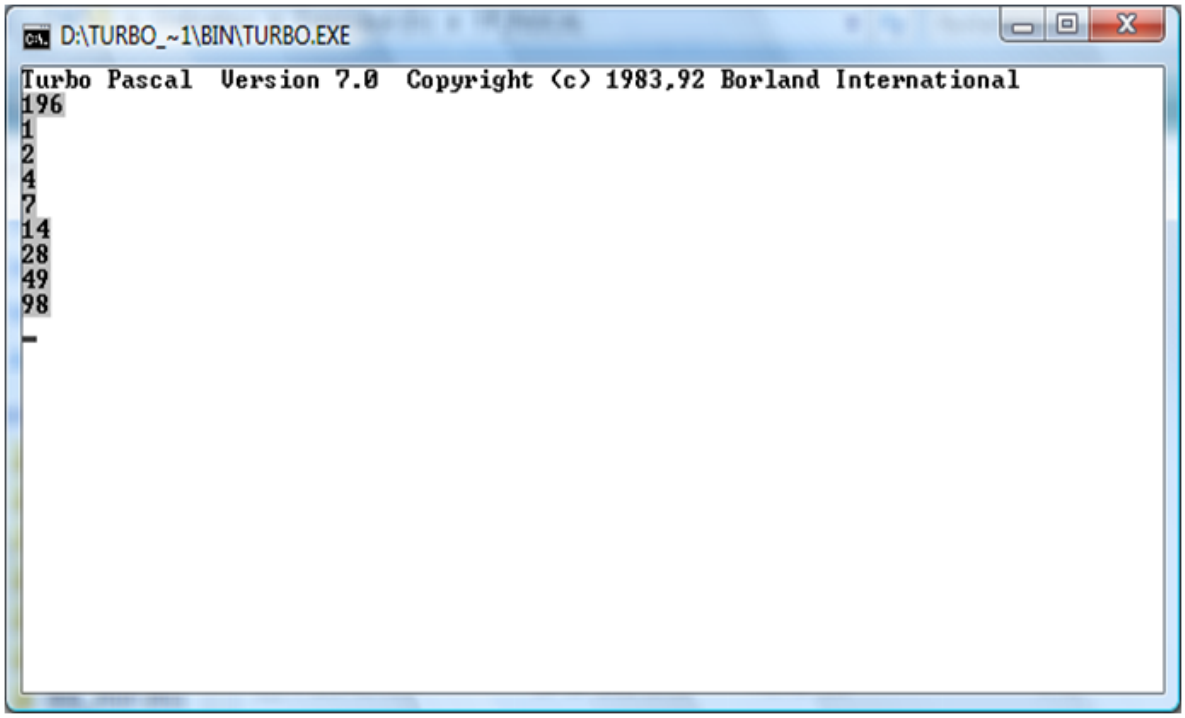


Figure 2.5: Execution of the executable code associated with the program calculating the divisors of an integer.

Algorithm correction

```
begin
    if Obtained results = Expected results then
    begin
        Execution completed
    end
    else
    begin
        Logical errors exist
        Review the translation of the algorithm
        or
        Review the problem analysis
    end
end
```

2.5 Exercises

Exercise 2.1 Having a processor, which is actually an automaton composed of a mobile arm supporting a pen and a tablet on which squared paper is placed, and knowing that this automaton only understands and executes two types of

commands:

- lower or raise the pen,
- move the pen in a given direction and over a delimited distance.

(Each primitive action is composed of two characters: the first character indicates whether the pen is lowered or not; it takes the value **A** if the pen needs to be lowered or the value **L** if the pen needs to be raised; the second character indicates the direction and the movement length of the pen as shown in Figure 2.6:
Example: if we want to draw a square, the sequence of commands is: (A3, A5, A7, A1).
You are asked to analyze and then write the sequence of commands allowing the automaton to execute the drawing in Figure 2.7, but they must be performed without raising the pen, without retracing an already drawn line, and without crossing it.
NOTE: The arrow indicates the initial position of the mobile arm.
What is the problem posed by this formalism? How can it be improved?

Indications When discussing the formalism problem proposed by this exercise, think about the properties of algorithmic language.

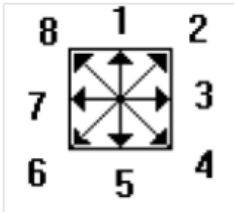


Figure 2.6: Pen movement direction.

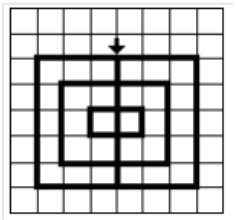


Figure 2.7: Drawing to be executed.

Exercise 2.2 Two farmers fill a 24-liter milk churn every time after milking their cows; however, they have two churns with capacities of 15 liters and 9 liters respectively. Can you help them by providing the solution that allows them to divide this milk into 2 equal parts?
Once the analysis is done, you are asked to construct the corresponding algorithm using the following formalism:
Let A, B, and C be the churns with capacities of 24, 15, and 9 liters respectively, the primitive actions will be of the form $A \longrightarrow B$:

- For example, if we take a part of A to fill B, we will denote this by the primitive action $A \longrightarrow B^*$;
- on the other hand, if we pour A into B without filling it, we will denote

$A \longrightarrow B$.

Indications Try to find several solutions and compare them in terms of the number of steps.

Exercise 2.3 Some time later, they want to offer a relative 4 liters of olive oil, but they only have 2 pots with capacities of 5 and 3 liters respectively. How will they do it? (Use the same formalism as in Exercise 2.2 to write the algorithm).

Exercise 2.4 How can the triangle in Figure 2.8 be filled in such a way that the number written in a box is equal to the sum of the 2 numbers in the 2 boxes below it?

Indications Formalize the problem mathematically as a system of equations.

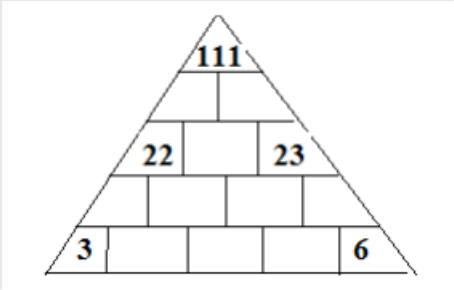


Figure 2.8: Triangle to be filled.

Exercise 2.5 You have 12 apparently identical coins, but you are sure that one of them is fake because it does not have the same weight as all the others. How will you find the fake coin and know if it is heavier or lighter, using a two-pan balance scale and a maximum of 3 weighings? Construct the analysis of this problem.

Exercise 2.6 We have 10 stacks of 10 identical-looking coins, but we know that one stack is composed entirely of fake coins. Knowing that a real coin weighs 5 grams and a fake coin weighs 6 grams. Provide the analysis that allows us to find the stack of fake coins in a single weighing and then write the corresponding algorithm.

Exercise 2.7 Three people, each wearing a hat on their head, are lined up one behind the other so that each can only see in front of them. Knowing that initially there are 5 hats, 3 red and 2 green, and that each person can see the

remaining hats, how will the person at the front know the color of their hat if they are only allowed, if they wish, to ask a single question to the other 2 people concerning them personally? (Provide the analysis)

Exercise 2.8 How does $2036 : 4 = 10$?

Exercise 2.9 Consider the sequence: 1, 11, 21, 1112, 3112, ... What is the next element?

Exercise 2.10 How can you get 1000 from an addition that contains only 8s?

3. Presentation of the Algorithmic Formalism

This chapter aims to define the algorithmic formalism *used to express proposed solutions to analyzed problems in a clear and unambiguous manner.*

We first begin by presenting the structure of an algorithm, which includes a header, a declaration part, and an algorithm body.

Next, we address the processing part through the various control structures.

To complete the chapter, we discuss the data part through the introduction of the concept of a variable and the various associated actions, as well as a more or less detailed description of what is called the environment of an algorithm (declaration part).

3.1 Preamble

We begin this chapter with a small exercise:

Exercise 3.1 Express in natural language the solution to the quadratic equation $ax^2 + bx + c = 0$.

If we ask the students of an algorithms class to propose a solution to Exercise 3.1, we might get as many proposals (in terms of expression) as there are students.

By way of illustration, we propose two solutions:

Proposed Solution 1

$a, b, c \longleftarrow$ the data
Calculate the discriminant

Case where it is> 0 : $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$
Case where it is zero : $x_1 = x_2 = -\frac{b}{2a}$
Case where it is negative : no solution

Proposed Solution 2

1. Input the values of (a, b, c)
2. calculate $\Delta = b^2 - 4ac$
3. If $\Delta < 0$ then no roots in $\mathbb{R}$
   If $\Delta = 0$ then a double root
   If $\Delta > 0$ then two roots

It is not difficult to conclude that writing an algorithmic solution to a given problem in natural language is often ambiguous. This ambiguity generally leads to different interpretations. Hence the necessity of using a common language called an *algorithmic formalism* (Definition 3.1).

Définition 3.1 An algorithmic formalism is a set of conventions (rules) for describing an algorithm.

The purpose of this chapter is to present this algorithmic formalism in detail.

3.2 Structure of an Algorithm

An algorithm is composed of three main parts as illustrated by Figure 3.1. The *header* of an algorithm is used to give a name to the algorithm using the reserved word **Algorithm** with the following syntax:

```
Algorithm algorithm_name
```

The header part has no influence on the execution of the algorithm. The name serves to identify the algorithm among others. As with the identifiers of other entities in the algorithm (constants, types, variables, procedures, functions, ...), the name of the algorithm is a sequence of alphanumeric characters whose first character must obligatorily be a letter. The only special character allowed is the underscore character `_`. Like reserved words, identifiers can be written interchangeably in uppercase or lowercase and can be of any length.

The declaration part or environment is used to declare all the objects (constants, types, variables, ...) or modules (procedures, functions) manipulated in the body part of the algorithm.

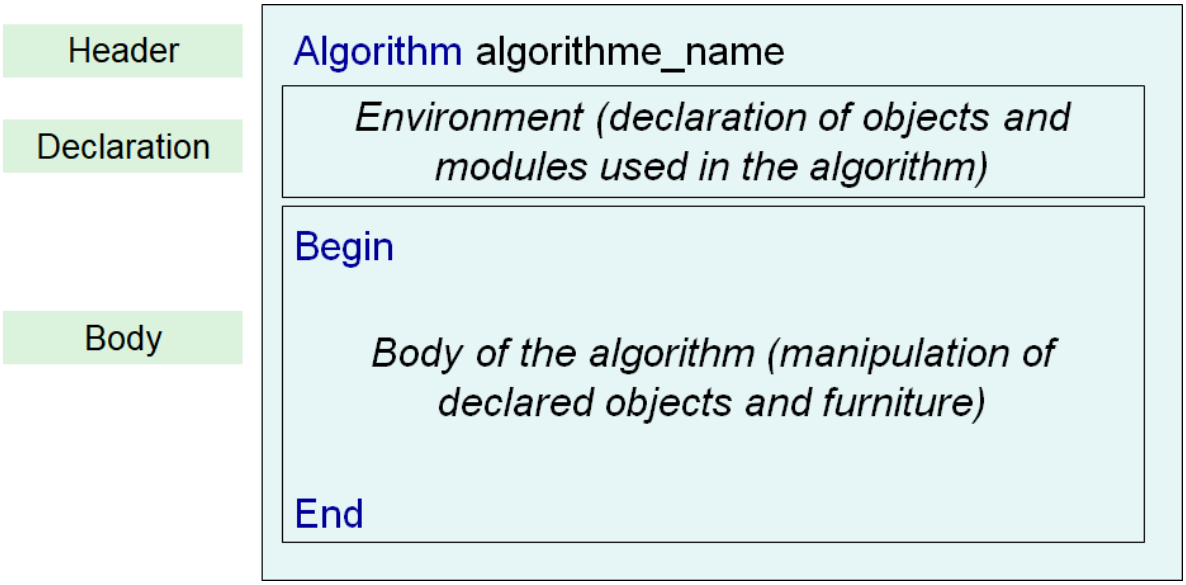


Figure 3.1: Structure of an algorithm

The body of the algorithm contains the actions that the algorithm must perform to accomplish the requested work. This part is delimited by the reserved words **begin** and **end**. It contains control structures that describe the sequence of actions.

3.3 Control Structures

The description of the sequence of actions within the body of the algorithm is done using control structures that allow for sequential, conditional, or repetitive processing.

3.3.1 Sequencing

Sequencing is the simplest control structure. It allows for the sequential execution of a set of actions according to the following syntax:

Action 1
Action 2
Action 3
...
Action n

The execution of the sequence is performed linearly according to the order of appearance of the actions in the algorithm. In other words, we first start by executing *Action* 1, then *Action* 2, up to *Action n*.

Exemple 3.1 Sequencing

```
Read (A)
Read(B)
C ← A*B
Write (C)
```

Example 3.1 illustrates this principle of sequencing. We start by reading variable A, then reading variable B, after which the result of the multiplication of A and B is assigned to variable C. Finally, we output the content of variable C.

3.3.2 Conditional Structure

The *conditional* structure allows choosing a process based on whether a condition is met or not. Thus, the conditional structure authorizes us to design an algorithm that will skip executing certain actions.

The syntax is as follows:

```
if condition then
begin
 | block
end
```

The *condition* is a boolean (logical) expression that evaluates to the boolean value **true** or **false**. A *block* consists of one or more actions. It starts with the reserved word **begin** and ends with **end**. In the case where a block consists of a single action, the **begin** and **end** delimiters of the block are optional.

The execution of the conditional proceeds as follows:

- evaluation of the *condition*;
- if the *condition* is true, the block is executed, then control passes to the statement following the conditional;
- if the *condition* is false, control passes to the statement following the conditional, without executing the block.

Exemple 3.2 conditional with a single-action block

```
Read (A)
positive ← A
if (positive < 0) then
begin
 | positive ← −1 ∗ positive
end
Write (‘the positive value is : ‘ , positive)
```

Since the conditional block in Example 3.2 contains a single action, we can omit the reserved words **begin** and **end** to obtain the following code fragment:

```
Read (A)
positive ← A
if (positive < 0) then
    positive ← −1 ∗ positive
Write (‘the positive value is : ‘ , positive)
```

In the case of Example 3.3, it is not possible to omit **begin** and **end** from the block.

Exemple 3.3 conditional with a multiple-action block

```
Read (A)
positive ← A
if (positive < 0) then
begin
    positive ← −1 ∗ positive
    Write (‘the positive value is : ‘ , positive)
end
```

3.3.3 Alternative Structure

The *alternative* structure is another form of the conditional structure. It allows expressing a choice between two processes based on the value of a condition.

The syntax for this structure is:

```
if condition then
begin
    block 1          /* block 1 executed if condition equals True */
end
else
begin
    block 2         /* block 2 executed if condition equals False */
end
```

If the *condition* is met, then *block 1* is executed; if it is not met, *block 2* is executed. In all cases, execution will continue after the alternative structure.

Example 3.4 shows the use of the alternative structure.

Exemple 3.4 alternative

```
Read (A)
Read (B)
if A > B then
begin
    max ← A
    A ← A + B
end
else
begin
    max ← B
    B ← A + B
end
Write ('the maximum is :', max)
```

It should be noted that indentation (spacing) when writing an algorithm is necessary for proper readability. A well-presented algorithm shows that its execution has been understood.

The rule for good presentation is quite simple. It can be introduced through the following two conditionals:

```
if start of block encountered then
begin
    shift (by one tab) to the right
end
if end of block encountered then
begin
    shift (by one tab) to the left
end
```

By way of illustration, it is obvious to distinguish the action blocks in terms of visibility between the two Algorithms *with_indentation* and *without_indentation*, which are in fact identical:

Algorithm with_indentation

```
begin
    Read (A)
    Read (B)
    if (A > B) then
        max ← A
    else
        max ← B
    Write ('the maximum is :', max)
end
```

Algorithm without_indentation

begin
Read (A)
Read (B)
if $(A > B)$ **then**
$max \leftarrow A$
else
$max \leftarrow B$
Write ('the maximum is :', max)
end

3.3.4 Nested Conditionals

There are situations where we need a part of a conditional structure to, in turn, contain another conditional structure. In such a situation, we speak of *nesting* conditional structures within one another.

It should be noted that this nesting property is not limited only to conditional structures, but it can be generalized to all the control structures we encounter in the algorithmic formalism.

