People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

Abdelhamid Ibn Badis University of Mostaganem

Faculty of Science and Technology

Course handout:

Computer Science I

Elaborated by:

M. BOUHARAOUA Farouk

Reviewed by:

Dr. DEJBBARA Mohamed Redha

Dr. ROUBA Baroudi

Academic Year:2025-2026

Table of Contents

Foreword

This course handout is the support material for the "Computer Science I" course, intended for first-year students in the Common Core Science and Technology (ST) program.

This course requires no prerequisites and aims to introduce general principles of computer science and to explore and understand the basic concepts of algorithms and programming.

This handout is structured into 6 chapters that cover:

- General principles of computer science (chapters 1 and 2)
- The structure of an algorithm, elementary data types, and elementary instructions (chapter 3).
- Control structures: conditional and iterative (chapters 4, 5, and 6).

To facilitate the translation of algorithms into programs, the presentation of the algorithm chapters follows a Pascalian style. Therefore, we have chosen the Pascal language for writing the programs. All syntax is presented in both algorithmic and Pascal formats. And most of the algorithms are translated into Pascal programs.

The concepts covered in this course are explained in varying degrees of detail, but the chapters as a whole provide a basic foundation for any novice programmer.

We wrote this course using various recognized works in the field and online resources. A list of bibliographic references is included at the end of the course materials.

Keywords: Computer; Numeral system; Algorithm; Program; Pascal Langage; Control structures.

Préface

Ce polycopié constitue le support du cours « Informatique I », destiné aux étudiants de première année du programme du Tronc Commun Sciences et Technologies (ST).

Ce cours ne requiert aucun prérequis et vise à introduire les principes généraux de l'informatique, ainsi qu'à explorer et comprendre les concepts fondamentaux de l'algorithmique et de la programmation.

Ce polycopié est structuré en 6 chapitres couvrant :

- Les principes généraux de l'informatique (chapitres 1 et 2).
- La structure d'un algorithme, les types de données élémentaires et les instructions de base (chapitre 3).
- Les structures de contrôle : conditionnelles et itératives (chapitres 4, 5 et 6)

Afin de faciliter la traduction des algorithmes en programmes, la présentation des chapitres consacrés aux algorithmes suit un style pascalien. C'est pourquoi nous avons choisi le langage Pascal pour l'écriture des programmes. Toute la syntaxe est présentée à la fois en format algorithmique et en format Pascal, et la plupart des algorithmes sont traduits en programmes Pascal.

Les concepts abordés dans ce cours sont expliqués avec des niveaux de détail variables, mais l'ensemble des chapitres fournit une base solide pour tout débutant en programmation.

Nous avons rédigé ce cours à partir de divers ouvrages reconnus dans le domaine, ainsi que de ressources en ligne. Une liste de références bibliographiques est incluse à la fin du document.

Mots Clés : Ordinateur ; système de numération ; Algorithme, Programme, Langage Pascal ; Structures de contrôle.

المقدمة

هذا الكُتيّب الدعمَ البيداغوجي لمقياس «معلوماتية 1» موجَّ لطلبة السنة الأولى في الجذع المشترك علوم وتكنولوجيا.

لا يتطلّب هذا المقياس أيَّ معارف مسبقة، ويهدف إلى تقديم المبادئ العامة لعلوم الحاسوب، إضافة إلى استكشاف وفهم المفاهيم الأساسية للخوارزميات والبرمجة.

يتم تنظيم هذا الكُتيّب في ستة فصول تتناول:

- المبادئ العامة لعلوم الحاسوب (الفصلان 1 و2).
- بنية الخوارزمية، وأنواع المعطيات الأساسية، والتعليمات الأولية (الفصل 3).
- بُنى التحكّم: البُنى الشرطية والبُنى التكرارية (الفصول 4، 5 و6).

ولتسهيل ترجمة الخوارزميات إلى برامج، تم تقديم فصول الخوارزميات بأسلوب قريب من أسلوب لغة الباسكال، ولهذا تم اختيار لغة Pascalلكتابة البرامج. وقد عُرضت جميع الصيغ النحوية (Syntaxe) باللغة الخوارزمية وبصيغة الباسكال، كما تُرجمت أغلب الخوارزميات إلى برامج مكتوبة بلغة الباسكال.

تُعرض المفاهيم الواردة في هذا المقياس بدرجات متفاوتة من التفصيل، غير أن مجموعة الفصول تقدّم أساساً متيناً لأي مبتدئ في البرمجة.

وقد تم إعداد هذا المقياس بالاعتماد على مجموعة من المراجع المعروفة في المجال، بالإضافة إلى مصادر إلكترونية. وتوجد قائمة بالمراجع الببليوغرافية في نهاية الوثيقة.

الكلمات المفتاحية: الحاسوب؛ نظام الترقيم؛ الخوارزمية؛ البرنامج؛ لغة باسكال؛ هياكل التحكم.

Chapter 1: Introduction to Computer Science

1.1 What is computer science?

Computer science is the science of automating information processing. It is composed of the two words "**information** " and "**automatic .** "

1.2 The information

Information is an element of knowledge that can be communicated, stored, and even processed.
It is necessary to distinguish between the **form** and **meaning of information:**
The **form** represents the **code.**
Whereas **meaning** represents **the interpretation of this code.**
In everyday life, in order to be communicated, information is coded in various forms using signals or symbols of different types (numbers, letters, light signals, graphic symbols, etc.).
The meaning attributed to information is obtained through interpretation of the code established by humans.
The table below presents some examples.

Form of information	**Nature of the signal**	**Meaning associated with information**
Red light	Luminous (light)	Vehicles stop
Telephone ringtone	Sound	Someone wants to reach the subscriber whose phone is ringing.
01/14/2021	Alphanumeric characters	14th day of January, in the year 2021

1.3 Information processing

Automatic information processing by machine (computer, robot, cell phone , etc.) consists of having this machine produce information called ***results*** from known information called ***data*** .

The figure below represents a simplified diagram of information processing.

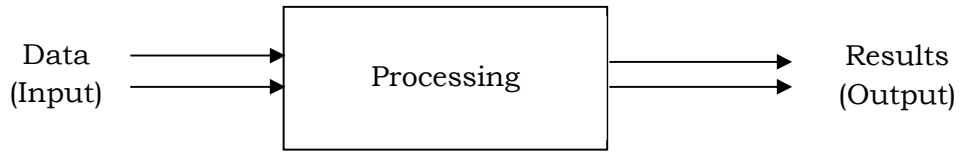


Fig. 1.1: Principle of information processing

1.4 Computer system

A computer system is composed of two complementary entities, both of which are necessary for the execution of the tasks assigned to it:

- **Hardware:** This is the entity that groups together all the electronic circuits and devices that make up the physical machine (computer).

- **Software:** This is the entity that groups together all the programs that must be introduced into the machine to make it perform a specific task (billing, inventory management, payroll calculation, etc.).

1.4.1 The hardware

This is the entire set of hardware that makes up the computer.
The computer is the machine that handles the automatic processing of information (texts, images, sounds, videos, etc.). It generally consists of a central unit connected to other devices called peripherals.

1.4.1.1 The central unit

The central unit is the most important component of the machine. It is the box that contains the vital devices that make up a computer. These include, among others:
- The ***motherboard*** : this is the device used to group and secure all the other devices of the central unit.

- The **processor** : It represents the brain (figuratively speaking) of the central unit. It is responsible for calculations and the execution of instructions.

- **Memory** : There are two types of memory:

- **RAM** (Random Access Memory): Its role is to store programs and data during their execution. It is volatile because it loses its contents as soon as it is powered off (as soon as the computer is turned off). RAM is accessible for both reading and writing.
- **ROM** (Read Only Memory): It is dedicated to containing the manufacturer's data and programs (BIOS) essential for starting the computer and launching the operating system. It is non-volatile memory. ROM is accessible for reading only.

- The **expansion cards** (electronic circuits) installed on the machine:
  - **Graphics card** : for display.
  - **Sound card** : for sound reproduction.
  - **Modem card** : for transmitting data via telephone lines.
  - **Network card** : for transmitting data over a network.

- **Disk drives** : floppy disk drive, CD-ROM drive, memory card reader, etc…

1.4.1.2 The peripherals

Peripherals are devices connected to the central unit. They allow the computer to communicate with the outside world.
The peripherals are grouped into four categories:

Input devices: these enable data to be entered into the computer.
For example: Keyboard, mouse, etc.

Output devices: these enable information to be retrieved from a computer.
For example: Printer, screen, etc.

Input/output devices: these enable information to be entered into and retrieved from a computer.
For example: Modem, network card, etc.

Storage devices: also called auxiliary memory. These enable information to be stored.
For example: Hard drive, memory card, flash drive, etc.

1.4.1.3 Communication ports

Hardware communication ports, also called connectors, are the locations (physical interfaces) on the central unit case to which peripherals can be connected.
For example: USB port, HDMI port, Ethernet (network) port, VGA (graphics) port, mouse port, keyboard port, etc.

1.4.1.4 Minimum components of a computer

In order to ensure interaction (communication) between human and machine, the computer must consist of at least the following three components:

1. the central unit;
2. a keyboard as an input device:
   To enable human to communicate with computer (by entering data);
3. and a screen as an output device:
   To enable the computer to communicate with human (by displaying information).

1.4.2 Software

This refers to all the programs that run on a computer. There are two main categories:

1. Basic software, which constitutes the operating system.
   An operating system is necessary for any computer to start up and use.
   For example: MS-DOS, Windows, Linux, Unix, OS2, etc.
2. Application software: these are additional programs installed and used according to the user's needs.

Among the different types of software available, we can distinguish:

- Office software: This category includes word processors, spreadsheets, and other tools necessary for administrative work.
  For example: Word, Excel, WordPerfect…
- Antivirus software: These programs protect the computer against viruses, which can be devastating, and check all new data before saving it to the hard drive.
  For example: Norton Antivirus, Avira…
- Database management systems: Designed to provide users with an environment for managing their databases (personnel, inventory, invoicing, etc.).
  For example: Access, DBase…

1.5 The amount of information

Information must be coded in binary (a sequence of 0s and 1s) so that it can be processed by the computer's electronic components:

- State 1: presence of the electrical signal (on).
- State 0: absence of electrical signal (off).

This unit of information (0 or 1) is called a **bit** (binary digi**t**).
With one bit, we can represent (or encode) $2^1 = 2$ information (0 or 1)
With 2 bits, we can represent $2^2 = 4$ information (00, 01, 10 or 11).
With 8 bits, we can represent $2^8 = 256$ information.
Therefore, with n bits, we can represent 2^n information.

To measure the storage capacity of a memory or storage device, we use the Byte, which can be likened to the amount of memory needed to store one letter of the alphabet. However, current memory sizes are such that multiples of the byte are more commonly used:

- 1 Byte = 8 bits
- 1 Kilobyte (or KB) = 1024 bytes = 2^{10} Bytes.
- 1 Megabyte (or MB) = 1024 kb.
- 1 Gigabyte (or GB) = 1024 MB.
- 1 Terabyte (or TB) = 1024 GB.

Examples

5 KB= 5*1024 Byte =5120 Byte

256 KB = 256/1024 MB = 0.25 MB

125368 bits=125368/8 Bytes =15671 Bytes=15671/1024^2 MB=0.015 MB.

Chapter 2: Number Systems

2.1 Introduction

In everyday life, we have become accustomed to representing numbers using ten different symbols (digits): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. This system is called the decimal system.

However, there are other forms of number systems that operate using a number of distinct symbols.

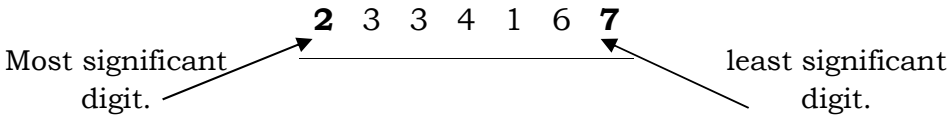
Example
- The binary system (bi: 2),
- The octal system (oct: 8),
- The hexadecimal system (hexa: 16),
- The decimal system (deci: 10).

These four systems are the most widely used in computing.
In fact, we can use any number of different symbols (not necessarily digits) in a number system. The number of distinct symbols in a number system is called the base.

2.2 Decimal System

The base of this system is 10.
In the decimal system, ten different symbols are used: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
Any combination (by juxtaposition) of these symbols forms a number.
The leftmost digit is called the most significant digit.

The rightmost digit is called the least significant digit.



2.2.1 Polynomial Expansion of a Decimal Number

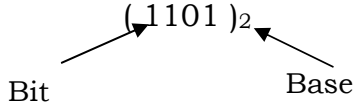
Consider the number 1978.
This number can be written in the following form:
1978 =1000 + 900 + 70 + 8
1978= 1*1000 + 9 *100 + 7 *10 + 8 *1
$1978 = 1*10^3 + 9*10^2 + 7*10^1 + 8*10^0$. ← This format is called **the polynomial form** of the number 1978.

The digit *1* is associated with the highest power (10^3), so it is the most significant digit. While the digit *8* is associated with the lowest power (10^0), so it is the least significant digit.

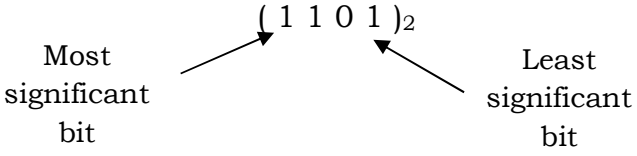
A real number can also be written in its polynomial form.
1978,265 = 1000 + 900 + 70 + 8 + 0,2 + 0,06 + 0,005
1978,265 = 1*1000 + 9 *100 + 7 *10 + 8 *1 + 2/10 + 6/100 + 5/1000
$1978{,}265 = 1*10^3 + 9*10^2 + 7*10^1 + 8*10^0 + 2*10^{-1} + 6*10^{-2} + 5*10^{-3}$

2.3 Binary System

The base of the binary system is *2*.
In the binary system, only two symbols {0, 1} are used to express any number.



In a binary number, each digit is called a "bit" (binary digit).



2.3.1 Polynomial form of a binary number

The binary number can also be expanded into their polynomial form.

Example

$(1110)_2 = 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0$

2.3.2 Binary to decimal conversion

Continuing the calculations of the number $(1110)_2$ above we will obtain:
$(1110)_2 = 8+4+2+0$
$= (14)_{10}$
The result of the polynomial expansion represents the equivalent of the number in decimal form. That is to say, the equivalent of the binary number *1110* in base 10 is the number *14* .
The same principle applies to real numbers:
$(1110{,}101)_2 = 1*2^3+1*2^2+1*2^1+0*2^0+1*2^{-1}+0*2^{-2}+1*2^{-3}$
$= (14{,}625)_{10}$

Conversion rule

To convert a binary number to the decimal system, simply use its polynomial expansion.

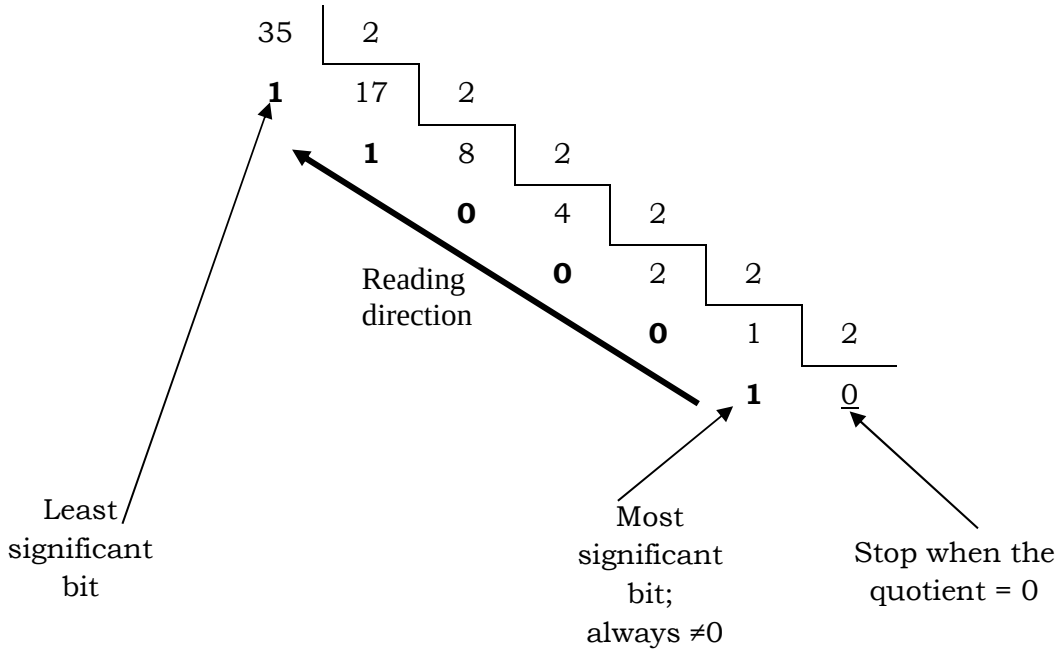
By continuing the calculations of the polynomial form of a binary number we will obtain the equivalent decimal number.