Exemple 3.5 Write an algorithm body that reads a grade and displays the associated comment as follows:

- grade from 0 to 8 inclusive : insufficient
- grade from 8 to 12 inclusive : average
- grade from 12 to 16 inclusive : good
- grade from 16 to 20 inclusive : very good

A possible solution to the problem in Example 3.5 is to make a succession of choices:

```
Read(grade)
if (grade ≤ 8) then
    Write ('insufficient')
if ((grade > 8) and (grade ≤ 12)) then
    Write ('average')
if ((grade > 12) and (grade ≤ 16)) then
    Write ('good')
if ((grade > 16) and (grade ≤ 20)) then
    Write ('very good')
```

In this solution, for each grade entered by the user, four tests are performed.
We can propose another solution that uses nesting:

```
Read(grade)
if (grade ≤ 8) then
    Write ('insufficient')
else   if (grade ≤ 12) then
            Write ('average')
        else if (grade ≤ 16) then
                Write ('good')
            else if (grade ≤ 20) then
                    Write ('very good')
```

In the **else** blocks, it is unnecessary to do the tests $grade > 8$, $grade > 12$, and $grade > 16$ because we are certain that the grade is strictly greater than 8, 12, and 16 respectively.

In this solution, we only have to perform four tests in the worst-case scenario.

3.3.5 Repetitive Structures

We have already seen in Section 2.2 that one of the properties of an algorithm is that it is generally repetitive.

The *repetitive structure*, also called a *loop* , allows a block of actions to be executed multiple consecutive times. It has three forms: :

- while ;
- repeat ;
- for.

While Structure

The *while* structure is the most classic repetition action; its syntax is as follows:

```
while condition do
begin
 | block
end
```

It allows the execution of an action block multiple times as long as the (continuity) condition is met. The execution proceeds as follows:

1. evaluate the *condition* ;
2. if the *condition* is true, the action *block* is executed and we restart at step 1.
3. if the *condition* is false, we exit the loop and execute what follows it.

Exemple 3.6 (while loop)

```
counter ← 1
while (counter <= 4 ) do
begin
    Write (counter)
    counter ← counter +1
end
```

Table 3.1 shows the execution trace of the algorithm fragment from Example 3.6:

Table 3.1: Execution trace of a *while* structure

counter	**condition (counter** <= 4	**continue (yes/no)**
1	Initialization (before entering the loop)	
1	(counter <= 4) = true	Yes (enter the loop)
2	(counter <= 4) = true	Yes
3	(counter <= 4) = true	Yes
4	(counter <= 4) = true	Yes
5	(counter <= 4) = false	No (exit the loop)

We can see that the loop can be executed 0, 1, or multiple times, or even an infinite number of times. Therefore, to write a loop, three guidelines must be carefully followed:

1. Correctly initialize the condition variable(s) before entering the loop (**counter** ← **1**).

2. Write a correct condition (**counter** <= **4**).

3. Modify the condition variable(s) (**counter** ← **counter + 1**).

The situation where the designer does not respect these guidelines is summarized by the following conditional:

```
if (guidelines not respected ) then
begin
    risk of not entering the loop
    Or
    of not being able to exit it                    /* infinite loop */
end
```

Since there is no clearly defined recipe to follow the rules for writing a loop, it follows that we can obtain equivalent algorithms with different loops:

```
counter ← 1
while (counter <= 4 ) do
begin
    Write (counter)
    counter ← counter +1
end
```

```
counter ← 1
while (counter <> 5 ) do
begin
    Write (counter)
    counter ← counter +1
end
```

```
counter ← 1
while (counter < 5 ) do
begin
    Write (counter)
    counter ← counter +1
end
```

```
counter ← 0
while (counter < 4 ) do
begin
    counter ← counter +1
    Write (counter)
end
```

Repeat Structure

The *repeat* structure is another form of repetitive structure. It is written as follows:

```
repeat
    block
until condition
```

The execution of the *repeat* loop can be described as follows:

1. execute the *action block* ;
2. evaluate the (stopping) *condition* ;
3. if the condition is false, restart at step 1 ;
4. if the condition is true, exit the loop and execute what follows it.

Example 3.7 illustrates the use of the repeat structure.

Exemple 3.7 (repeat loop)

counter ← 1
repeat
  Write (counter)
  counter ← counter +1
until *(counter* > 4*)*

It is important to remember that:

- We use the *while* and *repeat* forms when the number of iterations is unknown.
- The number of iterations is at least one for the *repeat* form, whereas it can be zero for the *while* form.

For Structure

The *for* loop is used to execute an action block a certain number of times known in advance.

The *for* loop has the following syntax:

for *control_variable* ***from*** *initial_value* ***to*** *final_value* **do**
begin
  block
end

The *block* is executed every time the value of the *control_variable* is between *initial_value* and *final_value*.

The execution of the *for* loop can be described as follows:

1. initialize the *control_variable* with the *initial_value* ;
2. test whether the *control_variable* is within the range [*initial_value*, *final_value*] or not ;
3. if the test is true then
   - execute the action *block* of the *for* loop ;
   - increment the *control_variable* (with each pass, the *control_variable* automatically moves to the next value in its domain) ;
   - restart at step 2 ;
4. if the test is false then exit the loop and execute the action that follows the end of the *for block*.

Exemple 3.8 (for loop)

```
for counter from 1 to 5 do
begin
  | Write (counter * 5)
end
```

The *for* loop in Example 3.8 outputs the values 5, 10, 15, 20, and 25.

It should be noted that the progression of the *control_variable* can be decreasing by using the following syntax:

```
for control_variable from final_value down to initial_value (step =
  decrement_value) do
begin
  | block
end
```

In this case, the *control_variable* is decremented after each pass.

Furthermore, the *repeat* and *for* loops do not increase the expressive power of the algorithmic language regarding repetitive structures. The *while* loop is sufficient to express all repetitive processes; in fact, it is the most natural form and the one systematically used in algorithms. Nevertheless, the other forms (*repeat* and *for*) can be used to lighten the writing of algorithms.

3.4 Data Part

In Section 3.3, we introduced the control structures used to describe the sequence of actions. These actions manipulate objects that contain data.

In our context, we distinguish two types of objects:

- objects that cannot change during the execution of an algorithm: **Constants** ;
- objects that can change during the execution of an algorithm: **Variables** (slates).

In the remainder of this section, we focus on the concept of *variable* as well as the associated simple actions (assignment, mathematical operations, inputs, and outputs).

3.4.1 Variables

Définition 3.2 A variable designates a memory location that allows a value to be stored. It is defined by:

- a unique *name* that designates it. The name of a variable must respect the properties of identifiers stated in Section 3.2 ;
- a *type* (domain of definition) indicating the set of values the variable can

take ;

- a *value* assigned and modified during the execution of the algorithm.

Definition 3.2 introduces the concept of a *variable*, which is comparable to that of a *slate* having the ability to contain information and be continuously modified.

3.4.2 Assignment

Définition 3.3 Assignment consists of attributing (assigning) a value to a variable. It is denoted by the sign $\leftarrow$.

The value assigned to a variable can be:

- a constant: $X \leftarrow 0$, $number \leftarrow 7$;
- the value of another variable: $X \leftarrow Y$, $Number \leftarrow Result$;
- the value of an expression: $X \leftarrow Y + 1$, $Number \leftarrow Result * 2$.

The action $X \leftarrow 0$ is read as: the variable X receives the value 0. If X had a value previously, this old value will be overwritten by the new value 0.

In the assignment operation, care must be taken that the value, the content of the variable, or the result of the expression on the right of the assignment sign is of the same type or a type compatible with that of the variable on the left.

3.4.3 Expressions

Définition 3.4 An expression is a sequence of operands (variables, constants, functions) combined by a set of operators, possibly with a set of opening and closing parentheses.

Definition 3.4 is illustrated using Figure 3.2.

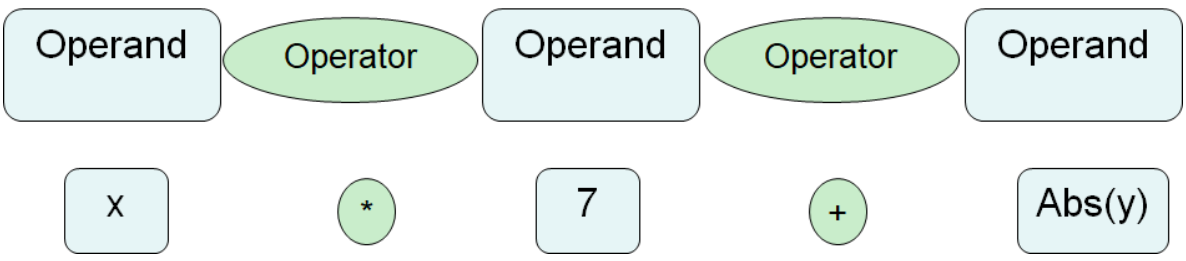


Figure 3.2: General form of an expression.

Types of Expressions

In algorithmic language and programming languages, there are several types of expressions, namely, arithmetic, logical, relational, string, set, and mixed.

In this algorithmic learning phase, we limit ourselves to arithmetic, logical, relational, and mixed expressions as shown in Figure 3.3. String expressions are covered in Chapter 4 of this handout, while set expressions will be studied in Chapter 5.

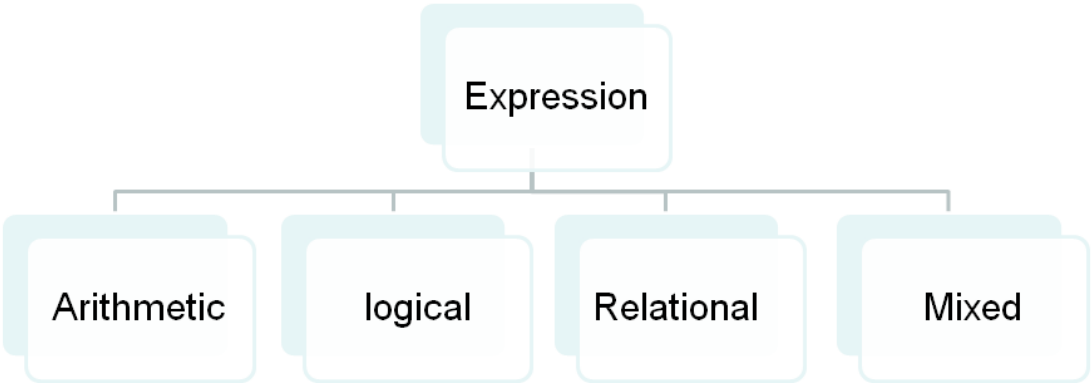


Figure 3.3: Types of expressions.

Arithmetic Expressions

Tables 3.2 and 3.3 summarize the operand types and the results of *binary* and *unary* arithmetic operations. A binary operation uses an operator that implements two operands. Operators with one operand are called unary operators.

Table 3.2: Binary arithmetic operations

Operator	Operation	Operand Types	Result Type
+	addition	integer/real	integer/real
−	subtraction	integer/real	integer/real
*	multiplication	integer/real	integer/real
/	division	integer/real	real/real
div	integer division	integer	integer
mod	modulo (remainder)	integer	integer

- If the operands of a binary operator are both of integer type, the result is of integer type.
- If at least one of the operands of the binary operators $+, -, *$ is of real type, the result is of real type.

Table 3.3: Unary arithmetic operations

Operator	Operation	Operand Types	Result Type
+	sign identity	integer/real	integer/real
−	sign change	integer/real	integer/real

- The value X/Y (with $Y \neq 0$) is always real regardless of the operand types. The content of Z after the execution of the action $Z \leftarrow 26/4$ is the real value 6.5.
- The value of I **div** $J = \lfloor I/J \rfloor$ (with $J \neq 0$) is the lower integer part. The content of variable L after the execution of the action $L \leftarrow 26$**div**4 is the integer value 6.
- The **mod** operator returns the remainder of the division of its two operands: I **mod** $J = I - (I$ **div** $J) * J$ (with $J \neq 0$).
- The sign of the result of I **mod** J is that of I.
- The result type of a unary operator is identical to that of the operand.

Logical Expressions

Table 3.4 shows the operand types and the results of logical operations.

Table 3.4: Logical operations

Operator	Operation	Operand Types	Result Type
not	logical negation (unary)	boolean	boolean
and	logical and (conjunction)	boolean	boolean
or	logical or (disjunction)	boolean	boolean

The result type of logical operations is governed by conventional boolean logic. Example 3.9 suggests some forms of using boolean expressions:

Exemple 3.9
$X \leftarrow True$, the expression here is a constant (the True value).
$Y \leftarrow not\ X$, the expression is a simple logical negation.
$Z \leftarrow (X\ and\ Y)\ or\ (not\ X)$, the expression consists of a conjunction, a disjunction, and a negation.

It should be noted that in the algorithmic language, the evaluation of logical expressions is done in a complete manner. In other words, each operand of the logical expressions is evaluated even if the result of the resulting expression is already known.

However, in most programming languages, it is possible to choose between two evaluation models for logical expressions: *complete* evaluation and *optimized* evaluation, sometimes called *short-circuit* evaluation.

For the optimized model, evaluation is done from left to right and stops as soon as the result becomes evident. That is to say, the evaluation of an operand is only done if its value is essential for knowing the result of the expression.

Relational Expressions

Table 3.5 shows the operand types and the results of relational operations.

Table 3.5: Relational operations

Operator	Operation	Operand Types	Result Type
=	equal	simple	boolean
>	greater than	simple	boolean
>=	greater than or equal to	simple	boolean
<	less than	simple	boolean
<=	less than or equal to	simple	boolean
<>	not equal	simple	boolean

By simple type, we mean the types maintained by the arithmetic and logical expressions already studied. For more details, see Section 3.5.2.

Mixed Expressions

A mixed expression is an expression that combines arbitrary operands with arbitrary operators.

Exemple 3.10
$x/y < z$ or x div $y = 0$ or a and not b and c mod $d > e$

The expression in Example 3.10 is complex; it can lead to confusion during its evaluation. A solution to this problem can involve two tricks:

1. Defining a hierarchy (priority) of operators.

2. Using parentheses.

Hierarchy (Priority) of Operators

The hierarchy of operators is defined in Table 3.6.

In addition to the operator priority defined in Table 3.6, it is necessary to respect the following rules when evaluating an expression:

Table 3.6: Hierarchy (priority) of operators

Operators	**Priority**	**Category**
not	First (high)	unary operator
*, /, div, mod, and	second	multiplication operators
+, -, or	third	addition operators
=, <>, <, >, <=, >=	fourth (low)	relational operators

1. An operand placed between two operators of different priorities will be linked to the operator with the highest priority. For example, in the expression $X*\mathbf{a}+b$, the operand **a** will be linked to the $*$ operator.

2. An operand placed between two operators of the same priority will be linked to the operator on the left. When evaluating the expression $X+\mathbf{b}-c$, the operand **b** will be linked to the + operator.

3. Expressions contained within parentheses are evaluated separately, and then their result is treated as a single operand. To evaluate the expression $(X+b)*c$, we start with the evaluation of the addition $X+b$, then the multiplication $(*)$ of c by the result of $X+b$.

Exemple 3.11
Evaluate the mixed expression $a + b * c / d \breve{} e * f + g - 10$

The evaluation of the expression in Example 3.11 is performed by applying the operator priority rules as illustrated in Figure 3.4.

Use of Parentheses

The use of parentheses allows overriding the aforementioned operator priority rules. Expressions inside parentheses are evaluated first, starting with the innermost parentheses. In this way, parentheses can change the evaluation order of the expression.

Moreover, parentheses can also be used to ensure better readability of an expression. An excessive number of parentheses can be used provided the evaluation of the expression remains valid, as illustrated in Example 3.12.

Exemple 3.12
Let the mathematical expression be $\frac{a*b*c}{\frac{c*d+5}{f-2}+g}$.
Using the algorithmic formalism, this expression can be written as follows: : $a*b*c/((c*d+5)/(f-2)+g)$.
For good readability of the expression, an excessive number of parentheses can be used: $(a*b*c)/(((c*d+5)/(f-2))+g)$.

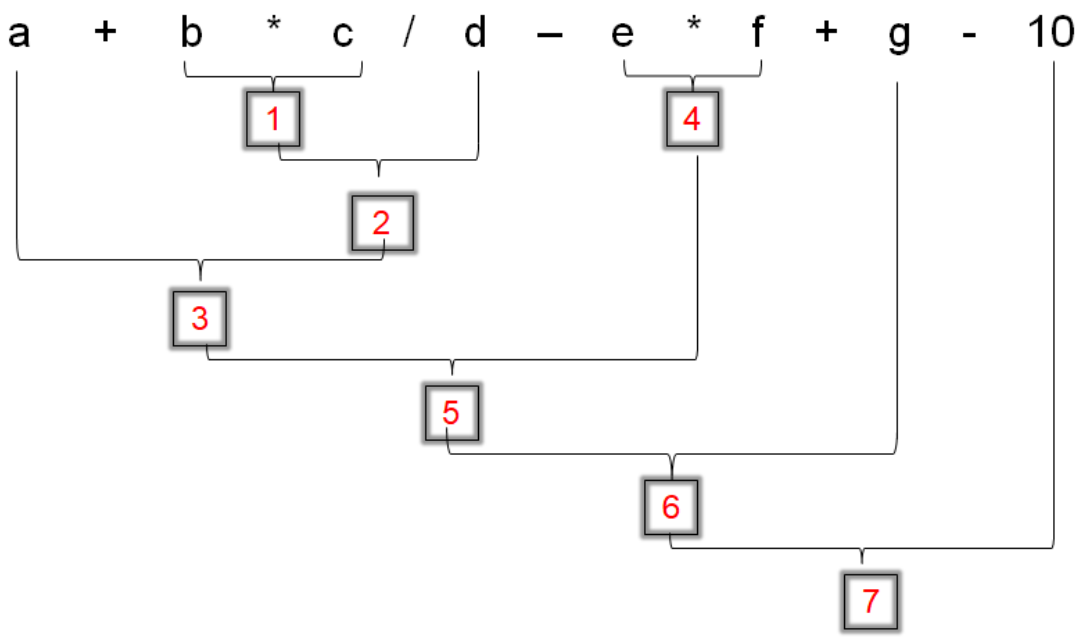


Figure 3.4: Evaluation of a mixed expression.

3.4.4 Read Action

The *read* action allows data coming from the outside to be introduced into variables. More precisely, this action fetches values from a standard input device (the keyboard) and assigns them to variables, generally called *parameters* according to the following syntax:

Read ($P1, P2, \ldots$)

The execution of this action consists of:

1. asking the user to enter the values on the input device (keyboard) ;
2. modifying the variables (parameters) passed in parentheses in order, taking into account type compatibility.

This read action is equivalent to:

$P1 \leftarrow$ first data
$P2 \leftarrow$ second data
...

Note that the read action can only apply to variables (data containers). *Reading* an expression or a constant would make no sense.

3.4.5 Write Action

The *write* action allows the results of an algorithm to be output. More precisely, this action allows the values of one or more expressions to be displayed on a standard output device (screen).

The syntax of the write action is as follows:

Write ($E1$, $E2$, ...)

---

the expressions E, $E2$, ... are evaluated and the results are displayed on the screen.

An expression Ei can be:

- A variable.
- A literal: string of characters enclosed in single quotes.
- An expression.

Exemple 3.13
Write ('the results of the equation are ', X1, ' and ', (-b-sqrt(delta))/2*a)).

In Example 3.13, the first and third parameters are strings of characters. The second parameter is a variable. The fourth parameter is an arithmetic expression.

If the single quote character is a character to be displayed, as in the string *'the equation's results are '*, it must be incorporated into the string by doubling the quotes as follows *'the equation''s results are '*.

3.5 Algorithm Environment

All objects (data) manipulated in the body of the algorithm and optionally their types must be declared in the environment part of an algorithm.

In this section, we focus on the declaration of constants, types, and variables. The declaration of modules will be covered in the book *Algorithmics and Data Structures: Part 2*.

3.5.1 Declaration of Constants

We begin with the definition (Definition 3.5) of constants, then we move on to their declaration.

Définition 3.5 A constant is a particular elementary object whose value cannot change during the execution of the algorithm.

The type of the constant is implied. It can be an integer, a real, a boolean, a character, or a string.

The declaration of a constant is done as follows:

Constant constant_name = value

Exemple 3.14

Constant Pi = 3.14
Constant message = 'insufficient memory !'
Constant max = 100
Constant found = True

In Example 3.14, we used the reserved word **Constant** for each declaration. However, it is possible to use this reserved word only once, applying it to all the constants that follow (Example 3.15):

Exemple 3.15

Constant Pi = 3.14
message = 'insufficient memory !'
max = 100
found = True

Note that every time the constant is encountered in an algorithm, it is replaced by its value.

3.5.2 Declaration of Types

Définition 3.6 A type is a domain of definition that determines the set of values an object can take as well as a set of operators on these values.

To introduce data types, we adopt the taxonomy of the Pascal language shown in Figure 3.5.

In this section, we focus on standard types according to the classification in Figure 3.6.

Structured and non-standard types are covered in Chapter 5.

Before diving into the details, it should be remembered that all simple types other than the real type are scalar types. The possible values of a scalar type form an ordered set, and each value is associated with a rank. In all scalar types, every

element other than the first has a predecessor, and every element other than the last has a successor.

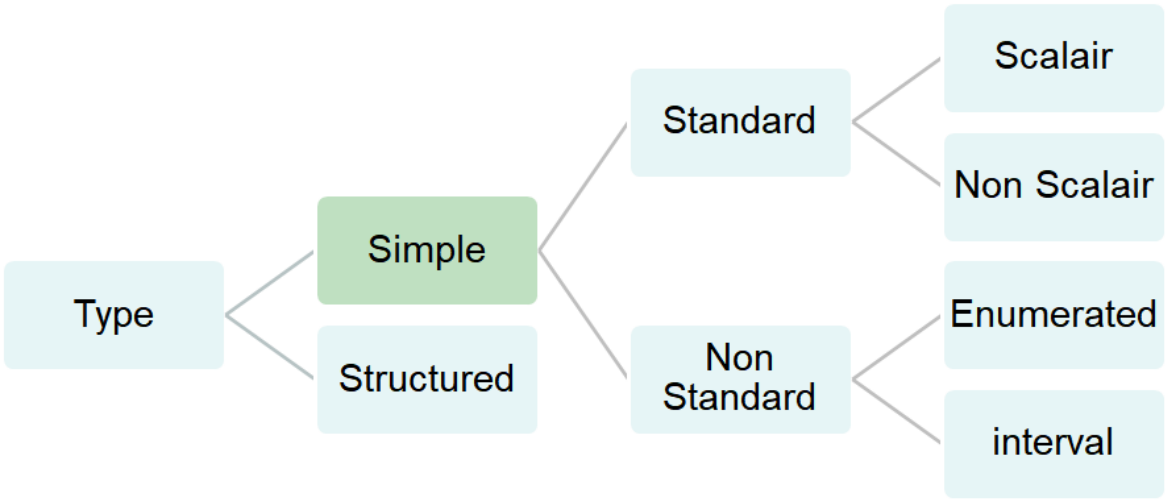


Figure 3.5: Taxonomy of types.

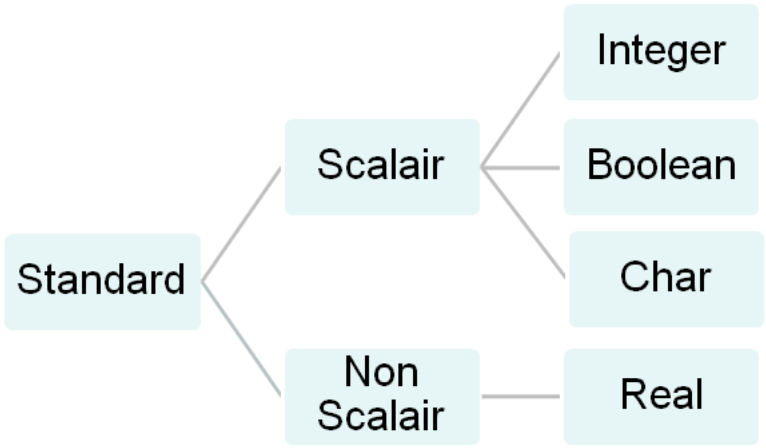


Figure 3.6: Taxonomy of standard types.

Integer Type

Définition 3.7 The *integer* type is the set of relative integers; however, it must be pointed out that while this set is infinite in mathematics, on a computer the values are limited by the length of the machine words.

For example, Table 3.7 shows the ranges of the *Integer* and *Word* types in the Pascal language.

Table 3.7: Integer and Word types in the Pascal language.

Type	Range	Format
Integer	−32768..32767	16 signed bits
Word	0..65536	16 unsigned bits

Boolean (Logical) Type

Définition 3.8 The boolean type constitutes the set of values {true, false}.

This type does not exist in all programming languages because it is not as essential as the other types.

Character Type

Définition 3.9 The character type is the set of alphanumeric characters (alphabetic and numeric, special signs, and the blank (or space) character). It corresponds to a single character.

More details regarding the character type are discussed in Section 4.7.1.

Real Type

Définition 3.10 The real type is the set of numbers having a fractional part. This set is also limited, but the limits are wider and depend on the internal representation (in the machine).

Table 3.8 shows the ranges of the *Real* and *Double* types in the Pascal language.

Table 3.8: Real and Double types in the Pascal language.

Type	Range	Size in bytes
Real	$2.9x10^{-39}..1.7x10^{38}$	6
Double	$5.0x10^{-324}..1.7x10^{308}$	8

Indications Standard types are predefined types; they do not need to be declared.

Furthermore, it is possible to declare a new type according to the algorithm designer's needs. Generally speaking, the declaration of a new type follows the syntax below:

More details on creating custom types are discussed in Chapter 5.

Type type_name : **Base_type**;

3.5.3 Declaration of Variables

Recall that the notion of a variable is used to identify a location in memory. Its declaration follows the syntax below:

Variable variable_name : Type

Example 3.26 makes the declaration of variables clearer.

Exemple 3.16
Variable Counter : integer
X1 : real
letter : character
found : boolean

When several objects are of the same type, we can group them in a single declaration (Example 3.17).

Exemple 3.17
Variable Counter1, counter2 : integer
X1 , X2, X3 : real

3.5.4 Comments

Définition 3.11 A comment is free text that can span several lines and is enclosed by the character sequences "/*" and "*/".

Example 3.18 is a comment.

Exemple 3.18 /* This is a comment */

Comments are used to allow an easy interpretation of the algorithm. This interpretation is important both to allow readers of the algorithm to understand the proposed approach, and also for the algorithm designer, in case they no longer remember their reasoning.

Note that a comment is not taken into account during compilation. In other words, it has no influence on the execution of the algorithm.

3.6 Summary Example

In this summary example, we resume writing an algorithm for solving a quadratic equation, respecting the algorithmic formalism that we agreed upon. This example brings together most of the concepts (declaration of variables, read and write operations, alternative action, assignment, expressions, comments, ...) that we have discussed in this chapter.

```
Algorithm Equation_2_degree
  Variable a, b, c, delta, x1, x2 : real                    /* declaration part */

  begin
  |                                                         /* Start of algorithm body */
  |  Write ('enter the parameters a, b, and c of the equation ax^2 + bx + c = 0 : ')
  |  Read (a, b, c)                                        /* reading parameters */
  |  if a=0 then
  |  begin
  |  |  Write ('first-degree equation')
  |  end
  |  else
  |  begin
  |  |  delta ← b*b-4*a*c
  |  |  if delta > 0 then
  |  |  begin
  |  |  |  x1 ← (-b-sqrt(delta))/(2*a)
  |  |  |  x2 ← (-b+sqrt(delta))/(2*a)
  |  |  |  Write ('the roots of the equation are : ', x1, x2)
  |  |  end
  |  |  else
  |  |  begin
  |  |  |                                                  /* delta <= 0 */
  |  |  |  if delta = 0 then
  |  |  |  begin
  |  |  |  |  x1 ← -b/(2*a)
  |  |  |  |  Write ('the double root of the equation is : ', x1)
  |  |  |  end
  |  |  |  else
  |  |  |  begin
  |  |  |  |                                               /* delta < 0 */
  |  |  |  |  Write ('No real roots for the equation ')
  |  |  |  end
  |  |  end
  |  end
  |                                                       /* End of the algorithm */
  end
```

3.7 Pascal Language Corner

The language corner section is not a substitute for a complete course on programming languages, but just a summary of the concepts already introduced for the algorithmic language. This summary enables the student to easily translate the designed algorithms into a programming language.

3.7.1 Structure of a Pascal Program

Listing 3.1 shows the anatomy of a Pascal program.

```
1  PROGRAM ProgramName;
2
3  CONST
4  (* global declarations of Constants *)
5
6  TYPE
7  (* global declarations of Types *)
8
9  VAR
10 (* global declarations of Variables  *)
11
12 (* definitions of subprograms *)
13
14 BEGIN
15 (* body of the program *)
16 END.
```

Listing 3.1: Anatomy of a Pascal program

Note that the name of a Pascal source program ends with a *PAS* extension (example: hello.PAS). However, that of the executable ends with *EXE* (example: hello.EXE).

3.7.2 Turbo Pascal Reserved Words

By convention, reserved words in Pascal are written in uppercase. However, the Turbo Pascal compiler does not distinguish between uppercase and lowercase.

- AND - ARRAY - ASM
- BEGIN
- CASE - CONST - CONSTRUCTOR
- DESTRUCTOR - DIV - DO - DOWNTO
- ELSE - END - EXPORTS

- FILE - FOR - FUNCTION
- GOTO
- IF - IMPLEMENTATION - IN - INHERITED - INLINE - INTERFACE
- LABEL - LIBRARY
- MOD
- NIL - NOT
- OBJECT - OF - OR
- PACKED - PROCEDURE - PROGRAM
- RECORD - REPEAT
- SET - SHL - SHR - STRING
- THEN - TO - TYPE
- UNIT - UNTIL - USES
- VAR
- WHILE - WITH
- XOR

3.7.3 Identifiers

An identifier must have a maximum of 63 characters and must necessarily begin with a letter of the alphabet. The list of accepted characters is:

- a to z
- A to Z
- 0 to 9
- _

Identifiers must imperatively be different from reserved words and the names of procedures and functions defined by the Pascal language.

3.7.4 Comments

Comments are placed between braces *{* and *}*, or between parentheses and asterisks *(** and **)*. The two symbols must not be mixed.

3.7.5 Declarations

The declaration section is reserved for the declaration of types, constants, and variables. Each declaration ends with a semicolon.

Declaration of Types

In Turbo Pascal, there are predefined types. They do not need to be declared :

- Integer types: *Shortint*, *Integer*, *Longint*, *Byte*, and *Word*.
- Boolean types: *Boolean*, *Bytebool*, *WordBool*, and *LongBool*.
- Character type: *Char*.
- Real types: *Real*, *Single*, *Double*, *Extended*, *Comp*.

The declaration of custom types is covered in Chapter 5.

Declaration of Constants

We distinguish two kinds of constants: untyped constants and typed constants.

The declaration of *untyped constants* is done as follows:

```
Const identifier = constant;
```

Here are some examples of untyped constant declarations (Example 3.19):

Exemple 3.19

```
Const   min = 0;
max = 1000;
middle = (max - min) div 2;
message = 'insufficient memory';
numberofweeks = 365 div 7;
alphaletter = chr(224);

```

Typed constants define the type and the value. In this way, they are comparable to initialized variables. Their declaration is as follows:

```
Const identifier : Type = constant;
```

The following examples show some declarations of typed constants (Example 3.20):

Exemple 3.20

```
Const   min : Integer = 0;
max : Integer = 1000;
pi : Real = 3.1416;
message : String [18]= 'insufficient memory';
alphaletter : Char = chr(224);

```

Declaration of Variables

The syntax for declaring variables is as follows:

```
Var identifier1, identifier2, ... : Type;
```

Here are some examples of variable declarations (Example 3.21):

Exemple 3.21

```
Var X, Y, Z : Real;
i, j, k  : Integer;
End, correct : Boolean;

```

3.7.6 Statements

In this section, we examine the following statements: assignment, If, While, Repeat, and For.

Assignment

The sign `:=` is used to express an assignment statement.

Here are some examples (Example 3.22):

Exemple 3.22

```
Var X, Y, Z : Real;
i, j, k  : Integer;
End, correct : Boolean;
Begin
...
X := Y + Z;
i := j - k + digit;
End := not correct;
```

```
9     correct := false;
10    ...
11    End.
12
```

If Statement

The *if* statement takes two forms:

```
1  if condition then
2  block 1;
```

```
1  if condition then
2  block 1
3  else
4  block 2;
```

condition is a simple or compound boolean expression. *block 1* and *block 2* are blocks of statements.

If statement blocks consist of more than one statement, they must be enclosed by the reserved words **Begin** and **End**.

When using the *if* statement, the following rules must be observed:

1. It is not permitted to place a semicolon before the **else** clause.
2. The **else** clause always relates to the nearest **if** that is not already associated with another **else**.

While Statement

The *while* statement must follow this syntax:

```
1  while condition do
2  block;
```

condition is a simple or compound boolean expression. *block* is a set of statements.

The variables in the *condition* must be initialized before the *while* statement so that upon the first pass, the *condition* can be evaluated.

The following two Examples (3.23 and 3.24) illustrate the use of the *while* loop:

Exemple 3.23

```
1
2     while i <> j do
3     i := i + 1;
```

```
4
```

Exemple 3.24

```
1
2     while i < 100 do
3     Begin
4     write(i);
5     i := i + 1
6     End;
7
```

Repeat Statement

The syntax of the *repeat* statement is:

```
1   repeat
2   block
3   until condition;
```

condition is a simple or compound boolean expression. *block* is a set of statements.

It should be noted that *block* is not enclosed by the reserved words **Begin** and **End**. They are implicitly replaced by **repeat** and **until**.

For Statement

The syntax of the *for* statement is:

```
1   for v_control := e_initial to e_final do
2   block;
```

v_control is the control variable. It must be of a scalar type. *e_initial* and *e_final* are the initial and final expressions. They must be compatible with this scalar type.

e_initial and *e_final* are calculated only once, upon entering the *for* statement.