2.3.3 Decimal to binary conversion

The principle consists of successively dividing the decimal number by 2, and taking the remainders of the (Euclidean) divisions in reverse order.
We successively divide the decimal number by 2, stopping only when the result of the division (the quotient) is zero. The equivalent binary number will be the sequence of remainders obtained; where the most significant bit corresponds to the last remainder and the least significant bit corresponds to the first remainder of the divisions.

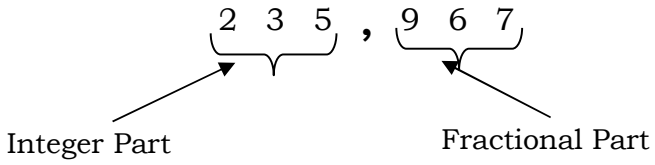
Example

$(35)_{10}=(100011)_2$



2.3.3.1 Decimal to binary conversion of a real number

A real number consists of two parts: the integer part and the fractional part.



The integer part (IP) is converted by performing successive divisions, as with an integer. The fractional part (FP) is converted by performing successive multiplications by 2. At each multiplication, the resulting integer part is taken into account (it corresponds to a

bit of the binary number). And the resulting fractional part is used for the next multiplication.

Example

Let the number be $(35,625)_{10}$
IP : $(35)_{10}=(100011)_2$
FP : $(0,625)_{10}$

0,625*2 = **1**,25
0,25*2 = **0**,5
0,5*2 = **1**,0

↓ Reading direction

We stop because the fractional part equals 0.
Therefore, $(0.625)_{10} = (0.101)_2$
And $(35,625)_{10} = (100011,101)_2$

Notes

- The precision is determined by the number of bits after the decimal point.
- We must stop the multiplications when the fractional part equals 0.
- We can also stop the multiplications when the fractional part has already been used (cyclic fractional part).
- Otherwise we stop intentionally (for irrational numbers).

2.4 Octal system

The basis of this system is *8*.
So eight symbols are used in the octal system: { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 }.

Example of the polynomial form of an octal number

$(324)_8=3*8^2+2*8^1+4*8^0$

2.4.1 Octal to decimal conversion

To convert a number from the octal system to the decimal system, simply use its polynomial expansion.

Examples

1. $(127)_8 = 1*8^2 + 2*8^1 + 7*8^0$
   $= 64 + 16 + 7$
   $= (87)_{10}$

2. $(104,65)_8 = 1*8^2 + 0*8^1 + 4*8^0 + 6*8^{-1} + 5*8^{-2}$
   $= 64 + 0 + 4 + 0,75 + 0,078125$
   $= (68,828125)_{10}$

2.4.2 Decimal to octal conversion

The same principle as above (decimal-to-binary conversion) is applied.
To convert the integer part of a decimal number to base 8, this part is successively divided by 8 until a quotient of zero is obtained. Finally, the remainders are retrieved in reverse order.
To convert the fractional part, this part is successively multiplied by 8.

Example
$(24,26)_{10} = (\ ?\ )_8$

IP : $(24)_{10}$

24	8	
0	3	8
	3	0

FP : $(0.26)_{10}$
0,26 * 8 = **2**.08
0.08 * 8 = **0**.64
0.64 * 8 = **5**.12
0.12 * 8 = **0**.96
0.96 * 8 = **7**.68
……..
We stop and we will obtain:
$(24,26)_{10} = (30,20507)_8$

2.5 Hexadecimal system

In the hexadecimal system (base 16), we use sixteen different symbols: the first 10 symbols are the symbols of the decimal system { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 } plus the letters A, B, C, D, E and F, (because the symbols must be distinct).
The decimal equivalents of the hexadecimal symbols are shown in the following table:

Decimal	Hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Tab. 2.1 : Hexadecimal to decimal correspondence

Example of the polynomial form of a hexadecimal number

$(95)_{16}=9*16^1+5*16^0$

2.5.1 Hexadecimal to decimal conversion

To convert a hexadecimal number to the decimal system, simply use its polynomial expansion.

Examples

1. $(17)_{16} = 1*16^1 + 7*16^0$
   $= 16 + 7$
   $= (23)_{10}$

2. $(ABE,2)_{16} = A*16^2 + B*16^1 + E*16^0 + 2*16^{-1}$
   $= 10*16^2 + 11*16^1 + 14*16^0 + 2*16^{-1}$
   $= 2560 + 176 + 14 + 0,125$
   $= (2750,125)_{10}$

2.5.2 Decimal to Hexadecimal Conversion

To convert a decimal number to hexadecimal, perform successive divisions for the integer part and successive multiplications for the fractional part.

Example

$(31,25)_{10}$ = (?)$_{16}$

IP : $(31)_{10}$

31	16	
15	1	16
	1	0

FP : $(0.25)_{10}$

0,25 * 16 = **4**.0

Therefore $(31,25)_{10} = (1F,4)_8$

2.6 Any-based system

Consider the following decimal number:

$1978,265 = 1*10^3 + 9*10^2 + 7*10^1 + 8*10^0 + 2*10^{-1} + 6*10^{-2} + 5*10^{-3}$

This number can also be written as follows:

$(1978,265)_B = 1*B^3 + 9*B^2 + 7*B^1 + 8*B^0 + 2*B^{-1} + 6*B^{-2} + 5*B^{-3}$

With B = *10* representing the base.

En générale, n'importe quel nombre N dans un système de numération à base B est exprimé comme suit :

In general, any number N in a base B number system is expressed as follows:

$$N = (S_k\ S_{k-1} \ldots S_1\ S_0)_B$$

With:

B : base of the number system.

S_i : symbol of the system, such that $\forall\ i \in [0..k]$, $S_i < B$

Therefore, in a number system with base B, we use B distinct symbols to represent numbers *{0, 1, …. B-2, B-1}*.

The value of each symbol must be strictly less than the base B.

2.6.1 Polynomial form of a number in a base B system

Every number in a base B system can be expanded into its polynomial form.

$$N = (S_k\ S_{k-1} \ldots S_1\ S_0)_B = S_k*B^k + S_{k-1}*B^{k-1} + \ldots + S_1*B^1 + S_0*B^0$$

Example

$(534)_6 = 5*6^2 + 3*6^1 + 4*6^0$

2.6.2 Conversion from any base system to the decimal system

In general, to convert a number from any base system (base *B*) to the decimal system (base *10*), it is sufficient to use its polynomial expansion.

Example

$(341)_6 = 3*6^2 + 4*6^1 + 1*6^0$

= 108 + 24 + 1
= $(133)_{10}$

2.6.3 Conversion from decimal to any base system

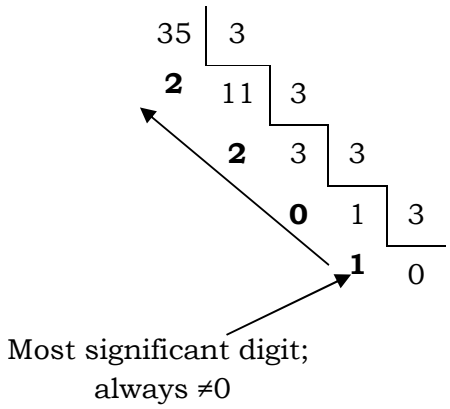
The same principle applies:
To convert the integer part of a decimal number to a base-B system:

- Successive divisions of the integer part by the base B are performed;
- The divisions are only stopped when a quotient of zero is obtained;
- Then the remainders are retrieved in reverse order.

It should be noted that the last remainder (most significant digit) must always be non-zero.
Similarly, to convert the fractional part of a decimal number to a base B system, we multiply this part successively by the base B.