With the reserved word **to**, the control variable is incremented by 1 at each repetition. If *e_initial* is greater than *e_final*, the *for* statement is not executed.

Note that there is another form of the for statement:

```
1   for v_control := e_final downto e_initial    do
2   block;
```

With the reserved word **downto**, the control variable is decremented by 1 at each repetition. If *e_final* is less than *e_initial*, the *for* statement is not executed.

Read Statement

Keyboard input syntax is achieved in several forms:

```
read(P1, P2, ...);

readln(P1, P2, ...);

readln;
```

where $P1, P2, \ldots$ constitute a list of variables that can have different types: *integer*, *longint*, *real*, *char*, *string*, *array*[] of *char*, and possibly other types that we have not yet seen.

Write Statement

Screen display syntax is achieved in several forms:

```
write(E1, E2, ...);

writeln(E1, E2, ...);

writeln;
```

where $E1, E2, \ldots$ constitute a list of expressions that can have different types: *integer*, *longint*, *real*, *char*, *boolean*, string constant, *string*, *array*[] of *char*, and possibly other types that we have not yet seen.

Furthermore, it is possible to impose a display format for an expression using the notation *Ei:w* (w is an integer expression). This requests the display of the expression *Ei* over w characters.

Additionally, if *Ei* is of the real type, we can use the template *Ei:w:d* (w and d being integer expressions) to impose a *fixed-point* display over a total of w characters with d characters after the decimal point.

3.8 C Language Corner

3.8.1 Structure of a C Program

Listing 3.2 shows the structure of a C program.

```
// preprocessor directives starting with #

// declaration of external variables

// declaration of secondary functions

main()
```

```
8   {
9
10    // body of the program
11
12  }
13
```

Listing 3.2: Anatomy of a C program

Preprocessor directives allow for the introduction (before compilation) of already created functions. They also allow for the declaration of constant equivalences.

Secondary functions can be placed either before or after the main function.

The *main()* function constitutes the main program. This is the mandatory part of a C program.

3.8.2 ANSI C Reserved Words

ANSI (American National Standards Institute) C contains 32 reserved words (keywords):

- auto
- break
- const, continue, case, char
- double, default, do
- else, enum, extern
- float, for
- goto
- int, if
- long
- register, return
- short, struct, signed, switch, sizeof, static
- typedef
- unsigned, union
- void, volatile
- while

3.8.3 Identifiers

The role of an identifier is to give a name to a program entity (variable, function, type defined by *typedef*, structure, union, enum, label).

As in most programming languages, an identifier is a sequence of alphanumeric characters where the first character must be alphabetical. The underscore `_` is considered a letter. Thus, it can appear at the beginning of an identifier.

Unlike the Pascal language, uppercase and lowercase letters are distinct.

The compiler may truncate identifiers beyond a certain length. However, modern compilers can recognize at least 31 characters.

It is recommended to use identifiers entirely in uppercase for preprocessor variables.

3.8.4 Comments

In C, comments are placed between `/*` and `*/`. They can appear anywhere in the program to briefly explain what it does. They must not be nested.

3.8.5 Declarations

Declaration of Types

There is a limited number of predefined types in the C language (Table 3.9):

Table 3.9: Predefined types in the C language.

Type	Description
char	Character
int	Integer
float	Single-precision real
double	Double-precision real

The qualifiers *short* and *long* allow influencing the memory size of integers and reals.

The qualifiers *signed* and *unsigned* can be applied to character and integer types to indicate whether the most significant bit should be considered as a sign bit or not.

That being said, the declaration of custom types is covered in Chapter 5.

Declaration of Constants

A constant can be declared in the following form:

```
1  const Type Identifier = Value;
```

It can also be defined using the *#define* directive:

```
#define Identifier  Value
```

The *#define* directive instructs the preprocessor to substitute every occurrence of *Identifier* with *Value* in the remainder of the source file.

Example 3.25 illustrates the two ways to define a constant.

Exemple 3.25

```
#define pi 3.14
main()
{
  const float pii = 3.14;
  ...
}

```

Declaration of Variables

The declaration clause of a variable specifies its identifier and its type.

```
Type Identifier1, Identifier2, ..., Identifiern;
```

In C language, unlike Pascal, there is no single section dedicated to the declaration of variables. However, every variable must be declared before it is used. Furthermore, an initial value can be assigned to a variable during its declaration.

Example 3.26 presents a variety of variable declarations:

Exemple 3.26

```
#include <stdio.h>
unsigned char c;
long v;
main()
{
  unsigned int i = 1, j;
  short size;
  unsigned long tmp;
  double x = 2.33e5, y = 3.01, z;
  char car = 'B';
  char sep = '\'';  /* sep receives an apostrophe */
  char c1 = '\x041';/* c1 receives 41 as the
hexadecimal ASCII code for A */
  ...
```

```
14      }
15
```

3.8.6 Statements

In this section, we examine the following statements: assignment, *if*, *while*, *repeat*, and *for*.

Assignment

Assignment is performed using the `=` sign according to the following syntax:

```
1  variable = expression;
```

Here are some examples (Example 3.27):

Exemple 3.27

```
1      main()
2      {
3        int i, j, k;
4        i = 4;        /* i receives 4 */
5        j = (i*3)+1; /* j receives 13, the result of (4*3)
   +1 */
6        k = j = 9;    /* j first receives 9, then k receives
    the value of j (9) */
7      }
8
```

Assignment in the C language can perform an implicit type conversion; the value of the expression on the right-hand side is converted to the type of the term on the left-hand side, as illustrated by Example 3.28.

Exemple 3.28

```
1      main()
2      {
3        int i;           /* i integer */
4        float x = 2.5; /* x real initialized to 2.5 */
5        i = x;           /* i receives 2, the result is
   truncated */
6        char c = 'B';    /* c character initialized to 'B'
```

```
   */
7      i = c;          /* i receives 66, the ASCII code for
    'B' */
8      }
9
```

Conditional Statements

There are two forms of conditional statements: *if-else* and *else-if*.

The *if-else* form is used to express decisions according to the following syntax:

```
1  if (condition)
2  block 1
3  else
4  block 2
```

The *else* part is optional, and each block can contain one or more statements.

Exemple 3.29

```
1      if (n < 0)
2      if (a == b)
3      a = - a;
4      else
5      b = - b;
6
```

In Example 3.29, the first *if* is without an *else*, and the *else* refers to the second *if*. The general rule states that *else* always refers to the innermost *if*.

The *else-if* form is used to express multi-way decisions according to the following syntax:

```
1  if (condition 1)
2  block 1
3  else if (condition 2)
4  block 2
5  else if (condition 3)
6  block 3
7  ...
8
9  else
10 block n
```

The conditions are evaluated in order. If any condition is true, the associated block is executed. If no condition is satisfied, the last block associated with *else* is executed. Note that the final *else* is optional.

The binary search example illustrates the use of the *else-if* form (Example 3.30).

Exemple 3.30

```
/* search for x in a sorted array v of size n */
int binsearch(int x, int v[], int n)
{
  int low, high, mid;
  low = 0;
  while (low <= high) {
    mid = (low + high) / 2;
    if (x < v[mid])
    high = mid - 1;
    else if (x > v[mid])
    low = mid + 1;
    else
    return mid; /* x exists in v */
  }
  return -1; /* x does not exist in v */
}

```

Repetitive Statements

In this section, we present the repetitive statements *while*, *for*, and *do-while*, as well as the loop exit statements *break* and *continue*.

- **While Statement**

  syntax

```
while (condition)
block

```

Exemple 3.31

```
    int n = 10, s = 0;
    while (n > 0)
    {
```

```
        s = s + n;
        n--;
      }

```

- **For Statement**

  syntax

```
for (initializations ; condition ; finalizations)
block

```

This structure begins with the *initializations* statements, which will be executed only once, at the beginning of the loop's execution.

The *condition* is evaluated and tested before entering the loop *block*.

The *finalizations* statements are executed at the end of each iteration.

The *for* structure is equivalent to:

```
initializations
while (condition)
{
  block
  finalizations
}

```

Exemple 3.32

```
    int n, i, fact;
    ...
    for (i=1, fact=1 ; i <= n ; i++)
    fact *= i;
    ...

```

- **Do-while Statement**

  syntax

```
do
block
while (condition);
```

```
4
```

Exemple 3.33

```
int n = 10, s = 0;
do
{
  s = s + n;
  n--;
}
while (n = 0);
```

Note that the *do-while* form in C is comparable to *repeat-until* in Pascal.

- **Break and Continue Statements**

  The *break* statement provides an early exit from *while*, *do-while*, and *for* loops.

  Meanwhile, the *continue* statement allows skipping directly to the next iteration of the current loop. In *while* and *do-while* loops, control passes immediately after *continue* to the *condition*, which will then be evaluated and tested. In a *for* loop, control passes to the *finalizations* statements.

Exemple 3.34

```
int i;
for (i = 0 ; i < 10 ; i++)
{
  ...
  if (a[i] == 0) /*ignore zero elements*/
  continue;
  printf("again");
  ...
  if (a[i] < 0) /*exit if negative element*/
  break;
  ...
}
```

Write Statement

The C language uses the *printf* function to display information on the screen:

```
int printf(format, E1, E2, ...);
```

format is a string that can contain:

- text to be displayed,
- conversion specifiers.

Conversion specifiers indicate the display format of the expressions *Ei*. These specifiers always begin with `%` and end with one or two conversion characters. The *printf* function returns the number of characters displayed.

Between the `%` sign and the conversion character, there can be, in order:

- The minus sign `-` which specifies a left alignment of the corresponding expression.
- A number that specifies the minimum field width.
- A dot `.` which separates the field width from the precision.
- A number, a precision that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point in a floating-point value, or the minimum number of digits for an integer.
- an *h* if the integer is to be printed in *short* format, or *l* for *long* format.

Table 3.10 presents the basic conversion specifiers.

Table 3.10: Basic conversion specifiers for *printf*

Format	**Expression type**	**Display type**
%d, %i	int	signed decimal
%o	unsigned int	unsigned octal
%x, %X	unsigned int	unsigned hexadecimal
%u	unsigned int	unsigned decimal
%c	unsigned char	a character
%s	char *	string
%f	double	fixed-point decimal
%e, %E	double	exponential notation decimal
%g, %G	double	decimal, the shortest representation between %f and %e

Example 3.35 shows various uses of the *printf* function.

Exemple 3.35

```c
    int i = 123, j = 45;
    printf("%i x %i = %li\n", i, j, (long)i*j);
    /* displays 123 x 45 = 5535 */
    char c = 'B';

    printf("The character %c has the code %i\n", c, c);
    /* displays The character B has the code 66 */

    long N = 15000000;
    printf("%d, %d", N);
    /* displays -7328, 35 */
    /* The first value is truncated, the second is
   relatively random */

    printf("%ld, %lx", N, N)
    /* displays 1500000, 16e360 */

    float X = 12.1234;
    double Y = 12.123456789;
    long double Z = 15.5;
    printf("%f", X); /* displays 12.123400 */
    printf("%f", Y); /* displays 12.123456 */
    printf("%e", X); /* displays 1.212340e+01 */
    printf("%e", Y); /* displays 1.212346e+01 */
    printf("%le", Z); /* displays 1.550000e+01 */

    printf("%f", 123.456); /* displays 123.456000 */
    printf("%12f", 123.456); /* displays __123.456000 */
    printf("%.2f", 123.456); /* displays 123.45 */
    printf("%5.0f", 123.456); /* displays __123 */
    printf("%10.3f", 123.456); /* displays ___123.456 */

    char *string = "algorithms and DS";
    printf("%s \t %10s", string, string);
    /* displays algorithms and DS     algorithm */

```

In addition, the *sprintf* function performs the same conversions as *printf*, but it saves the results in a string variable:

```
sprintf(output, format, E1, E2, ...);
```

with *output* being a string variable that records the result.

Read Statement

The *scanf* function is the input counterpart to *printf*, providing conversion options in the reverse direction:

```
scanf(format, P1, P2, ...);
```

The *scanf* function reads characters from the standard input, interprets them according to the conversion specifiers in *format*, and saves the converted input data into the parameters *Pi*, whose names must be preceded by the `&` sign.

scanf stops when it exhausts the *format* string or when an input does not match the conversion specifier. It returns as a result the number of parameters successfully received and assigned.

Table 3.11 summarizes the basic conversion specifiers.

Table 3.11: Basic conversion specifiers for *scanf*

Format	Parameter type	Data type
%d	int*	signed decimal
%o	int*	octal
%u	unsigned int*	unsigned decimal
%x	int*	hexadecimal
%c	char*	a character
%s	char*	string
%f	float*	fixed-point float
%e	float*	exponential float
%g	float*	fixed-point or exponential float

Example 3.36 illustrates some uses of the *scanf* function.

Exemple 3.36

```
    int i;
    float x;
    char c;

    scanf("%d", &i);
    /* reads an integer value and assigns it to i  */

    scanf("%d%f", &i, &x);
    /* reads an integer value for i and a real value for
   x */

    scanf("%c", &c)
    /* reads a character and assigns it to the variable c
    */

```

Finally, note that the expression *c = getchar()* plays the same role as calling the *scanf("%c", &c)* function, but it is faster.

3.8.7 Operators

This section focuses on the different operators in the C language, namely: arithmetic, relational, boolean logical, bitwise logical, compound assignment, increment, decrement, sequential, conditional, and cast operators.

Arithmetic Operators

The arithmetic operators are presented in Table 3.12. Integer division truncates any fractional part. The `%` operator cannot be applied to *float* or *double* data types.

Table 3.12: Arithmetic Operators

Operator	Operation
+	addition
−	subtraction
*	multiplication
/	division
%	modulo (remainder of integer division)

Relational Operators

The relational operators are presented in Table 3.13.

Table 3.13: Relational Operators

Operator	Operation
>	strictly greater than
>=	greater than or equal to
<	strictly less than
<=	less than or equal to
==	equal to
! =	not equal to

Boolean Logical Operators

Table 3.14 summarizes the boolean logical operators. Expressions connected by the `&&` and `||` operators are evaluated from left to right, and the evaluation stops as soon as the truth or falsehood of the result is known.

Table 3.14: Boolean Logical Operators

Operator	Operation
&&	logical AND
||	logical OR
!	logical NOT

Bitwise Logical Operators

The C language provides six operators (Table 3.15) for bit manipulation. They apply only to operands of type *char*, *short*, *int*, and *long*, whether they are *signed* or *unsigned*.

Compound Assignment Operators

Most binary operators have a corresponding compound assignment operator *op*=, where *op* is the binary operator. Table 3.16 presents the compound assignment operators and their meanings.

Table 3.15: Bitwise Logical Operators

Operator	Operation
&	bitwise AND
|	bitwise inclusive OR
∧	bitwise exclusive OR (XOR)
<<	left shift
>>	right shift
~	one's complement (NOT)

Table 3.16: Compound Assignment Operators

Operator	Usage	Equivalence
+=	$x+=y$	$x=x+y$
−=	$x-=y$	$x=x-y$
*=	$x*=y$	$x=x*y$
/=	$x/=y$	$x=x/y$
%=	$x\%=y$	$x=x\%y$
&=	$x\&=y$	$x=x\&y$
∧=	$x\wedge=y$	$x=x\wedge y$
|=	$x\|=y$	$x=x\|y$
≪=	$x\ll=y$	$x=x\ll y$
≫=	$x\gg=y$	$x=x\gg y$

Increment and Decrement Operators

The increment and decrement operators (Table 3.17) are used in either postfix notation ($x++$, $x--$) or prefix notation ($++x$, $--x$). In postfix notation, the old value of x is returned, whereas in prefix notation, the new value is returned, as illustrated in Example 3.37.

Table 3.17: Increment and Decrement Operators

Operator	Usage	Equivalence
++	x++	x = x + 1
−−	x−−	x = x - 1

Exemple 3.37

```
    int x, y, z = 2;
    x = ++z; /* x is 3 */
    z = 2;
    y = z++; /* y is 2 */

```

Sequential Operator (comma)

The sequential operator `,` (comma) allows for grouping successive expressions within a single expression. These expressions are evaluated in sequence from left to right.

Exemple 3.38

```
    x = ((y=3), (y+2)); /* x is 5 */

```

Conditional Operator

The ternary conditional operator `?:` provides an alternative way to express a conditional statement.

syntax

```
condition ? expression_1 : expression_2;
```

If the *condition* is satisfied, *expression_1* is evaluated and its result becomes the value of the conditional expression. Otherwise, *expression_2* is evaluated and that becomes the value of the conditional expression.