Example

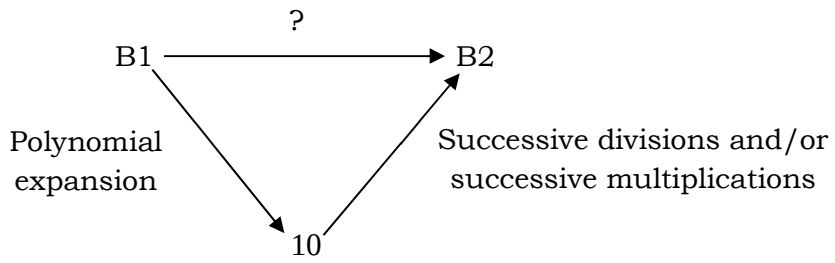
$(35,125)_{10}$ = (?)$_3$
IE : $(35)_{10}$



FP : $(0.125)_{10}$
0,125 * 3 = **0**.375
0.375 * 3 = **1**.125 ← fractional part already used
We can stop here since the fractional part **0,125** has already been used, and we will obtain :
$(35,125)_{10} = (1022,01)_3$
Otherwise, we will obtain $(35,125)_{10} = (1022,01\ 01\ 01\ .....)_3$

2.7 Converting from base B1 to base B2

There is no direct method for converting a number from base B1 to base B2.
To perform the conversion, we would need to use base 10. The idea is to convert the number from base B1 to base 10, and then convert the resulting decimal number to base B2.



Example

$(34)_5 = (\ ?\ )_7$

We begin with the polynomial expansion.

$(34)_5 = 3*5^1 + 4*5^0$

$=(19)_{10}$

Next, we divide successively *19* by *7*

19	7	
5	2	7
	2	0

$(19)_{10} = (25)_7$

Therefore, $(34)_5=(25)_7$

2.8 Octal-to-binary conversion and vice versa

We have two options for converting from base *8* to base *2* and vice versa.
The first method involves using the base *10* system seen previously.
The second possibility allows us to go directly from base *8* to base *2* and vice versa, without going through base *10*.
It should be noted that $8=2^3$: This means that each symbol of base *8* is represented by *3* symbols (bits) of base *2*.

2.8.1 Octal to binary conversion

To convert an octal number to binary, simply replace each symbol of the octal number with its corresponding 3-bit binary value (explode into 3 bits) according to the following table.

Octal	**Binary**		
	2^2	2^1	2^0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Tab.2.2 : Octal-binary correspondence.

Filling in Table 2.2 above is simple:
We start by filling in the base *8* symbols { *0 , 1 , 2 , 3 , 4 , 5 , 6 , 7* }.
Next we fill in the *3* columns (which represent the *3* bits) of base *2*.
The value 2^0 =1 of the 1st column (on the right), means that each binary symbol must be displayed alternately once (0 displayed once, 1 displayed once, 0 displayed once, 1 displayed once, etc).
The value 2^1 =2 of the 2nd column means that each binary symbol must be displayed alternately twice (0 displayed twice, 1 displayed twice, 0 displayed twice, 1 displayed twice).
The value 2^2 =4 in the 3rd column (on the left) means that each binary symbol must be displayed alternately four times (0 displayed four times, 1 displayed four times).

Examples

1. $(345)_8=(\ ?\ )_2$

We replace each octal symbol with the 3 corresponding binary symbols.

$$(\ 3 \quad 4 \quad 5\ )_8$$
$$\downarrow \quad \downarrow \quad \downarrow$$
$$(011 \quad 100 \quad 101\ )_2$$

Therefore $(345)_8 = (\ 011100101\ )_2$

2. $(65,76)_8 = (\underline{110}\ \underline{101}, \underline{111}\ \underline{110})_2$

3. $(35,34)_8 = (\underline{011}\ \underline{101}, \underline{011}\ \underline{100})_2$

2.8.2 Binary to octal conversion

The basic idea is to form groups of 3 bits.
The grouping is done from right to left for the integer part. If the last group on the left does not contain 3 bits, it will be padded with zeros.
The grouping is done from left to right for the fractional part. If the last group on the right does not contain 3 bits, it will be padded with zeros.
Then, using Table 2.2, each group (of 3 bits) is replaced by the corresponding octal symbol.

Examples :

1. $(11001010010110)_2 = (\mathbf{0}11\ 001\ 010\ 010\ 110)_2 = (31226)_8$
2. $(110010100,10101)_2 = (110\ 010\ 100\ ,\ 101\ 01\mathbf{0})_2 = (624,52)_8$

2.9 Hexadecimal to binary conversion and vice versa

For these conversions, we can go directly from hexadecimal to binary without going through decimal.
Since $16=2^4$, we can represent each hexadecimal symbol by 4 binary symbols (bits).

2.9.1 Hexadecimal to binary conversion

To convert a hexadecimal number to binary, simply replace each symbol of the hexadecimal number with its corresponding 4-bit binary value according to the following table:

Hexadécimal	**Binaire**			
	2^3	2^2	2^1	2^0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1
E	1	1	1	0
F	1	1	1	1

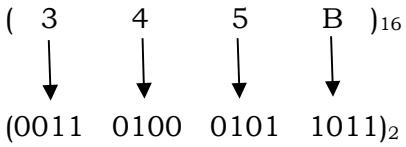
Tab.2.3 : hexadecimal to binary correspondence.

To fill the table Tab 2.3 above, we start by filling in the base 16 symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. Then we fill in the 4 columns (which represent the 4 bits) of base 2.
To fill in the first 3 columns, we apply the same principle as described previously (section 2.8.1).
To fill the 4^{th} column, the value $2^3 = 8$ means that each binary symbol must be displayed alternately eight times (0 displayed eight times, 1 displayed eight times).

Examples

1. $(345B)_{16}=(\ ?\ )_2$

We replace each hexadecimal symbol with the 4 corresponding binary symbols.



Therefore $(345B)_{16}=(0011010001011011)_2$

2. $(AB3,4F6)_{16}=(\underline{1010}\ \underline{1011}\ \underline{0011},\ \underline{0100}\ \underline{1111}\ \underline{0110})_2$

2.9.2 Binary hexadecimal conversion

The idea is to form groups of 4 bits.
The grouping proceeds from right to left for the integer part and from left to right for the fractional part. The last groups on the right and left will be padded with zeros if they do not contain 4 bits.
Using table 2.3, we replace each group (of 4 bits) with the corresponding hexadecimal symbol.

Examples

1. $(11001010100110)_2 = (\ \underline{\mathbf{00}11}\ \ \underline{0010}\ \ \underline{1010}\ \ \underline{0110}\ )_2 = (32A6)_{16}$
2. $(110010100,10101)_2 = (\ \underline{\mathbf{000}1}\ \ \underline{1001}\ \ \underline{0100}\ ,\ \underline{1010}\ \ \underline{1\mathbf{000}}\ )_2 = (194,A8)_{16}$

2.10 Direct conversion from base X to base Y and vice versa

As a general rule, we can always perform direct conversions from base X to base Y and vice versa, without going through base 10, if base X is a power of base Y.
If $X = Y^z$ then we can always replace each symbol of the base X with Z symbols of the base Y.

2.10.1 Conversion from base 2 to base 4 and vice versa

Base 4	Base 2	
	2^1	2^0
0	0	0
1	0	1
2	1	0
3	1	1

Tab.2.4 : Base 4 to binary correspondence.

$(12,03)_4 = (\underline{01}\ \underline{10}, \underline{00}\ \underline{11})_2$

2.10.2 Conversion from base 3 to base 9 and vice versa

Base 9	Base 3	
	3^1	3^0
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2
6	2	0
7	2	1
8	2	2

Tab.2.5 : Base 9 to base 3 correspondence.

$(54,82)_9 = (\underline{12}\ \ \underline{11}\ ,\ \underline{22}\ \underline{02})_3$

2.10.3 Conversion from base 4 to base 16 and vice versa

Base 16	Base 4	
	4^1	4^0
0	0	0
1	0	1
2	0	2
3	0	3
4	1	0
5	1	1
6	1	2
7	1	3
8	2	0
9	2	1
A	2	2
B	2	3
C	3	0
D	3	1
E	3	2
F	3	3

$(AE,61)_{16} = (\underline{22}\ \ \underline{32}\ ,\ \underline{12}\ \underline{01})_4$

Chapter 3: Introduction to algorithms

3.1 Introduction

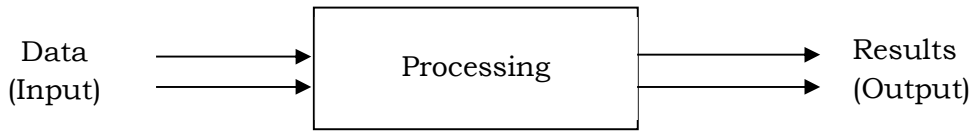
The term algorithm comes from the name of the mathematician " *Al Khawarismi ",* whose treatise on algebra describes calculation methods to follow to solve problems that often come down to solving equations.
For example, to solve the equation 8x + 3 = 4x –5;
You must follow the instructions opposite:

- subtract 4x from both members
- subtract 3 from both members
- divide both members by 4
- display the solution.