Exemple 3.39

```
    z = (x > y) ? x : y; /* z = max(x, y) */

```

Cast Operator

It is possible to force the type of an expression by using the conversion operator, called a *cast*.

syntax

```
(type) expression;
```

Exemple 3.40

```
1     int n = 10, p = 3, x;
2     float y;
3     x = n/p; /* x is 3 */
4     y = (float) n/p /* y is 3.33333... */
5
```

Operator Precedence Rules

Table 3.18 summarizes the rules of precedence and associativity for all C language operators, including those not presented in this section (Some of the operators not presented here will be studied later in the chapters corresponding to arrays and custom types. The rest will be studied in the second volume *Algorithms and Data Structures - Part 2*).

Operators on the same line have the same precedence. The lines are listed in descending order of precedence.

Table 3.18: Operator Precedence and Associativity

Category	**Operator**	**Associativity**
reference	() [] −>	left to right
unary	+ − ++ −− ! ∼ ∗ & (cast) sizeof	right to left
arithmetic	∗ / %	left to right
arithmetic	+ −	left to right
shift	≪ ≫	left to right
relational	< <= > >=	left to right
relational	== !=	left to right
bitwise	&	left to right
bitwise	∧	left to right
bitwise	|	left to right
logical	&&	left to right
logical	||	left to right
conditional	?:	right to left
assignment	= += −= ∗= /= &= ∧= |= ≪= ≫=	right to left
sequential	,	left to right

3.9 Exercises

The pedagogical objectives of this series of exercises are:

- Problem analysis (Building an analysis)
- Algorithm design
  - Moving from an analysis to an algorithm.
  - Object manipulation (assignment + expressions).
  - Mastery of control structures.
  - Tracing an algorithm.

We have voluntarily included a significant number of exercises in this series, as we believe it is crucial for an algorithm designer to master algorithmic formalism perfectly so that they can, later on, propose efficient, concise, and elegant algorithms.

Exercise 3.2 For which values of variables a and b does this expression return true?
$((a * b >= 15)\ and\ (\ not\ (a < 4)\ or\ (b >= 4\ and\ b <= 7)))$

1. a = 5, b = 2
2. a = 6, b = 2
3. a = 5, b = 10
4. a = 9, b = 2

Indications It is important to know how to evaluate mixed expressions. They help you trace your algorithms correctly and learn to formulate conditions. Simply put, you must respect operator precedence rules and the use of parentheses.

Exercise 3.3 How would you write an **If** condition that returns true if the variable i is in the interval $[1, 10[\cup]20, 30]$?

1. If ((i >= 1 and i < 10) and (i > 20 and i <= 30))
2. If ((i >= 10 and i < 20) or (i > 20 and i <= 30))
3. If ((i >= 1 and i < 10) or (i > 20 and i <= 30))
4. If ((i >= 1 or i < 10) or (i > 20 and i <= 30))

Indications Try to design the condition on your own first, then compare it with the list of choices.

Exercise 3.4 Given the following code, what does it display?

```
Algorithm test
Variable test1, test2 : boolean
begin
    test1 ← false
    test2 ← (true or test1)
    Write ('Sol ')
    if (test2) then
    begin
        Write('Do ')
    end
    else
    begin
        if (test1 and test2) then
            Write('Mi ')
        else
            Write('Fa ')
    end
    if ((test1 and test2) or (test1 or Not(test2))) then
        Write('La ')
    else
        Write('Do ')
end
```

Indications You are asked to trace (manual execution) the algorithm. Work on the trace carefully, as tracing an algorithm is a very pedagogical way to master computer science logic.

Exercise 3.5 Given the following code, what does it display?

```
Algorithm IFELSE
  Variable x, y, z : integer

  begin
    x ← 2
    y ← 5
    z ← 10
    if (x * y = z) then
    begin
      Write('1 ')
      x ← x+3
      z ← z-2
    end
    else
    begin
      Write('2 ')
    end
    if (z MOD 2 = 0) then
    begin
      if (x + z = 10) then
      begin
        Write('3 ')
      end
      else
      begin
        Write('4 ')
      end
    end
    else
    begin
      Write('5 ')
    end
  end
```

Indications You are asked to trace the algorithm. Work on the trace carefully, as tracing an algorithm is an excellent pedagogical tool.

Exercise 3.6 What will be obtained in the variables $a, b, n1$ and $n2$ after executing the action blocks in Table 3.19?

Indications After solving the exercise, try to think about how to keep only the indispensable actions.

Table 3.19: Action blocks.

Block A	**Block B**	**Block C**
$a \leftarrow 5$	$n1 \leftarrow 5$	$n1 \leftarrow 5$
$b \leftarrow a + 4$	$n2 \leftarrow 7$	$n2 \leftarrow 7$
$a \leftarrow a + 1$	$n1 \leftarrow n2$	$n2 \leftarrow n1$
$b \leftarrow a - 4$	$n2 \leftarrow n1$	

Exercise 3.7 Find the boolean values taken during the following algorithm:

Algorithm Eval_expression

```
Variable a, b : integer
         b1, b2, b3, b4 : boolean

begin
    a ← 10
    b ← 4
    b1 ← (10 > 10)  AND  (5 = 5)
    b2 ← (a = 10)  OR  (b = 5) OR (3 = 6)
    b3 ← (a > b)  AND  ((5 =5 ) OR (b < a))
    b4 ← (False)  AND  (True)  OR  (a > b)
end
```

Indications You can use a truth table to evaluate boolean expressions.

Exercise 3.8 Suppose that the variables n, p, and q are of integer type and contain the values 8, 13, and 29 respectively. Determine the values of the following expressions:
$n + p/q$, $n + q/p$, $(n + q)/p$, $n + p/n + p$

Indications Pay attention to operator precedence.

Exercise 3.9 Write an algorithm that swaps the values of 2 real variables:

1. Using an intermediate variable.
2. Without using an intermediate variable.

Indications The task of swapping the values of two variables is simple, but it is widely used in other algorithms, particularly sorting algorithms.

Exercise 3.10 Given the following declarations and actions:

```
Variable n, p : integer
         x : real

begin
    n ← 10
    p ← 7
    x ← 2.5
end
```

Provide the type and value of the following expressions:

1. x + n / p
2. (x + n) / p
3. 5. * n
4. (n + 1) / n
5. (n + 1.0) / n

Exercise 3.11 Write an algorithm that assigns values to two integer variables named a and b, and displays their values, their product, and their sum in this form:
$a = 3, b = 5$
$a * b = 15$
$a + b = 8$

Indications Be careful, the values 3 and 5 for variables a and b are just examples. Your algorithm must handle the general case. Furthermore, you must respect the display format.

Exercise 3.12 Write an algorithm that asks for two integers and provides their sum and product. The dialogue with the user should look like this:

```
Enter two numbers
35      3
their sum is 38
their product is 105
```

Indications Be careful, the values 35 and 3 are just examples. Your algorithm must handle the general case. Respect the display format.

Exercise 3.13 Write an algorithm that reads the pre-tax price of an item, the number of items, and the VAT rate, and displays the corresponding total price including tax. The dialogue will appear as follows:

unit price (excl. tax):
12.5
number of items:
8
VAT rate:
19.6
total price (excl. tax): 100.
total price (incl. tax): 119.6

Indications Be careful, the values 12.5, 8, and 19.6 are just examples. Your algorithm must handle the general case.

Exercise 3.14 An insurance company offers three rates (Green, Orange, and Red) based on the age and number of accidents of drivers (Table 3.20).
Write an algorithm that displays the rate after entering a driver's age and number of accidents.

Table 3.20: Insurance rates.

Number of accidents	Under 25 years old	25 years and older
0 accidents	Orange	Green
1 or 2 accidents	Red	Orange
3 to 6 accidents	Not insured	Red
7 accidents or more	Not insured	Not insured

Indications You can provide several variants of this algorithm depending on whether you consider age or the number of accidents first, and whether you use simple or complex conditions.

Exercise 3.15 You want to compare two phone subscription offers (Table 3.21). The bill is calculated with a fixed monthly fee and a part proportional to the time spent calling (indicated in minutes).

Exercise 3.16 Complete Table 3.22 representing the correspondence between continuity conditions and stop conditions.

Table 3.21: Phone subscription offers.

Offer	Fixed Fee	Price per minute
Telecom 1	200 DA	2 DA
Telecom 2	300 DA	1.50 DA

Table 3.22: Correspondence between continuity conditions and stop conditions.

Stop condition	Continuity condition
(nb = 4) AND (age < 25)	
(dice = 6) OR (nb_tries > 5)	
(dice1 = 6 AND (dice2 = 6) OR (nb_tries > 5)	
(dice1 = 6) OR (dice2 = 6)	

Indications This is an interesting exercise that teaches us how to transition from a continuity condition to a stop condition. In other words, how to move from a *while* loop to a *repeat* loop and vice-versa.

Exercise 3.17 Write an algorithm that reads two integers and determines if they are sorted in ascending order and, in all cases, displays their difference (between the larger and the smaller).

Exercise 3.18 Write an algorithm that reads three integers and finds if they are sorted in strict ascending order, meaning if a, b, and c designate these numbers, they must verify the mathematical inequality: $a < b < c$.

Exercise 3.19 Write an algorithm that inputs an integer and indicates if it is even or odd.

Indications You can use the modulo operator.

Exercise 3.20 Write an algorithm that reads a price excluding tax and calculates and displays the corresponding price including tax (with a VAT rate of 17%). It then establishes a discount based on the following rates:

- 0% for a total including tax less than 1000 DA,
- 1% for a total including tax greater than or equal to 1000 DA and less than 2000 DA,
- 2% for a total including tax greater than or equal to 2000 DA and less

than 5000 DA,

- 5% for a total including tax greater than or equal to 5000 DA.
- The algorithm will display the discount obtained and the new total including tax.

Exercise 3.21 Write an algorithm that reads an integer representing a month of the year (1 for January, 4 for April, ...) and displays the number of days in that month (assume it is not a leap year). Account for cases where the user provides an incorrect number, i.e., not between 1 and 12.

Indications A leap year is a year with 366 days instead of 365 days, meaning it includes February 29th.

Exercise 3.22 Given the following code, what does it display?

```
Algorithm while_loop
    Variable k, m : integer

    begin
        k ← 11010
        m ← k MOD 2
        while ( k > 0) do
        begin
            m ← (m + k MOD 10) * 2
            k ← k DIV 10
        end
        Write (m)
    end
```

Indications You are asked to trace (manual execution) the algorithm.

Exercise 3.23 The following code is executed and displays the following output: 15 16 17 18 19 2

```
Algorithm while_loop
   Variable m, p, n : integer
            test1, test2 : boolean

   begin
      p ← 4
      n ← 10
      test1 ← false
      test2 ← (test1 and (p < n))
      for m from (n DIV 2) to n-1 do
      begin
         if (test1 or test2) then
         begin
            Write ((p + m), ' ')
            test2 ← false;
         end
         else
         begin
            Write ( (n + m), ' ')
            (* missing code *)
         end
         if (test1 and test2) then
         begin
            Write( 1, ' ')
         end
         else
         begin
            Write( 2, ' ')
         end
      end
   end
```

What is the missing code?

1. test1 ← true
2. test1 ← not test1
3. test1 ← not test2
4. test1 ← test2

Indications You are asked to trace the algorithm.

Exercise 3.24 Write an algorithm that asks the user for a positive integer less than 100 repeatedly until the answer is satisfactory. The dialogue with the

user should look like this:

give a positive integer less than 100:
453
give a positive integer less than 100:
25
thank you for the number 25

Indications This is an interesting exercise for data validity checking. Use the *repeat* loop.

Exercise 3.25 In Exercise 3.24, the user is asked the same question regardless of whether it is the first request or a retry after an incorrect answer. Improve it so the dialogue looks like this:

give a positive integer less than 100:
453
Please, less than 100:
0
Please, positive:
25
thank you for the number 25

Indications This is an interesting exercise for data validity checking. Use the *repeat* loop.

Exercise 3.26 Rewrite the algorithm requested in Exercise 3.24 using a *while* repetition.

Indications Use the rules for transitioning from stop conditions to continuity conditions learned in Exercise 3.16.

Exercise 3.27 Rewrite the algorithm requested in Exercise 3.25 using a *while* repetition.

Indications Use the rules for transitioning from stop conditions to continuity conditions learned in Exercise 3.16.

Exercise 3.28 Write an algorithm that displays the squares of integers from 7 to 20.

Exercise 3.29 Write an algorithm that reads two integers into variables nd and nf and writes the doubles of the numbers between these two limits (inclusive).

Exercise 3.30 Construct an algorithm to perform the multiplication of 2 integers using successive additions.

Exercise 3.31 Construct an algorithm that calculates the N-th (with $N > 2$) term of the FIBONACCI sequence defined by:

$$\begin{cases} U_0 &= U_1 = 1 \\ U_n &= U_{n-1} + U_{n-2} \end{cases}$$

FIBONACCI sequence: 1, 1, 2, 3, 5, 8, 13, ...

Exercise 3.32 Knowing that a prime number is a number that has no divisors except 1 and itself, construct an algorithm that gives the first N prime numbers.

Exercise 3.33 Knowing that there are only 4 numbers between 100 and 500 such that the sum of the cubes of their digits equals the number itself, construct an algorithm to find these 4 numbers. Example: $153 = 1^3 + 5^3 + 3^3$

Exercise 3.34 A perfect number is a number that is equal to the sum of all its divisors except itself. Construct an algorithm to find all perfect numbers between 1 and 1000.

Indications A perfect number is a natural integer n such that $\sigma(n) = 2n$, where $\sigma(n)$ is the sum of the positive divisors of n. 6 is a perfect number because $2 \times 6 = 12 = 1 + 2 + 3 + 6$, or simply $6 = 1 + 2 + 3$.

Exercise 3.35 Construct an algorithm to find the LCM (least common multiple) of 2 numbers A and B.

Indications There are multiple solutions depending on the LCM calculation methods you have studied in your curriculum.

Exercise 3.36 Given 2 numbers A and B such that $A_x = B_{10}$ (where x and 10 are the bases in which A and B are written), construct an algorithm that determines the base in which A is written.
Example: $A = 20405_x$ and $B = 8453_{10}$
The base in which number A is written is base 8.

Exercise 3.37 Mathematics is a fascinating science, yet it remains a daunting subject for many. However, one of its domains cannot leave any curious mind indifferent: mathematical curiosities. Among these is "the band of 9."

Indeed, if you take three (3) numbers (A, B, C), each composed of three (3) digits, such that: A + B = C, and if the nine (9) digits used are: 1, 2, 3, 4, 5, 6, 7, 8, 9, then the sum of the digits constituting the result (C) is always equal to 18.

Examples:
152 + 487 = 639 238 + 419 = 657 357 + 462 = 819 784 + 152 = 936

Construct a solution that will allow you to find all cases (i.e., A, B, and C) that respect this "oddity," as well as their total count.

Exercise 3.38 Construct an algorithm that allows us to obtain a number N having the following successive values:
1
22
333
4444
55555
666666
7777777
88888888
999999999
As well as the sum of the digits composing the set of these numbers.

Exercise 3.39 In the theory of perfect numbers, EULER demonstrated that the expression: $2^{n-1}(2^n - 1)$ always yields a perfect number when the quantity in parentheses is a prime number. Construct an algorithm that allows us to obtain the first 5 perfect numbers.

Example: when n = 2, $2^n - 1$ is equal to 3, which is prime, and $2^{n-1}(2^n - 1)$ is equal to 6, and 6 is a perfect number.

4. Arrays and Strings

4.1 Introductory Example

We approach this chapter by discussing a small problem.

Suppose we want to process and preserve the grades of a class of 30 students. For example, this task might involve:

- Ranking the students in the class.
- Calculating the number of students with a grade higher than 10.

According to the algorithmic formalism we have learned so far, we currently only have simple variable types at our disposal. To solve the problem mentioned above, we need to have the grades of all 30 students available simultaneously. To do this, we could use 30 different variables named, for example: *grade1*, *grade2*, . . . , *grade30*.

For example, to calculate the number of students with a grade higher than 10 (while preserving the grades), we would have to write:

1. 30 read actions for the grades:

Read (grade1)
Read (grade2)
. . .
Read (grade30)

---

2. 30 conditional actions to perform the calculation:

```
Nbr_stud ← 0
if grade1 > 10 then
   Nbr_stud ← Nbr_stud + 1;
if grade2 > 10 then
   Nbr_stud ← Nbr_stud + 1;
...
if grade30 > 10 then
   Nbr_stud ← Nbr_stud + 1;
```

The problem with this solution is the tediousness of writing the algorithm. This writing becomes even more fastidious if the number of students jumps to 100 or 1000.

An alternative to this problem is to consider a single composite data structure that can hold multiple values. This structure is called an *array*. This is indeed the subject of the first part of this chapter.

Furthermore, most programming languages offer the array data structure as a built-in service.

4.2 One-Dimensional Arrays

We begin by defining the notion of *one-dimensional arrays* as well as the associated concepts (Definitions 4.1, 4.2, and 4.3).

Définition 4.1 A *one-dimensional array*, also called a *vector*, is a data structure designated by a **unique identifier** allowing the storage of a finite number of elements of the same type, which are directly accessible by their *indices*.

Définition 4.2 An *index* is a scalar type variable that allows access to the elements of an array. An index indicates the rank (position) of the element.

Définition 4.3 The *size* of an array, sometimes called its dimension, is the maximum number of its elements.

The declaration of an array follows the following syntax:

```
Variable Array_Name : array [MinDim..MaxDim] of component_type
```

Exemple 4.1
Variable grades : **array** [1..30] of **real**

In Example 4.1, *grades* is the name of the array; it is a unique identifier for the entire composite structure containing several elements. ***array*** is the reserved word

indicating that the variable *grades* is of type ***array***. The indices 1 and 30 are respectively the lower and upper bounds of the *grades* array. These indices define the size of the array, in this case 30 elements of type *real*, which represents the array's component type. In other words, the type of the elements within an array is the type of the array itself.

Figure 4.1 illustrates the concept of the array declared in Example 4.1.

Figure 4.1: Example of an array structure

4.3 Using a One-Dimensional Array

The advantage of arrays lies in the fact that an array is a single structure gathering multiple elements, and each element is individually accessible.

Accessing an array element relies on the concept of an *indexed variable*, which is implemented through an indexing operation. This consists of specifying an index value in square brackets after the array name. Consequently, the access time to an element is constant, regardless of which element is requested.

When an array is declared, a contiguous memory space is allocated to it. this space must be defined before use and cannot be changed.

The indexed variable (array element) is used just like a simple variable. Therefore, it can:

1. Be the subject of an assignment, as shown in Example 4.2.

Exemple 4.2

```
grades[1] ← 15.10
grades[27] ← 13.00
```

2. Appear in the list of a read action (Example 4.3).

Exemple 4.3

```
For i from 1 to 30 do
begin
    Write ('enter the grade at rank ', i)
    Read (grades[i])
end
```

3. Appear in the list of a write action (Example 4.4).

Exemple 4.4

```
For i from 1 to 30 do
begin
    Write ('The grade at rank ', i,' is equal to: ', grades[i])
End
```

4. Appear in an expression (Example 4.5).

Exemple 4.5

```
Average ← 0
For i from 1 to 30 do
begin
    Average ← Average + grades[i]
End
Average ← Average / 30
```

To clearly understand the importance and necessity of using arrays, let's return to the opening problem of this chapter, which involves calculating the number of students with a grade higher than 10. This time, we write Algorithm *Grade_Sup_10* using an *array* data structure.

Algorithm Grade_Sup_10

```
Constant max = 30
Variable i, Nbr_stud : integer
         grades : array[1..max] of real

begin
    Nbr_stud ← 0
    For i from 1 to max do
    Begin
        Write ('enter the grade at rank ', i)
        Read (grades[i])
        if grades[i] >= 10 then
            Nbr_stud ← Nbr_stud + 1
    End
    Write ('The number of students with a grade higher than 10 = ',
     Nbr_stud)
end
```

4.4 Algorithms on Arrays

The objective of this section is to present two main methods that utilize arrays, namely, a sorting method and a method for searching for a value within an array.

In our context, we are more interested in the practical use of arrays than in the various specific technical nuances of sorting and searching techniques.

4.4.1 Sorting an Array

Sorting consists of reorganizing a sequence of objects to put them into a logical order. By default, sorting implies an ascending order. Otherwise, a descending order must be explicitly specified.

There are several reasons to study sorting algorithms:

- Techniques similar to those used in elegant sorting methods are effective for solving other problems.
- We often use sorting algorithms as a starting point for solving other problems.

In our case, the most important reason is to master the use of the array structure.

In the literature, several sorting methods exist:

- Bubble sort,
- selection sort,
- insertion sort,
- counting sort,
- shell sort,
- quick sort,
- merge sort.

We reiterate that our goal is not a complete study of sorting methods, but rather the utilization of arrays. For this reason, we limit ourselves to a single algorithm: bubble sort.

Bubble Sort

Although the bubble sort method is recognized as an inefficient method, we present it here for pedagogical reasons due to its simplicity.

Bubble sort takes place within the same array according to the following principle:

1. Traverse the array by comparing consecutive elements. If they are poorly ordered, swap them.
2. Repeat until there are no more swaps to perform.

The realization of the aforementioned principle using algorithmic formalism results in Algorithm *Bubble_Sort*.

Algorithm Bubble_Sort

```
Constant n = 5
Variable grades = array[1..n] of real
         swap : boolean /* indicates if a swap was performed */
         i : integer
         Temp : real

begin
   /* input grades */
   For i from 1 to n do
   begin
        Write ('enter the grade at rank ', i)
        Read (grades[i])
   end
   /* Sort the array */
   repeat
        swap ← false
        For i from 1 to n-1 do
        begin
             if grades[i] > grades[i+1] then
             begin
                  swap ← true
                  Temp ← grades[i]
                  grades[i] ← grades[i+1]
                  grades[i+1] ← Temp
             end
        end
   until not swap
end
```

Tracing

To fully understand how an algorithm works and to globally verify its validity, it is natural to proceed with its tracing (manual execution).

Tracing first consists of drawing a table where each column is reserved for a variable. Then, one starts executing the algorithm action by action, noting each time the change in variable values in the associated columns.

As a warm-up, let's trace Algorithm *Bubble_Sort*.

The first loop is used to input data into the *grades* array. Suppose that at the end of this first loop, the content of the *grades* array is as shown in Figure 4.2.

Figure 4.2: The content of the *grades* array to be sorted.

The *repeat* loop handles the sorting operation of the *grades* array. One pass of this loop consists of going from one end of the array to the other, comparing neighboring elements and swapping them if they are poorly ordered. The swap is performed using an intermediate variable (*temp*). This swap moves up through the array like a bubble.

At the end of the first pass (Figure 4.3), the *grades* array is partially sorted. The values are not yet in their final positions, so there are still pairs of poorly ordered elements. The content of the boolean variable *swap* (*swap* = True) indicates that at least one swap (permutation) was performed between two neighboring elements during the current pass. Consequently, one or more additional passes will be needed.

Figure 4.3: First pass of bubble sort.

The tracing of the algorithm continues with the second and third passes (Figure 4.4 and Figure 4.5 respectively), but still with the presence of one or more swaps. This implies that a new pass is still necessary.

Figure 4.4: Second pass of bubble sort.

Figure 4.5: Third pass of bubble sort.

In the fourth pass (Figure 4.6), it is observed that the variable *swap* remains unchanged (*swap* = False), revealing that no swap was performed. Thus, the *grades* array is sorted, and the algorithm must stop.

4.4.2 Searching for an Element in an Array

Searching for an element is one of the most frequent tasks performed on arrays. It consists of determining whether a specific element is present in the structure or not. If the element is found, the algorithm must return its position.

Two search techniques are widely studied: sequential search and binary search.

Sequential search involves scanning the array from beginning to end until the desired value is found or the end of the array is reached. This can hinder performance, especially when the size of the array is substantial.

In the following section, we study **binary search**, also known as **dichotomous search**. This method is only applicable if the array is already sorted.

Binary Search

Binary search relies on the "divide and conquer" principle. Let *tab* be a sorted array containing n elements. We wish to search for a value X. If X exists, the algorithm displays its index; otherwise, it signals its absence.

The principle can be described in the following steps:

Figure 4.6: Final pass of bubble sort.

1. **Initialization:** The initial search range is the entire array, denoted as *tab[inf..sup]* with $inf = 1$ and $sup = n$.

2. **Iteration:** At each step, we define:

   - The midpoint of the current range: $m = (inf + sup)$ div 2.
   - A subdivision into three zones: $[inf..m-1]$, the element at $[m]$, and $[m+1..sup]$.

3. **Comparison:** Three cases are possible:

   - $tab[m] = X$: The element is found; the algorithm terminates.
   - $tab[m] < X$: X cannot be in the range $[inf..m]$. The search continues in the upper range $[m+1..sup]$.
   - $tab[m] > X$: X cannot be in the range $[m..sup]$. The search continues in the lower range $[inf..m-1]$.

This process repeats until the element is found or the search range becomes empty ($inf > sup$).

Algorithm Binary Search Algorithm

```
Constant n = 5
Variable tab : array[1..n] of real
    inf, sup, m : integer
    X : real
    found : boolean
begin
begin
    Write("Enter the element to search for: ")
    Read(X)
    inf ← 1
    sup ← n
    found ← False
    while (inf ≤ sup) AND (NOT found) do
    begin
        m ← (inf + sup) div 2
        if tab[m] = X then
        begin
            found ← True
        end
        else
        begin
            if tab[m] < X then
            begin
                inf ← m + 1
            end
            else
            begin
                sup ← m − 1
            end
        end
    end
    if found then
    begin
        Write("Element ", X, " exists at position: ", m)
    end
    else
    begin
        Write("Element ", X, " does not exist in the array")
    end
end
```

Exemple 4.6 Consider a sorted array of 5 elements: $[2, 4, 7, 10, 15]$. We will illustrate the search for values $X = 7$ and $X = 3$.

The trace for $X = 7$ (Figure 4.8) shows a successful search at position 4. Con-

Figure 4.7: Representation of the array for binary search.

versely, for $X = 3$ (Figure 4.9), the range eventually becomes empty, confirming the element's absence.

Figure 4.8: Trace for the search of value 7 (Successful).

Figure 4.9: Trace for the search of value 3 (Unsuccessful).

Efficiency Evaluation

The efficiency of binary search is remarkable. Because the search range is divided by two at each iteration, the maximum number of steps for an array of size n is approximately $\log_2(n)$.

For comparison, given an array of size 2^{30} (over one billion elements):

- **Sequential search** requires, in the worst case, $1,073,741,824$ iterations.
- **Binary search** requires, at most, only **30 iterations**.

The performance difference is monumental.

4.5 Two-Dimensional Arrays

If we return to the problem posed in Section 4.1, but this time, suppose we want to process and store the grades for several subjects for a class of 30 students.

Should we use multiple arrays, associating one array with each subject? It is possible, but manipulating several arrays for the same problem can be quite rigid!

Another perspective on the problem consists of thinking in dimensions. In other words, we can add a dimension for the subjects in the same way we added a dimension for the students.

In this case, we refer to a *two-dimensional array* (or simply a *matrix*), with one dimension for students and another for subjects.

Two-dimensional arrays are declared in the following form:

From a representation standpoint, it is standard that *dimension1* corresponds to the rows and *dimension2* corresponds to the columns.

Here is an illustrative example of declaring a two-dimensional array named *subj_grades* which consists of 30 rows for students and 8 columns for subjects (Example 4.7):

Exemple 4.7
Variable subj_grades : **array** [1..30, 1..8] **of real**

Variable Array_Name : **array** [dimension1, dimension2] of **component_type**

Figure 4.10: Graphical representation of a 2-dimensional array.

Figure 4.10 is a graphical representation of the declaration in Example 4.7.

Each element of the matrix is identified by two indices: a row index and a column index. Thus, *subj_grades[3, 5]* refers to the element located in the 3rd row and the 5th column.

The notation *subj_grades[i, j]* is a generalization to 2 indices of the concept of the indexed variable introduced in Section 4.3.

4.6 Using a Two-Dimensional Array

As with one-dimensional arrays, accessing an element of a two-dimensional array relies on the concept of an *indexed variable*.

Recall that an indexed variable is used just like a simple variable. The following examples demonstrate the various ways to manipulate two-dimensional arrays:

Assigning Values

Assigning a value to an element in a two-dimensional array is done using the indexed variable concept introduced in Section 4.5. This operation is illustrated in Example 4.8.

Exemple 4.8

```
subj_grades[1,1] ← 15.10
subj_grades[27,8] ← 13.00
```

Reading Elements

To read elements into a two-dimensional array, it is necessary to use two nested loops, each corresponding to a dimension. Example 4.9 shows how to read elements into the matrix *subj_grades*.

Note that in Example 4.9, the reading is done row by row, although it is also possible to perform the same process column by column. This simply requires swapping the nesting order of the loops.

Exemple 4.9

```
For i from 1 to 30 do
begin
    For j from 1 to 8 do
    begin
        Write ('enter a grade')
        Read (subj_grades[i, j])
    End
End
```

Writing Elements

As with reading, writing (or populating/displaying) a two-dimensional array is performed using two nested loops. The writing operation is shown in Example 4.10.

Exemple 4.10

```
For i from 1 to 30 do
begin
    For j from 1 to 8 do
    begin
        Write ('The grade at index [', i, ', ' , j, '] is equal to: ' ,
 subj_grades[i,j])
    End
End
```

4.7 Strings

In this section, we first discuss the character type to set the stage for the study of strings.

4.7.1 Character Type

A character is the basic element of a string. It is used to designate any symbol that can appear in a text. In other words, this domain includes:

- Alphabetic letters (uppercase + lowercase): 'A', 'c', ...
- Numerical characters (digits): '3', '8', ...
- Special characters: ',', '?', '#', '$', ...
- The space character (blank): ' '.

Values of the character type are ordered according to the internal character codes (ASCII, IBM-PC, ISO-xxx, ...).

Character Representation

Recall that regardless of the nature of the information (image, sound, text, video) processed by a computer, it is always represented as a sequence of binary elements (bits).

In the case of characters, a correspondence must be established between the character-type elements and a binary representation.

Table 4.1 summarizes examples of text data encoding standards that have evolved over time.

Table 4.1: Examples of character encoding standards.

Bits	Characters	Name	Remarks
6	64	Display Code	Allows representing the 26 uppercase Latin letters, digits 0..9, and punctuation symbols: , : ; . ( ) .... Sufficient for writing programs and printing results.
7	128	ASCII	(American Standard Code for Information Interchange), adapted to the American language: uppercase and lowercase letters without accents, digits, punctuation symbols. Widely used on Intel processors.
8	256	EBCDIC	(Extended Binary Coded Decimal Interchange Code), commonly used on mainframes, particularly IBM family systems.
16	65536	UNICODE	(UNIversal CODE), a unique standard to represent all characters used globally, including ideograms. Used notably on Pentium-type processors.

ASCII Encoding

Base ASCII is a 7-bit code defining a set of 128 characters (Figure 4.11). It is used in most personal computers and workstations.

Figure 4.11: ASCII Code

ASCII code includes:

- The 33 control characters (codes 0 to 31 and 127) are non-printable characters used for actions such as:
  - CR: Carriage Return
  - BEL: Bell (audible beep)

  - ESC: Escape

- Codes 65 to 90 represent uppercase Latin letters.
- Codes 97 to 122 represent lowercase Latin letters (Adding 32 to the decimal ASCII code converts uppercase to lowercase).
- Code 32 represents the space (SP: Space).
- Codes 48 to 57 represent the digits.

ASCII Extensions

ASCII was developed for the English language. While it handles technical texts in languages using the Latin alphabet, it lacks accented characters. In the 1990s, the ISO 8859 standard was created as an 8-bit extension of ASCII.

- The first 128 characters are identical to ASCII.
- The following 128 are language-specific.

For example, ISO 8859-6 is used for Arabic, while ISO 8859-1 or ISO 8859-15 are used for French accents.

Representation of Character Constants

A character constant with a graphic representation (ASCII 32 to 126) is represented by a single character enclosed in single quotes: 'a', 'D', '3', '*', '!', ... This notation distinguishes character constants from variables and numerical constants.

Specifically, the space character (ASCII 32) is written as ' '. The apostrophe is conventionally written by doubling it: '''' (four apostrophes in total).

Operators on the Character Type

The character type is a scalar type; its elements are ordered by their internal codes (ASCII). Therefore, the following can be applied:

- Relational operators: <, >, <>, =, <=, >=.
- Succession operators: *succ* and *pred* return the successor and predecessor characters respectively.
  - *succ*('A') = 'B'
  - *pred*('C') = 'B'

Additionally, conversion operators *chr* and *ord* can be used:

- The *chr* function returns the character associated with a decimal code (*chr*(65) returns 'A').
- The *ord* function returns the decimal machine code of a character (*ord*('C') returns 67).

Declaration Examples

Example 4.11 shows the declaration of a character constant (*my_choice*) and a character variable (*your_choice*).

Exemple 4.11
Constant my_choice = 'y'
Variable your_choice : **char**

4.7.2 String Type

In Section 4.7.1, we saw that the *char* type allows for the manipulation of single characters. In many applications, we need to handle sequences of characters representing entities such as a first name, an email address, or even a text document.

One solution is to use an array of characters. However, arrays have limitations regarding their static length and their rigidity when manipulating sequences of characters. Hence, it is necessary to design another data type that overcomes these limitations. Such a type is called a *string*.

The string type is designated by the reserved word *string*. Its domain consists of the set of all possible sequences of characters (including the empty sequence). These sequences are enclosed in single quotes and can consist of 0, 1, or several characters, as illustrated in Example 4.12.