3.2 The algorithm

Writing an algorithm is essential for solving a problem. It allows you to describe the steps to follow to achieve the desired results from a set of data.

An algorithm is a sequence of elementary instructions that are applied in a specific order to data (input) and that provide results (output) in a finite number of steps.



An algorithm consists of these three operational steps:
1- Input of the necessary data (inputs).
2- Execution of elementary and ordered instructions on this data (processing).
3- Output of the results obtained (outputs).

Exemple

To make coffee, you need to:
1. Heat water
2. Add the coffee
3. Stir
4. Serve

These four steps are not detailed enough. It would be better to describe them more thoroughly, breaking down each instruction into a more understandable (elementary) sequence of instructions:

1. Heat water.
    1.1 Take a saucepan.
    1.2 Put water in the saucepan.
        1.2.1 Put the saucepan under the tap.
        1.2.2 Turn on the tap.
        1.2.3 Fill the saucepan.
        1.2.4 Turn off the tap.
    1.3 Place the saucepan on the stove burner.
    1.4 Turn on the stove burner.
    1.5 Wait for it to boil.
    1.6 Turn off the stove burner.
2. Add the coffee
3. Stir
4. Serve

Division of instruction into a series of elementary instructions.

3.3 Languages

The description of the instructions, as well as the objects manipulated by these instructions, is done via a language.

The language used to formulate instructions must be understandable by the executor. For a human, it is enough to use everyday language (their native language), but for a machine, it is necessary to use a language that it understands.

3.3.1 The programming language

A programming language is a high-level language consisting of a set of commands and keywords necessary for writing instructions that can be understood by the machine. Examples: Pascal, C, Java, Basic, Fortran, etc.

3.3.2 The program

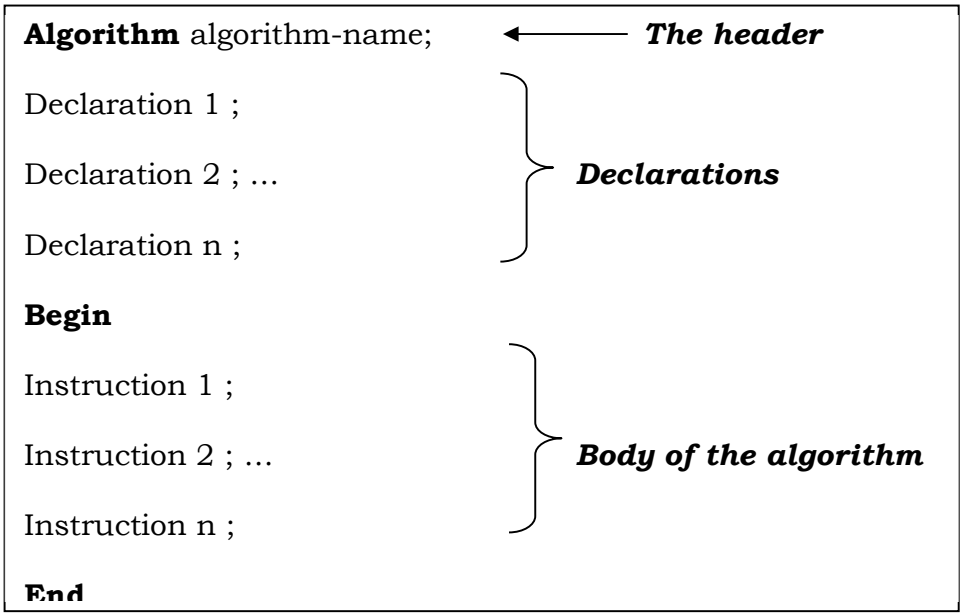
A program is the translation of an algorithm into a programming language (that is, the translation into a language understandable by the machine).

3.3.3 Algorithmic Language

Algorithmic language is a pseudo-language that takes into account the characteristics of the machine. It remains more flexible than a programming language. Indeed, algorithmic language is a mixture (arrangement) between everyday language (native language) and a programming language (the Pascal programming language is considered the most appropriate).

3.4 The structure of an algorithm

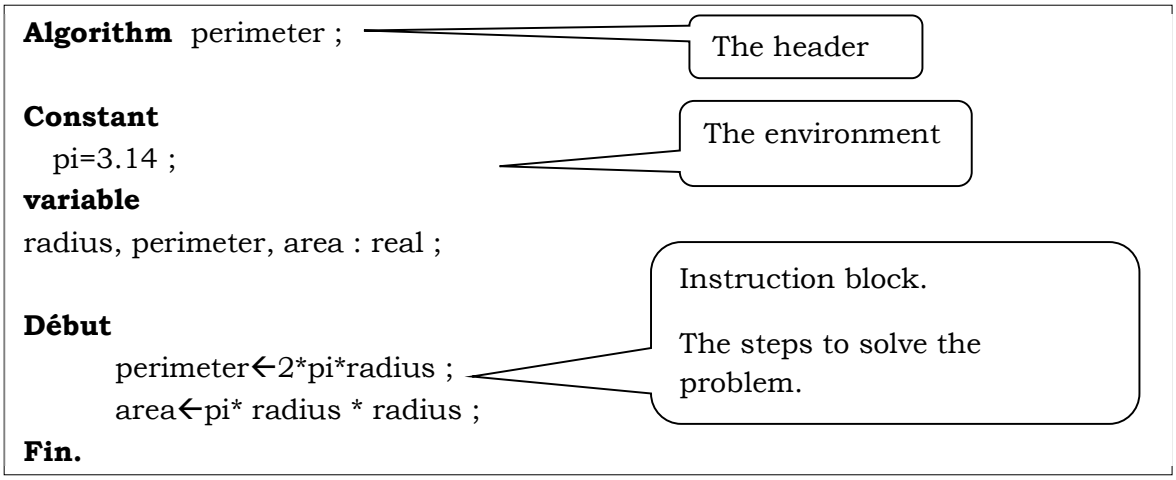
An algorithm is composed of three parts: the header, the declarations (the environment) and the body of the algorithm (block of instructions).



- The header: is used to identify the task to be solved (identification of the algorithm). The header is defined by the keyword **algorithm** followed by a name of the task under consideration.
- The declarative part: is a list of all the objects used in the body of the algorithm (the problem-solving environment).
  Programming languages use declarative instructions to define objects in the environment.
- The body of the algorithm: describes the method for solving the problem (block of instructions).
  The body of the algorithm is delimited by the words **Begin** and **End** followed by a ".".

Example

The calculation of the perimeter and area of a circle in algorithmic language is defined as follows:



3.5 The declaration of an object

When an object is declared in its environment, it is defined by some of its characteristics according to its nature.
Depending on its nature, an object is either a constant or a variable.
A variable is characterized by its name and type. Its value, which is subject to change, does not characterize it.
Whereas a constant is characterized by a name and its value which cannot be modified under any circumstances.

3.5.1 The name of an object

An object's name (identifier) must follow a specific syntax:

- It consists of a string of characters containing only:
  - alphabetical letters: ' a' ..' z ' and 'A' .. 'Z'.
  - numbers: 0.. 9.
  - and the underscore character: " _ ".
- It must begin with a letter.
- It does not contain subscripts or superscripts.
- It must be different from all the keywords.

Examples

- *radius, absolute_value , x1* : are correct names.
- *1x* : is not a correct name because it does not begin with a letter.
- *Squàre root* : is not a correct name because it contains the space and the "à".
- x_1 : is not a correct name because it consists of a subscript.
- *Begin* : is not a correct name because it is a keyword.

For readability reasons, it would be preferable to give objects meaningful names.

3.5.2 Variable declarations

In an algorithm, a variable is declared by defining its name and its type (numeric, alphanumeric, ...).
This step is called variable declaration.
To declare a variable in an algorithmic language, the keyword **variable** is used according to the following syntax:

Variable
Variable_name: data_type;

with :
Variable_name : is the name (identifier) assigned to the variable ;
data_type : is the type of information to which this variable belongs (type of this variable).

Example

variable
radius : real ;

To declare a variable in a Pascal language, the keyword **var** is used according to the following syntax:

Var
Variable_name: data_type;

3.5.3 Declaration of a constant

A constant is declared by defining its name and value.
To declare a constant in an algorithm, we use the keyword **constant.** according to the following syntax:

Constant
Constant_name = value;

with :
Constant_name **:** is the name (identifier) assigned to the constant;
value **:** is the value of this constant.