Exemple 4.12

- 'This is a string'
- 'D'
- ''

Operators on the String Type

- **Relational operators** (<, >, <>, =, <=, >=): As with individual characters, the comparison of strings is based on the order of the internal character codes (lexicographical order). Two strings are equal when they have the same length and consist of the same sequence of characters.

- **Concatenation**: The *concat* function is used to join two strings together. For example, *concat*('University of ', 'Ghardaia') results in the string 'University of Ghardaia'.

- **Length**: The *length* function provides the number of characters in a string. For example, *length*('University of Ghardaia') returns the integer value 22.

Declaration Examples

Example 4.13 shows the declaration of a string constant (*message*) and a string variable (*name*).

Exemple 4.13

Constant message = 'I am learning programming'

Variable name : **string**

4.8 Pascal Language Corner

4.8.1 One-Dimensional Arrays

The reserved word *array* is used to declare an array of data of the same type. To declare variables of the array type, we can proceed as follows:

1. Declare an array type:

```
type type_name = array[index_type] of component_type;
```

where *index_type* is any ordinal type and *component_type* is the type of data stored in the array. It can be any type, including another array type (except for the file type, which will be discussed in the second part of the *Algorithms and Data Structures* series).

Exemple 4.14

```
type vector = array[1..50] of real;

```

In example 4.14, *index_type* = 1..50 is a subrange type, which is an ordinal type.

2. Declare a variable of the array type.

In this second step, we use the array type, such as the one declared in the previous step.

```
var var_name1, var_name2, ... : array_type_name;
```

Exemple 4.15

```
    var X, Y : vector;

```

Another way to declare an array is to proceed directly as follows:

```
var array_name : array[index_type] of component_type;
```

Exemple 4.16

```
    var X, Y : array[1..50] of real;

```

Example 4.17 shows an array declaration where *index_type* is an enumerated type.

Exemple 4.17

```
    type day = (saturday, sunday, monday, tuesday,
   wednesday, thursday, friday)
    var day_revenue : array[day] of real;

```

Note that the memory space reserved for an array consists of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

4.8.2 Two-Dimensional Arrays

Two-dimensional arrays can be declared in two ways:

Either as an array of arrays:

```
array[index_type1] of array [index_type2] of component_type
```

Or as a matrix:

```
array[index_type1, index_type2] of component_type
```

These two ways of declaring two-dimensional arrays are illustrated in Example 4.18.

Exemple 4.18

```
1    var subj_grades1 : array[1..30] of array [1..8] of
     real;
2    var subj_grades2 : array[1..30, 1..8] of real;
3
```

Thus, it is possible to access an array element in two ways:

```
1  array_name [index1] [index2]
```

or

```
1  array_name [index1, index2]
```

Note that the concept of two-dimensional arrays can be generalized to multi-dimensional arrays.

4.8.3 Strings

The declaration of a string is done using the reserved word *string*.

```
1  var var_name : string[n];
```

In this case, the string size is limited to n characters, where n is an integer between 1 and 255.

Exemple 4.19

```
1    var last_name, first_name : string[30];
2
```

The declaration in Example 4.19 means that the string variables *last_name* and *first_name* can contain a maximum of 30 characters.

Another variant for declaring strings consists of omitting the specification of the maximum string size.

```
1  var var_name : string;
```

In such a situation, the string variable can have a default length of 255 characters.

String Functions

Table 4.2 describes the predefined functions in Turbo Pascal related to strings.

Table 4.2: String functions.

Function	Description	Example
length(S : string) : Integer	Returns the actual length of a string.	Writeln('Length = ', length(S));
concat(S1, [, S2, ..., SN] : string) : string	Concatenation of several strings.	S := concat('Info', 'ASD1');
pos(SS, S : string) : byte	Searches for the substring *SS* in string *S* and returns the integer value representing the position of the first character of *SS* in *S*. Returns zero if *SS* is not found.	if pos(' ', S) > 0 then ...;
copy(S: string, index : integer, count : integer) : string	Returns a substring of length *count* starting from position *index* in string *S*.	S := copy(S, 2, 3);

String Procedures

Table 4.3 describes the predefined procedures in Turbo Pascal related to strings.

Table 4.3: String procedures.

Procedure	Description	Example
delete(var S : string; index : integer; count : integer)	Removes a substring of length *count* starting from position *index* in string *S*. If the resulting string exceeds 255 characters, it is truncated.	Writeln(delete(S,4,5));
insert(source : string; var S : string; index : integer)	Inserts the *source* substring into string *S* at position *index*.	insert('max ', S, 10);
str(X [: size [: decimals]]; var S : string) : string	Converts a numerical value X (integer or real) into its string representation according to *size* and *decimals* parameters.	str(425.12, S); str(425.12:5:2, S);
val (S : string; var v; var code : integer)	Converts string *S* representing a number into its numerical value and stores it in **v**. If invalid, *code* contains the index of the offending character; otherwise, *code* is zero.	val ('12564', v, err); if (err > 0) then ...;

4.9 C Language Corner

4.9.1 One-Dimensional Arrays

Syntax

```
type Identifier [size];
```

where *size* is a positive constant expression indicating the number of elements in the array. An array is always indexed from 0 to *size - 1*.

Unlike Pascal, the C language does not offer any possibility for global assignment of arrays.

Note: It is possible to initialize an array during its declaration as follows:

```
type Identifier [size] = {V1, V2, ..., Vn};
```

Example 4.20 presents different ways to initialize arrays.

Exemple 4.20

```
#define N 4
int main()
{
  int T1[N] = {4, 10, -1, 35};
  float T2[N] = {1.6};
  char T3[N] = {'A', ' ', ' ', 'D'};
  int T4[] = {12, 0, 4};
  return 0;
}
```

All elements of array *T1* are initialized. Only the first element of array *T2* is initialized. For array *T3*, the first and last elements are initialized. As for *T4*, this represents the creation and initialization of an array with three elements.

The following notation is used to access an element of a one-dimensional array:

```
T1[0] = T4[1];
T2[2] = 3.0;
...
```

4.9.2 Two-Dimensional Arrays

Like many programming languages, C allows the definition of multi-dimensional arrays. For instance, the declaration of a two-dimensional array is done as follows:

```
type Identifier [row_size][column_size];
```

It is also possible to initialize a multi-dimensional array as illustrated in Example 4.21.

Exemple 4.21

```
#define N 4
#define M 4
int tab[N][M] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};

```

An array element is referenced by specifying the indices of the dimensions as shown in the following notations:

```
tab[3][4] = 12;
...
tab[i][j] = tab[0][0];
...
```

4.9.3 Strings

The C language does not have a "true" *string* type. Instead, a string is a one-dimensional array of characters that benefits from specific processing treatments.

The syntax for declaring and eventually initializing a string is as follows:

```
char Identifier[size] = "text\0";
```

Exemple 4.22

```
char last_name[20];
char first_name[20] = "text\0";

```

Note that it is also possible to declare a string as a *pointer* to *char* (Pointers will be studied in the second volume, *Algorithms and Data Structures - Part 2*).

Accessing a specific character in a string is done in the same way as accessing an element in an array. For example, *first_name[0] = 'F'*.

For string manipulation, we have already seen (Section 3.8.6) the *printf* and *scanf* functions along with conversion specifiers.

In addition, the header file *string.h* contains declarations allowing for string manipulation as illustrated in Table 4.4.

Table 4.4: Basic functions for string manipulation in C.

Function	**Result Type**	**Description**
strlen(s)	integer	Returns the length of the string.
strcpy(s, t)	string	Copies string t into string s and returns s.
strcat(s, t)	string	Appends string t to the end of string s and returns s.
strcmp(s, t)	integer	Compares the two strings s and t. Returns 0 if $s = t$, another value otherwise.
strncpy(s, t, n)	string	Copies at most n characters from t to s and returns s.
strncat(s, t, n)	string	Appends at most n characters from t to the end of s and returns s.

4.10 Exercises

Exercise 4.1 What results will this algorithm provide?

Algorithm Tab1

```
Variable number : array[1..5] of integer
         i : integer

begin
    For i from 1 to 5 do
        number[i] ← i * i
    For i from 1 to 5 do
        Write (number[i])
end
```

Indications In this type of question, you are required to trace the algorithm.

Exercise 4.2 What results will this algorithm provide when given the following input values: 2, 5, 3, 10, 4 and 2?

Algorithm Tab2

```
Variable c : array[1..6] of integer
         i : integer

begin
    For i from 1 to 6 do
        Read (c[i])
    For i from 1 to 6 do
        c[i] ← c[i] * c[i]
    For i from 1 to 3 do
        Write (c[i])
    For i from 4 to 6 do
        Write (2 * c[i])
end
```

Indications You are required to trace the algorithm.

Exercise 4.3 What does this algorithm provide?

```
Algorithm Tab_Sequence
  Variable sequence : array[1..8] of integer
           i : integer

  begin
      sequence[1] ← 1
      sequence[2] ← 1
      For i from 3 to 8 do
          sequence[i] ← sequence[i-1] + sequence[i-2]
      For i from 1 to 8 do
          Write (sequence[i])
  end
```

Indications In this type of question, you are required to trace the algorithm.

Exercise 4.4 Write an algorithm that fills an array randomly and then displays it.

Indications You may assume you have a function *random(n)* that provides an integer in the interval $[0, n[$.

Exercise 4.5 Write an algorithm that determines the maximum value and its position in an array of size N.

Exercise 4.6 For an array of integers of size N, write an algorithm that calculates the following values:

- The count of positive numbers in the array.
- The count of negative numbers.
- The average of the positive numbers.
- The average of the negative numbers.

Indications Be careful with division by zero.

Exercise 4.7 Write an algorithm that declares and fills an array containing the six vowels of the Latin alphabet.

Exercise 4.8 Write an algorithm that reads a character and indicates if it is a vowel, using an array containing the 6 vowels of the alphabet.

Indications This is a follow-up to Exercise 4.7.

Exercise 4.9 Write an algorithm that determines the number of occurrences of a given character in a character array.

Exercise 4.10 Write an algorithm that constructs a new array (Table 4.7) from two arrays (Tables 4.5 and 4.6) of the same length that have been previously entered. The new array will be the sum of the corresponding elements of the two starting arrays.

Table 4.5: Array 1.

5	**8**	**7**	**9**	**1**	**5**	**4**	**6**

Table 4.6: Array 2.

7	**6**	**5**	**2**	**1**	**3**	**7**	**4**

Table 4.7: Resulting Array.

12	**14**	**12**	**11**	**2**	**8**	**11**	**10**

Exercise 4.11 What will be the results provided by this program?

Algorithm mat1

```
Variable mat : array[1..8, 1..2] of integer
           k, m : integer

begin
        For k from 1 to 4 do
            For m from 1 to 2 do
                mat[k, m] ← k + m
        For k from 1 to 4 do
            For m from 1 to 2 do
                Write (mat[k, m])
end
```

Exercise 4.12 Given the declaration:

Variable x : **array**[1..2, 1..3] **of integer**

Write an algorithm that reads 6 values for array **x**, requesting them "row by row", and then rewrites them "column by column", as in:

enter the values for row number 1
5 9 7
enter the values for row number 2
8 10 3
here is column number 1
5
8
here is column number 2
9
10
here is column number 3
7
3

Exercise 4.13 Write an algorithm to determine the position of the largest element in a two-dimensional array. Specifically, store the two indices of this largest element in integer variables named *imax* and *jmax*.

Exercise 4.14 Given an array of integers with N rows and P columns. Write an algorithm that fills this array and calculates the average of its values.

Exercise 4.15 Write an algorithm that determines the transpose of an $(N \times P)$ matrix.

Indications The transpose of a matrix is obtained by swapping its rows and columns.

Exercise 4.16 Write an algorithm that removes all spaces from a given sentence.

Exercise 4.17 The mirror word of a given word is obtained by reading it backwards (example: *emhtirogla* is the mirror word of *algorithme*). Write an algorithm to generate the mirror word of a given word.

Exercise 4.18 Write an algorithm that checks for a palindrome word in a character array. A palindrome is a word that reads the same forwards and backwards. (Example: *radar*, *level*)

Exercise 4.19 Write an algorithm that reads a sentence character by character (ending with a period) and identifies and displays all the digit characters that appear in that sentence.

Exercise 4.20 Write an algorithm that reads a sentence character by character (ending with a period) and determines and displays the lowercase alphabetic letters that **do not** appear in the sentence.

Exercise 4.21 A text may hide, in order, the letters of a word. For example: *'My early red car is inside'* contains the word *'Merci'*. Write an algorithm that determines whether a suggested text contains a given word or not.

5. Custom Types

Up to this point, in Chapters 3 and 4, we have dealt with simple data types (integer, boolean, character, and real) and structured data types (arrays and strings).

These data types share two common factors. The first is that they are predefined. *In other words, they do not need a definition before use. The second common factor is that they are* homogeneous*; they consist of a fixed number of information pieces of the same type.*

However, in real-world applications, an algorithm designer may need to manipulate other kinds of information beyond predefined types, such as enumerated, subrange, record, *and* set *types. The definition and manipulation of these custom data types are the focus of this chapter.*

Specifically, in Section 5.6, other data types specific to the C language are covered, namely the union *type and* bit fields.

5.1 Enumerations

Définition 5.1 The enumerated type defines an ordered set of values designated by constant identifiers (maximum 256). The order is established by the sequence in which the identifiers are listed.

The declaration of an enumerated type follows this syntax:

```
Type type_name = (element_0, element_1, ..., element_n-1)
```

The rank (or ordinal value) of an enumerated constant is determined by its position in the list. In this case, *element_0* has a rank of 0.

Exemple 5.1
Type
Color = (Red, Green, Blue)
Days = (Friday, Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday)
Months = (January, February, March, April, May, June, July, August, September, October, November, December)

Based on the declarations in Example 5.1, *Red*, *Friday*, and *January* are constants of the types *Color*, *Days*, and *Months*, respectively.

The function *Ord* applied to an enumerated type constant returns its rank. For example, *Ord*(Green) returns 1.

5.2 Subranges

Définition 5.2 A subrange type is a subset of values from an existing scalar type, known as the host type.
Defining a subrange type involves specifying the lower and upper bounds of the range.

The declaration of a subrange type is performed as follows:

Type type_name = constant .. constant

Both constants must belong to the same scalar type and represent the lower and upper bounds of the interval.

Exemple 5.2
Type
Letter = 'a'..'t'
Working_Days = Sunday..Thursday
Month_Num = 1..12
First_Quarter = January..March

In the declarations of Example 5.2, the host type for the subrange *Letter* is the scalar type *character*, and 'a' is less than or equal to 't'. Similarly, the host type for the subrange *First_Quarter* is the enumerated type *Months* defined in Example 5.1, where *January* is less than or equal to *March*.

5.3 Records

Définition 5.3 A record is a collection of data pieces of different types, called fields, which can be accessed individually or collectively for reading and writing.

Exemple 5.3

A date is composed of:

- a day (1 to 31),
- a month (1 to 12),
- a year (0 to 9999).

An employee is defined by a set of information:

- last name, first name (strings),
- date of birth (date),
- gender (character or boolean),
- number of children (integer).

The general form for declaring a record is as follows:

```
Type type_name = record
   field_name_1 : type_1
   field_name_2 : type_2
   ⋮
   field_name_m : type_m
   end
```

Example 5.4 illustrates the declaration of the *date* and *employee* types from Example 5.3.

Exemple 5.4

```
Type Tdate = record
      day : 1..31
      month : 1..12
      year : 0..9999
      end
```

```
Type Temployee = record
    last_name, first_name : string
    birth_date : Tdate
    gender : boolean
    num_children : integer
    end
```

Thus, we can define the variables *today*, *tomorrow*, and *yesterday* of type **Tdate**:

```
Variable today, tomorrow, yesterday : Tdate
```

Similarly, we can declare the variables *tenured_employee* and *person* of type **Temployee**:

```
Variable tenured_employee, person : Temployee
```

After declaring a record variable, we can access a field by specifying the record name followed by the field name(s), separated by the dot operator `.` (Example 5.5).

Exemple 5.5

```
yesterday.day ← 2
tomorrow.month ← today.month
tenured_employee.last_name ← 'Foulane'
person.birth_date.year ← 2020
```

5.4 Sets

Définition 5.4 A set is an unordered collection of elements of the same type, with a finite number of elements, upon which standard mathematical operations and relations can be performed: union, intersection, complement, equality, inclusion, membership, ...

Declaring a Set Type

A set type can be defined from a base type for its elements in the following manner:

```
Type set_type_name = set    of    base type
```

Thus, a set variable *set_var* can be declared as follows:

Variable set_var : **set_type_name**
or
Variable set_var : **set** of **base type**

A variable of type **set_type_name** can take as values any subset of the **base type**. Example 5.6 illustrates the declaration of set variables and types.