Example

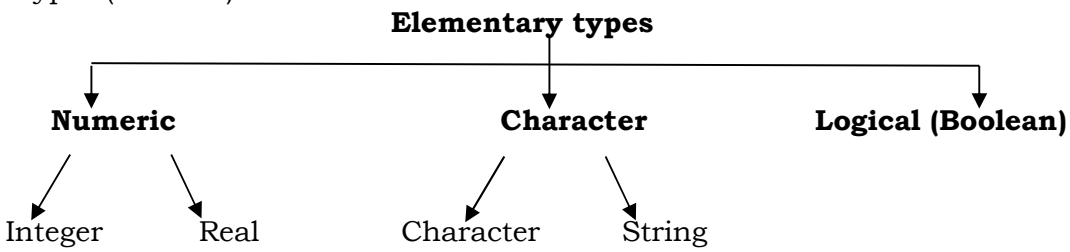
Constant
pi = 3.14;

To declare a constant in a Pascal language, we use the keyword **const** according to the following syntax:

Const
Constant_name = value;

3.6 Elementary data types

A data type is elementary if it defines a simple set of values.
The predefined types available in programming languages are:
- Numeric types (integers and real numbers),
- Character types (characters, strings),
- Logical types (Boolean).



3.6.1 The integer type

It defines a limited value domain, unlike the set of integers in mathematics, which is infinite. The value domain of the integer type is a closed bounded interval ([- 32 768 .. 32 767] occupying 2 bytes).
The operators associated with the integer type are (common operators):

- The addition (+).
- Subtraction (-)
- Multiplication (*)
- The Euclidean (integer) division denoted by **div .**
- The modulo operator (remainder of euclidean division) denoted by **mod** .

Examples

27 div 5 = 5; 27 mod 5= 2; 9 mod 3 = 0.
27 div 5 = 5 ; 27 mod 5= 2 ; 9 mod 3 = 0.

3.6.2 The real type

The real type takes its values from a subset of the real numbers (in mathematics).
It is equipped with the following operators (standard operators):

- The addition (+).
- Subtraction (-)
- Multiplication (*)
- The actual division (/)

As well as other predefined mathematical functions:

- Sqrt ( ): is a function that provides the square root of a number.
- Abs( ): is a function that provides the absolute value of a number.

Examples

Sqrt(16) =4 ; Abs(-8) =8 .

3.6.3 The character type

- The character type is the set of all printable characters (keyboard keys). It includes:
  - the letters ' a' ..' z ' and 'A' .. 'Z'.,
  - the numbers 0.. 9 ,
  - punctuation marks (., ; , ?, !, ...),
  - the symbols used as operators (+, *, -, /, <, =, ...),
  - special characters (%, @, $, &, ...)
- A character must be enclosed in two apostrophes.
- The set of characters is ordered.
- Among the predefined functions for manipulating characters are:
  - Succ ( ): a function that provides the immediate successor of a character.
  - Pred ( ): a function that provides the immediate predecessor of a character.

Examples

Succ('f')= 'g' ; Pred('f')='e' .

3.6.4 The string type

- A string type is a sequence of characters.
- A string is enclosed by two apostrophes.
- Any apostrophe appearing in a string must be doubled.
- Several predefined functions are dedicated to manipulating strings, such as:
  - Length ( ): a function that provides the length of a string.
    The length of a string can be 0 (an empty string).
  - Concat ( ): a function that allows you to concatenate two strings.

Examples

Length ('computer')=8 ; Concat('computer', 'science') = 'computer science'.

3.6.5 The logical type (Boolean)

The two values of this type are **true** and **false .**
The most commonly used logical operators are:

- The conjunction (AND)
- Disjunction (OR)
- The negation (NOT)

The AND, OR, and NOT operators are defined by the truth table below:

A	B	A AND B	A OR B	NOT A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Tab.3.1 : Truth Table

Note

The result obtained from a comparison (using the relational operators: <, ≤, =, ≠, >, ≥) is a boolean value.

3.7 The elementary instructions

An elementary instruction is an instruction expressed by a simple command.
Every instruction is defined by:

- A syntax that specifies the writing formalism adopted to formulate the command.
- A semantics that defines the actions performed by the machine in response to this command.

We will frequently use expressions to formulate instructions.
An **expression** is a sequence of operations performed on operands (variables, constants, values, etc.). Each expression has a value and a type.

3.7.1 The assignment

A variable can be likened to a box, identified by its name and can contain information.
To use the contents of this box, simply call it by its name.
Assignment allows you to store the value of an expression in a variable.

Fig.2.1 : Assigning the number 6 to the variable a

Syntax

In algorithmic language, the assignment instruction is given by the following syntax:

| *X←E ;*

With :

X: the variable name.

←: the symbol of assignment.

E: an expression.

We say that the variable *X* receives the value of the expression *E.*

In Pascal, the assignment instruction is symbolized by " **:=**" according to the following syntax:

| *X :=E ;*

Semantics

The assignment is carried out in two steps:

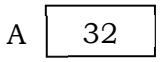
1- The expression *E* is evaluated, let *v be* the value obtained.

2- The value *v* is stored in the variable *X.*

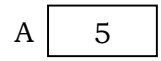
The assignment is valid only if the type of the expression *E* is compatible with the type of the variable *X.*

Examples

1. Let *A be* an integer variable whose state at a given time is represented by:

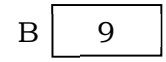


The execution of operation A ←5 has the following effect:

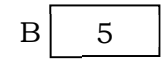


The value 5 overwrites the value 32, which is then permanently lost.

2. Let *B be* an integer variable,



The execution of operation B ←A has the following effect:



We note that this assignment has no effect on the variable *A* (the value of *A* is still 5).

3. Let *X be* a variable of real type, the instruction X ←A is valid.
It respects the inclusion of the set of integers in the set of real numbers.
The value 5 is converted to a real number before being assigned to the variable *X.*
However, the assignment A ←X causes a type mismatch error.

4. The execution of the operation B ←(B-2)*A has the following effect:

B	15

The initial value of variable *B* (5) is used in the calculation of the expression. The resulting value is then assigned to this variable.

3.7.1.1 Incrementing and decrementing

Let variable A be of type integer.
The instruction A ←A+1 is called an increment operation.
We say that we increment *A.*
The inverse operation of incrementing is decrementing (A ←A-1).
We say that we decrement *A.*

3.7.2 Reading instruction

The read instruction allows the user to transmit data to the machine (to the process).

Syntax

In algorithmic language, the read instruction is given by the following syntax:

```
Input(X);
```

with :
X: variable name.

Several read instructions can be grouped into a single one:
The instruction Input(A, B, C) is equivalent to Input(A); Input(B); Input(C).

In Pascal, the read instruction is given by the following syntax:

```
read(X);
```

Semantics

The value transmitted by the user, via an input device (keyboard), is assigned to the variable X.
Each time the machine (the process) receives a n input *(X)* command , program execution is suspended while awaiting input of a value. The user must then enter a value via the keyboard. Once the input is validated (by pressing the Enter key ↵), program execution resumes and the value entered by the user is assigned to the variable *X.*
If the entered value is incompatible with the variable type, reading it will cause an error.

3.7.3 Writing instruction

The write instruction allows the process (machine) to communicate information (messages or processing results) to the user (display on screen or printer, etc.).

Syntaxe

In algorithmic language, the writing instruction is given by the following syntax:

Display(E) ;

with :

E : Expression.

Several writing instructions can be grouped into a single one:
The instruction display(A, B, C) is equivalent to display(A); display(B); display(C).

In Pascal, the writing instruction is given by the following syntax:

Write(E) ;

Semantics

The execution of display*(E) instruction* takes place in 2 steps:

1- The expression *E* is evaluated.
2- Its value is communicated to the user via an output device (screen or printer).

Examples

Display *(2*x-3)* has the effect of displaying the value of the expression 2*x-3;
If for example *x* equals 10, this instruction has the effect of displaying the value **17** .
display ('Enter an integer greater than 10') displays the string constant:

Enter an integer greater than 10 on the screen.

Note

Displaying messages is very useful in programming.
It guides the user, following an input command, by indicating what the machine expects of them. In addition, it explains the results obtained from the processing.

3.8 Comments

To make an algorithm (or a program) more readable and understandable, it must be commented.
Comments are short texts inserted into an algorithm to explain certain parts of its body and they have no effect on the execution of the algorithm.