Exemple 5.6

```
Type color = (blue, red, yellow)          /* enumerated type */
     T_set_c = set of color          /* set type */

Variable green, black : T_set_c
Variable white, purple : set of color
```

Here, the type **color** is an enumerated type; it serves as the base type for the set type **T_set_c**. The variables *green*, *black*, *white*, and *purple* are of the same type despite being declared differently. They can take any subset of the **base type** as values. In Example 5.6, this corresponds to the set of all possible color combinations: [], [*blue*], [*red*], [*yellow*], [*blue*, *red*], [*blue*, *yellow*], [*red*, *yellow*], [*blue*, *red*, *yellow*].

Example 5.7 shows how to assign values to set variables.

Exemple 5.7

```
black ← [blue, red, yellow]
white ← []                          /* [] represents the empty set */
purple ← [blue, red]
green ← [blue, yellow]
```

Physical Representation of a Set

The storage in memory of a set variable is performed in a contiguous array where the presence of each element is marked using a single bit (1 for a present element, 0 otherwise).

To illustrate the physical representation of a set type, consider Example 5.8.

Exemple 5.8

```
Type quality = (active, intelligent, athletic, courageous, skillful)
     personality = set of quality

Variable A : personality
```

The variable A is of type set. If $A = [intelligent, skillful]$, its physical representation is illustrated by Table 5.1.

Table 5.1: Physical representation of a set type variable.

active	intelligent	athletic	courageous	skillful
0	1	0	0	1

Operations on Sets

Let sets $A = [1,3]$ and $B = [3,6]$, Table 5.2 presents the possible operations on sets.
[Image of Venn diagrams for union, intersection, and difference of two sets]

Table 5.2: Set operations.

Mathematical Symbol	Algorithmic Symbol	Description	Example
$\cup$	+	Union	$A + B = [1,3,6]$
$\cap$	*	Intersection	$A * B = [3]$
$-$ or \	-	Difference	$A - B = [1]$
=	=	Equality	$A = [1,3]$
$\neq$	<>	Inequality	$A <> B$
$\subset$	<	Strict inclusion	$[3] < B$
$\subseteq$	<=	Inclusion	$[3,6] <= B$
$\supset$	>	Strict containment	$A > [3]$
$\supseteq$	>=	Containment	$A >= [3,1]$
$\in$	in	Membership	3 in B

5.5 Pascal Language Corner

5.5.1 Creating a New Type

Generally, the syntax for declaring a new type is as follows:

```
1  TYPE identifier_1, identifier_2, ... : Type_Name;
```

This declaration is used to define data types such as *identifier_1*, *identifier_2*, ..., as illustrated in Example 5.9.

Exemple 5.9

```
1  TYPE    int1, int2, int3 : Integer;
2  car, character : Char;
3
```

5.5.2 Enumerated Type

The syntax for declaring an enumerated type is:

```
TYPE enumerated_identifier = (item1, item2, ...);
```

Here are some examples of enumerated type declarations (Example 5.10):

Exemple 5.10

```
TYPE COLORS = (Red, Green, Blue, Yellow, Magenta,
Cyan, Black, White);

TRANSPORT = (Bus, Train, Airplane, Ship);
```

5.5.3 Subrange Type

The syntax for declaring a subrange type is as follows:

```
TYPE subrange_identifier = lower_bound .. upper_bound;
```

The following examples show some subrange type declarations (Example 5.11):

Exemple 5.11

```
TYPE number = 1 .. 100;
sub_color = Green .. Black;
sub_trans = Bus .. Airplane;
```

5.5.4 Records

The declaration of a record type is performed using the keyword *record*:

```
TYPE record_identifier = record
ident_list_1 : type_1;
ident_list_2 : type_2;
```

```
    ...
    ident_list_n : type_n
end;
```

Where *ident_list* represents one or more identifiers separated by commas.

Example 5.12 presents a record type declaration:

Exemple 5.12

```
    TYPE    exp = record
    X, Y : real;
    B : boolean;
    S : string[10];
    T : array [1..10] of integer
    end;

```

To simplify access to the fields of a record, we can use the *with* statement, as illustrated in Example 5.13:

Exemple 5.13

```
    with exp do begin
    writeln ('X = ', X);
    writeln ('Y = ', Y);
    if B then
    S := 'male'
    else
    S := 'female';
    end;

```

5.5.5 Sets

The set type is defined using the reserved word *set* as follows:

```
TYPE    set_identifier = set of base_type;
```

The *base_type* must not contain more than 256 possible values.

Set type variables can be declared in two ways. Either by using an already declared set type:

```
var v_set1, v_set2, ... : set_identifier;
```

Or by defining the set type directly within the *var* clause itself:

```
var v_set1, v_set2, ... : set of base_type;
```

Here is an example of a set type declaration:

Exemple 5.14

```
    Type letters = set of char;
    var uppercase_letters : letters;

```

The declaration in Example 5.14

5.6 C Language Corner

5.6.1 Creating a New Type

In C, defining a new type is done using the *typedef* keyword in the following manner:

```
typedef int integer;
```

typedef is used to create an *alias (synonym)* ; here, the new type *integer* can be used in declarations, typecasting, etc., in the same way the *int* type is used.

Exemple 5.15

```
    typedef int integer;
    integer main()
    {
      integer i = 20;
      ...
      return 10;
    }

```

5.6.2 Structures

A structure (record) is defined using the reserved word *struct*:

```
struct structure_name
{
  type_1 field_1;
  type_2 field_2;
  ...
  type_n field_n;
};
```

Note the mandatory semicolon at the end of a structure definition.

Declaring a variable of a structure type is done as follows:

```
struct structure_name variable_name;
```

Example 5.16 illustrates how to declare structures and their associated variables.

Exemple 5.16

```
    struct point /* structure declaration */
    {
      int x;
      int y;
    };
    int main()
    {
      ...
      struct point pt; /* variable declaration of
  structure type */
      ...
    }

```

It is also possible to initialize a structure during its declaration in a sequential or selective manner (Example 5.17).

Exemple 5.17

```
    struct date /* structure declaration */
    {
      int day;
      int month;
      int year;
    };
    int main()
```

```
8     {
9       /* sequential initialization */
10      struct date birth_date = {1, 1, 2000};
11      /* selective initialization */
12      struct date rec_date = {.year = 2009, .day = 27, .
    month = 12};
13      /* mixed initialization: sequential + selective */
14      struct date def_date = {3, .year = 2018};
15    }
16
```

Accessing a field of a structure is achieved using the dot operator `.` as follows:

```
1  variable.field;
```

Example 5.18 clarifies the field access operation.

Exemple 5.18

```
1     struct date /* structure declaration */
2     {
3       int day;
4       int month;
5       int year;
6     };
7     int main()
8     {
9       struct date birth_date;
10      birth_date.day = 1;
11      birth_date.month = 3;
12      birth_date.year = 1969;
13    }
14
```

Furthermore, it is possible to define fields occupying a specific number of bits. This is applicable for integer types (*int* or *unsigned int*).

The use of *bit fields* is particularly useful in systems programming, which requires manipulating specific hardware registers.

For instance, the status register of the *MC 68060* can be described using a bit field structure:

```
1  struct state_reg
2  {
3    unsigned int trace : 2;
```

```
    unsigned int priv : 2;
    unsigned int : 1;              /* unused    */
    unsigned int mask : 3;
    unsigned int : 3;              /* unused    */
    unsigned int extend : 1;
    unsigned int negative : 1;
    unsigned int zero : 1;
    unsigned int overflow : 1;
    unsigned int carry : 1;
};
```

5.6.3 Unions

A *union* is a grouping of objects of different types that can only contain one of its members at a time. Like *struct* types, this type is declared using the keyword *union*:

```
union union_name
{
  type_1 member_1;
  type_2 member_2;
  ...
  type_n member_n;
};
```

Exemple 5.19

```
union day
{
  char letter;
  int num;
};
int main()
{
  union day yesterday, today, tomorrow;
  yesterday.letter = 'M';
  today.num = 5;
  tomorrow.num = (today.num + 2) % 7;
}

```

Accessing a member of a union is also done using the dot operator `.` (Example 5.19).

5.6.4 Enumerations

An enumeration is defined using the keyword *enum* as follows:

```
enum enum_name {const_1, const_2, ..., const_n};
```

Exemple 5.20

```
enum natural1 {Zero, One, Two, Three, Four, Five};
enum natural1 n = Two;
printf("n = %d.\n", (int)n);
printf("Four = %d.\n", Four);

```

In fact, the declaration in Example 5.20 creates two things: an enumerated type called *natural1* and constant identifiers *Zero*, *One*, ... encoded as integers 0,1,.... Consequently, the output is:

```
n = 2.
Four = 4.
```

However, it is possible to modify the value(s) of the constant(s) during type declaration:

```
enum natural3 {Six=6, Seven, Eight, Nine, Ten};
```

5.7 Exercises

Exercise 5.1 A medical claim form includes the following information concerning the insured: last name, first name, date and place of birth, personal address, name and address of the employer, and the payment method.

- Describe this custom data type.

Indications Note that you are working here with a composite type made of several entities called fields. Ensure that the values of the fields are atomic (not further decomposable).

Exercise 5.2 A complex number is defined by: $a + ib$.

- Read two complex numbers and display their product.
- Compare two complex numbers.

Indications

- To read a record, you must proceed field by field.
- In comparing two complex numbers, try to define an order relation (e.g., based on the modulus).

Exercise 5.3 In a library, there are 1000 index cards. The information found on the card for a library book is as follows: Code, author (last name and first name), title, publisher, and year of publication.

- Display the authors who published in the year 2011 with the publisher *BERTI*.

Indications You can use an array of records. To access the fields of a record, use the dot operator `.`.

Exercise 5.4 A student is identified by: student ID number, last name, first name, date of birth (decomposed into day, month, and year as integers), and an array containing the averages for 5 studied teaching units (as illustrated in Table 5.4).

1. Write in algorithmic notation, C, and Pascal the data structures necessary to define a student and a structure allowing the management of 1500 students. (The date of birth can be stored in a separate structure).
2. Write in algorithmic notation, C, and Pascal the following primitives:
   - Input for a student (filling the various fields of the structure).
   - Display of information concerning a student.
   - Filling the student array with 1 student (calling the Input_Student primitive).
   - Displaying all students present in the array (calling the Display_Student primitive).
   - Displaying a student searched by their last name in the array.

Table 5.3: Example of student data structure.

ID	Last Name	First Name	Birth Date	Unit Averages
10293	Smith	John	15/04/2002	[14, 12.5, 10, 16, 11]

Table 5.4: Student Record File

Student Number	10601234				
Student Last Name	FOULANE				
Student First Name	Elfoulani				
Date of Birth - birth day - birth month - birth year	 29 02 1988				
Grades Table	12	15	17	13	14

Indications Question No. 1 focuses on data. Here we need a composite type. Therefore, we use *records* and the technique of nested records.
Question No. 2 focuses on processing (processing a single student without worrying about the array, then processing one or more students within the student array).

Exercise 5.5 Write a function that determines whether a character is a letter, a digit, a blank (space), a punctuation character, or another type of character.

Indications You can use the *set* type.

Exercise 5.6 Write a function that determines whether a given color is contained in the national flag or not, by testing membership in a set of chosen colors.

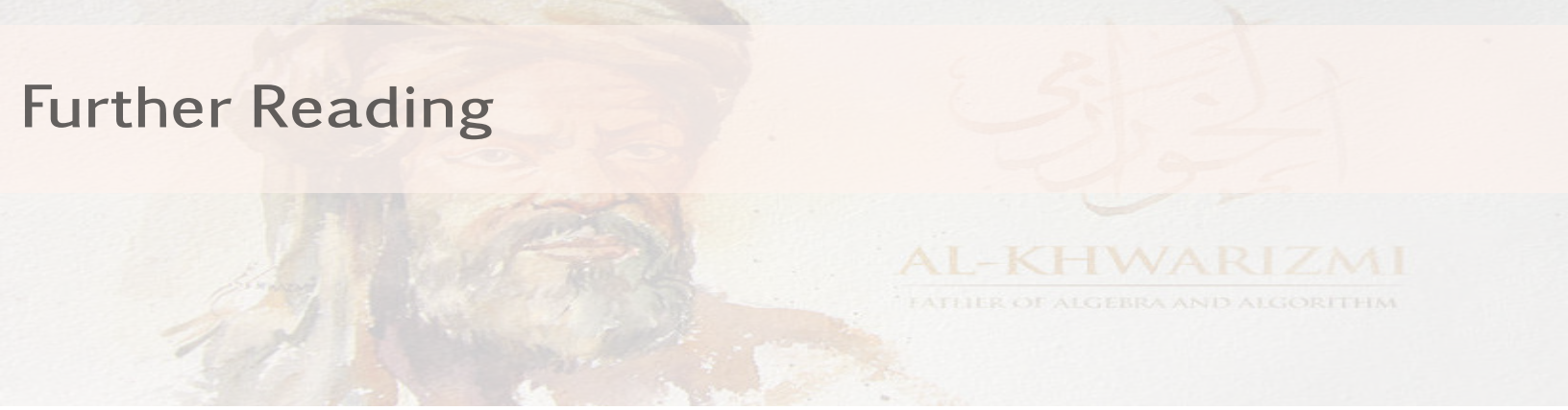
Indications You can use the *set* type and the *enumerated* type.

Exercise 5.7 A group of people is numbered from 1 to N. Write the variable declarations for ELDERLY_PEOPLE, MEN, SMOKERS whose value is a set of people from this group. Assuming that appropriate values have been assigned to these variables, write the expressions to obtain sets whose values are:

- The complete group of people.
- Non-smokers.
- Elderly men who smoke.
- All elderly people plus male smokers.

Indications Use the *set* type as well as operations on sets (union, intersection, difference, . . .).

Further Reading



[1] H. Baba-Hamed and S. Hocine. *Algorithmique et structures de données statiques : cours et exercices avec solutions*. Office des Publications Universitaires, 2006.

[2] Brahim Bessaa. *Exercices corrigés d'Algorithmique*. Les Fascicules du LMD. Pages Bleues, 2018.

[3] Aditya Y. Bhargava. *Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People*. 2nd. Shelter Island, NY: Manning Publications, 2024.

[4] Doug Cooper and Michael Clancy. *Oh! Pascal!* 3rd. New York, NY: W. W. Norton & Company, 1993.

[5] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th. Cambridge, Massachusetts: MIT Press, 2022.

[6] Claude Delannoy. *Programmer en Turbo Pascal 7*. Eyrolles, Neuvième tirage 2007.

[7] Claude Delannoy. *S'initier à la programmation: Avec des exemples en C, C++, C#, Java et PHP*. Eyrolles, 2008.

[8] David Griffiths and Dawn Griffiths. *Head First C: A Brain-Friendly Guide*. Sebastopol, CA: O'Reilly Media, 2012.

[9] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Englewood Cliffs, NJ: Prentice Hall, 1988. ISBN: 978-0131103627.

[10] Stephen G. Kochan. *Programming in C*. 4th. Upper Saddle River, NJ: Addison-Wesley Professional, 2014.

[11] Jean-Pierre Laurent and Jacqueline Ayel. *exercices commentés d'analyse et de programmation*. Dunod, 1985.

[12] Sanford Leestma and Larry Nyhoff. *Pascal: Programming and Problem Solving*. 4th. New York, NY: Macmillan, 1993.

[13] Anany Levitin. *Introduction to the Design and Analysis of Algorithms*. 3rd. Boston, MA: Pearson, 2011.

[14] Steve Oualline. *Practical C Programming*. 3rd. Sebastopol, CA: O'Reilly Media, 1997. ISBN: 978-1565923065.

[15] Chantal Richard and Patrice Richard. *initiation à l'ALGORITHMIQUE : 135 exercices corrigés.* Collection DIA. Belin, 1985.

[16] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011, pp. 1–955.

[17] Reema Thareja. *Data Structures Using C*. 2nd. New Delhi, India: Oxford University Press, 2014.

[18] Clovis L. Tondo and Scott E. Gimpel. *The C Answer Book: Solutions to the Exercises in 'The C Programming Language'*. 2nd. Englewood Cliffs, NJ: Prentice Hall, 1989.

[19] Sylvie Tormento and Sophie Boutin. *Algorithmes cours et exercices*. BTS informatique de gestion. Bréal, 1989.

[20] Philippe Trigano. *Méthodologie de la Programmation & Management des Informations*. BERTI, 1992.

[21] *Turbo Pascal : Guide de référence*. Borland, 1991.

[22] *Turbo Pascal : Guide du programmeur*. Borland, 1991.

[23] Djamel-Eddine Zegour. *Apprendre et enseigner l'algorithmique - Tome 1 : Cours et annexes.* Livre en ligne.

[24] Djamel-Eddine Zegour. *Apprendre et enseigner l'algorithmique - Tome 2 : Sujets d'examen corrigés.* Livre en ligne.