By convention, comments are written as follows:
1. One-line comment: which begin with //

Example // This is a single-line comment

2. Multi-line comment: which is enclosed by { and }, by **(*** and ***)** or by **/*** and ***/.**

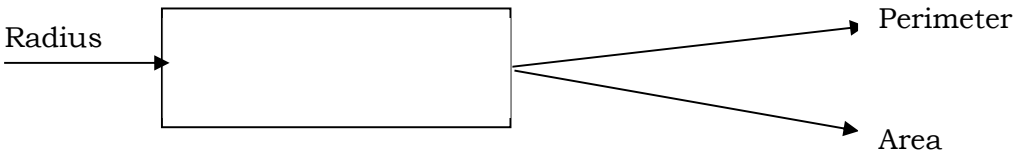
Example {This is a multi-line comment}

3.9 Application exercise

Write the algorithm and the equivalent program in Pascal language that allow you to calculate and display the perimeter and area of a circle whose radius is entered by the user.

Solution :

The solution to this exercise requires a single input value (data), which is the radius, and two output values (results), which are the perimeter and area of a circle.




```
Algorithm perimeter_area_circle;
Constant
pi=3.14 ;
variable
r, p, a :real ;
Begin
display ('enter the value of the  radius of the circle) ;
input (r) ;
p←2*pi*r ;
a←pi*r*r;
display ("the perimeter of the circle is:", p);
display ("the area of the circle is:", a);
End.
```

Declaration of pi (π), r (radius), p (perimeter) and a (area)

The machine asks the user to enter the radius value.

The machine reads the value of the radius.

Calculation of perimeter and area

Display of perimeter and area values

The following program is the translation of the previous algorithm into Pascal language.

```
Program  perimetre_area_cercle ;
Const
pi=3.14 ;
Var
r, p, a :real ;   { r: radius, p : perimeter, a : area }
Begin
Write ('enter the radius of the circle') ;
Read (r) ;
p :=2*pi*r ;
a :=pi*r*r;
Write('the perimeter of the circle is:', p) ;
Write('the area of the circle is:', a) ;
End.
```

This is a comment.

3.9.1 The execution of the program

The execution of an algorithm (program) is performed instruction by instruction (from the **begin** keyword to the **end** keyword).
The execution sequence illustrates the effect of executing an algorithm (program) on the state of the machine, and more specifically on the states of the screen and memory.
The screen state can only be modified by executing read and write instructions. Modifying the value of a variable modifies the memory state.

Example

The execution of the preceding program is represented by the following states:

4- Reading changes the value of the variable r

1- Initially, the values of the variables are unknown.

pi	r	p	a
3,14	?	?	?
	3	18,84	28,26

5- The assignments modify the values of the variables p and a

Memory

```
Give the value of the radius of the circle.
3.
The perimeter of the circle is: 18.84
The area of the circle is: 28.26
```

2-The machine displays this message

3- The user enters the value 3 (for the radius)

6- The machine displays the results (perimeter, area)

Screen

1- Initially, the values of the variables are unknown. Only the value of the constant pi is known. It is initialized to 3.14.

2- The machine displays the message "Enter the value of the circle's radius."

3- The user enters the value 3 (for the radius).

4- Reading the value modifies the value of the variable r: The machine reads the value 3 and stores it in the variable r.

5- Assignments modify the values of variables p and s: The machine calculates the values of s and p.

6- The machine displays the results of p and a (perimeter, area).

Chapter 4: Conditional control structures

4.1 Introduction

So far we have seen algorithms with instructions that follow each other sequentially (sequential structures); that is to say, the instructions will be executed, from beginning to end, one after the other.
However, this is not always satisfactory. Sometimes, it is necessary to decide whether to execute an instruction or a block of instructions (should a block of instructions be executed or not?). For this, conditional control structures are used, which offer the possibility of selecting, during execution, the block of instructions to be executed based on a condition.

There are 3 conditional control structures:
- Simple conditional structure
- Alternative conditional structure
- Conditional structure with multiple choices.

The first two structures will be presented in this chapter, the third structure will be presented in the next chapter.

4.2 The simple conditional structure

The simple conditional structure allows us, based on a condition, to choose between executing or ignoring a block of instructions.
Sometimes, it is necessary to execute one or more instructions only if a condition is true and to do nothing if the condition is false.

Syntax

In algorithm, the syntax of the simple structure is as follows:

If condition **then**
<Instruction block> ;
endif;

Note that an instruction block may contain one or more instructions.

Semantics

The structure is executed in two steps:

1. Evaluate the condition.
2. If it is verified (true), the instruction block is executed.
   If it is not verified (false), the instruction block is ignored (not executed).

In Pascal, one of these two syntaxes is used:

1st syntax

```
if   condition then
     Instruction1;
```

2nd syntax

```
if   condition  then
begin
  Instruction1 ;
  Instruction2 ;
end;
```

The choice of syntax depends on the number of instructions per block (one or more):

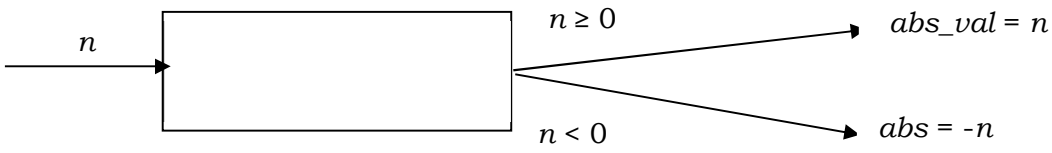
1. If the block contains only one instruction, then the first syntax is used;
2. If the block contains several instructions (2 or more instructions) then we use the second syntax.

If several instructions (two or more) depend on the condition, then you must use **begin** and **end** (delimiting the instructions with **begin** and **end**). However, if only one instruction depends on the condition, then you do not use **begin** and **end**.

Example

Write an algorithm that displays the absolute value of an integer.

Let *n be* the number to be entered by the user (via keyboard).
And abs_val the absolute value to display as the result.



The algorithm in this example is as follows: :

```
Algorithm absolute_value;
Variable
n, abs_val: integer;
begin
display (“enter an integer”) ;
input (n) ;
Abs_val ←n ;
If nb<0 then
   Abs_val ←- n ;
endif
display (“the absolute value of the number is:”, abs_val) ;
End.
```

In this algorithm, we assume that the number (*n*) is positive and we only calculate its absolute value if it is negative. Therefore, no processing is required if the condition is false (*nb* > 0).

The translation of this algorithm into Pascal language is as follows:

```
program absolute_value;
Var
n, abs_val : integer ;
begin
write (‘enter an integer ‘) ;
read (n) ;
Abs_val :=n ;
if nb<0 then
   Abs_val := -n ;
write (‘the absolute value of the number is:”, abs_val) ;
end.
```

4.3 The alternative conditional structure

The alternative conditional structure allows us, based on a condition, to select the block of instructions to execute from among two blocks.

Syntax

In algorithms, the syntax of the alternative structure is as follows:

If condition **then**
<Instruction block 1>
Otherwise
<Instruction block 2>;
endif;

Semantics

The structure is executed in two steps::

1. Evaluate the condition.
2. If it is verified (true), only *Instruction Block 1* will be executed;
   And if it is not verified (false), only *Instruction Block 2* will be executed.

In Pascal, the syntax is as follows:

if *condition* **then**
<*Bloc d'instruction1*>
else
<*Bloc d'instruction2*> ;

Reminder

You should always use **begin** and **end** to delimit the block of instructions if it contains several instructions (2 or more).

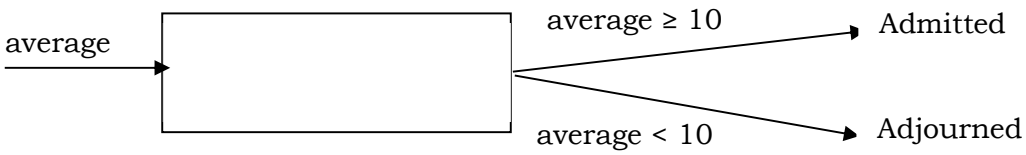
Note

In Pascal, the keyword **else** must never be preceded by a semicolon (**;**).

Example 1

Write an algorithm to read a student's average and display the word "Admitted" if the average is greater than or equal to 10 and the word " Adjourned " otherwise.
Translate this algorithm into Pascal language.

To display one of these two options, the value of the average entered by the user (via a keyboard) must be tested. This test is performed using the alternative structure.
The problem can be expressed as follows:
If the average is greater than or equal to 10 **then** display "Admitted" **otherwise** display "Adjourned" **.**



The algorithm for this example is as follows:

```
Algorithm mention;
Variable
Average: real;
Begin
display ('enter the student''s average');
input (average);
If (average ≥ 10) then
  display ('Admitted')
otherwise
  display ('Adjourned');
Endif
End.
```

The translation into Pascal language is as follows:

```
program   mention ;
 Var
average: real ;
Begin
write ('enter the student''s average');
read (average) ;
if  (average >=10)    then
      write('Admitted')
else
      write ('Adjourned') ;
End.
```

Example 2

Write an algorithm that reads a number and determines whether that number is a multiple of 3.

The algorithm is as follows:

```
Algorithm multiple ;
Variable
n : integer;
Begin
Display ("enter an integer");
input (n) ;
If   (n mod 3 =0)    then
   Display (n, 'is a multiple of 3)
otherwise
   Display (n ,' is not a multiple of 3") ;
endif
end.
```

The translation into Pascal language is as follows:

```
program multiple ;
Var
n : integer ;
begin
write (“enter an integer”);
read (n) ;
if  (n mod 3 =0)    then
      write (n, ‘is a multiple of 3)
else
   write (n ,‘ is not a multiple of 3”) ;
end.
```

The variable *n* which represents the number to be read must be of integer type, since the **mod** (modulo) function only applies to integers.

Reminder

The `mod` function returns the remainder of the integer division. Thus, *n mod 3* expresses the remainder of the integer division of *n* by 3.

4.4 Nested Alternative Structures

In some processes, it is necessary to select one block of instructions to execute from among several (more than two blocks).
When the number of possible cases is greater than two, it would be necessary to use nested alternative structures (one structure inside another).

Note

(The number of " **if** ... **then** ... **otherwise**" structures) = (number of cases) - 1.

Example

Write an algorithm that reads a student's average and displays one of the following messages:

- “Good” if the average is above 13.
- “ Medium” if the average is between 10 and 13.
- “ Weak” if the average is below 10.

In this example, the number of possible cases is 3. A simple alternative structure cannot solve the problem. Therefore, we use the structure of nested alternatives.
(number of " **if** ... **then** ... **otherwise** " structures) = (number of cases) -1 = 3 -1 = **2** .

The algorithm is as follows:

```
Algorithm mention;
Variable
Average: real;
begin
display (“give the student''s average”);
input (average);
If (average > 13) then
  display (“Good”)
otherwise
     If (average ≥ 10) then
        display (“Medium”)
     otherwise
        display (“Weak”);
    endif
endif
End.
```

The translation into Pascal is as follows:

```
program  mention ;
Var
average: real ;
begin
write (‘give the student''s average’);
read (average) ;
if  (average > 13) then
      write (‘Good’)
else
      if  (average > =10) then
            write (‘Medium’)
      else
            write (‘Weak’) ;
end.
```

4.5 Application exercise

Write the algorithm and the Pascal programme that enable a second-degree equation to be solved.

Solution

A second-degree equation is as follows : $ax^2 + bx + c = 0$.
With x is the unknown and a, b, and c are the coefficients, with a not equal to 0.
The input data are the coefficients a, b, and c.
The output results are the different values of the unknown x, depending on the value of the discriminant (Δ).

$$x = \begin{cases} No\ solution, & \Delta < 0 \\ Double\ soluiton : \dfrac{-b}{2 * a}, & \Delta = 0 \\ \dfrac{-b \pm \sqrt{\Delta}}{2 * a}, & \Delta > 0 \end{cases}$$

We will use the nested alternative structure with two '**if...then... otherwise**' statements, since there are three cases (depending on the value of Δ).
Plus another test on the value of the coefficient a to ensure that it is different from zero, so as not to have a first-degree equation.

The algorithm is as follows::

```
algorithm second_degree_eq;
Variable
a,b,c,delta,x1,x2 : real ;
begin
  display ('enter the first coefficient a=') ;
  input (a) ;
  if a=0 then
    display ('enter a value for a.≠0')
  otherwise
    display ('enter the second coefficient b=') ;
    input (b) ;
    display ('enter the third coefficient c=') ;
    input (c) ;
    delta←b*b-4*a*c;
    if delta <0 then
     display ('No solution')
    otherwise
      if delta=0 then
        x1←-b/(2*a);
        display ('Double solution, x=', x1) ;
      otherwise
        x1← (-b-sqrt(delta))/(2*a);
        display ('x1=', x1) ;
        x2← (-b+sqrt(delta))/(2*a);
        display ('x2=', x2) ;
      endif;
    endif;
  endif;
End.
```

The program in Pascal is as follows::

```
program second_degree_eq;
Var
a,b,c,delta,x1,x2 : real ;
begin
  write ('enter the first coefficient a=') ;
  read (a) ;
  if a=0 then
    write ('enter a value for a.≠0')
  else
  begin
    write ('enter the second coefficient b=') ;
    read (b) ;
    write ('enter the third coefficient c=') ;
    read (c) ;
    delta:=b*b-4*a*c;
    if delta <0 then
     write ('No solution')
    else
      if delta=0 then
      begin
        x1:=-b/(2*a);
        write ('Double solution, x=', x1) ;
      end
      else
      begin
        x1:=(-b-sqrt(delta))/(2*a);
        write ('x1=', x1) ;
        x2:=(-b+sqrt(delta))/(2*a);
        write ('x2=', x2) ;
      end;
  end;
End.
```

Chapter 5 : Multiple-choice structure

5.1 Introduction

It is useful to be able to choose one of several treatments based on the value of a variable. If there are only two possible actions, the alternative structure (if...then... otherwise) will be the most appropriate and is well suited to solving this type of problem. Otherwise, a direct branch to the desired treatment can often be achieved using the case statement.

5.2 Multiple choice structure (depending on case)

The multiple choice structure labelled 'depending on the case' is an instruction that allows a branch to be made based on the value of an expression. In some cases, it can replace nested alternative structures. It is used when there are multiple choices to manage.
The multiple-choice structure is equivalent to a series of nested **if...then... otherwise** statements. It should be preferred, whenever possible, to a nested **if...then... otherwise** structure, which is not actually optimally readable.
The multiple-choice structure can only be used in cases where the logical expressions (conditions) are in the form of equalities between discrete values (integer, character).
Unlike nested **if...then... otherwise** statements, with **depending on case**, the branching condition must relate to the comparison of an expression with constant values, and not to any condition (branching based on conditions such as delta>0, delta<0 or x=0.01 must be performed using nested **if...then... otherwise** statements).
For example, conditional control structures containing logical expressions (conditions) such as a>0, a=0.1, b/2=1, must only be implemented using nested alternative structures (it is not possible to use multiple-choice structures).

The structure **depending on case** is not suitable for the following conditions:
• a>0: because it is not an equality comparison.
• a=0.1: because the value 0.1 is of type real (which is not discrete).
• b/2=1: because the expression b/2 is of type real (which is not discrete).
Therefore, the case structure can be used for equality comparisons between integers or between characters... (*a=0, a='x'* ...).

Syntax

```
Depending on case express
   case_1 : <Instruction block_1 >;
   case_2 : <Instruction block_2 >;
…
   case_n : < Instruction block_n >;
[Otherwise < Instruction block _n+1> ; ]
End;
```

The otherwise section is optional.

With

Express : an expression whose value is of discrete type.

case_1, case_2, ... case_n : lists of constants of the same type as the E*xpress* expression.

Each list consists of a sequence of constants, separated by commas.

Note

Anything between square brackets " **[ ]** " is optional. It may not be used (depending on the needs).

Semantics

After evaluating the expression, the program branches to the list containing the value of that expression, and the corresponding instruction block is executed.

If the value of that expression does not appear in any of these lists, the program branches to the **otherwise** clause and *instruction block_n+1* is executed.

The presence of the **otherwise** clause in the instruction is optional.

In Pascal, the syntax is as follows:

```
Case   express  of
   case _1 : <Instructions block_1 >;
   case _2 : <Instruction block_2 >;
…
   case _n : < Instruction block_n >
[; else <Instruction block_n+1 > ] ;
end;
```

5.3 Application exercises

Exercise 1

We want to know the discount applicable to a railway user. The discount is determined according to the user category:
If category=1, then discount=10%
If category=2, then discount=50%
If category=3, then discount=100%
If category=4 then discount =30%
For any other category value, discount=0%.

1. Using alternative instructions, write an algorithm that allows you to:
   - Read the ticket price and the user's category.
   - Calculate and display the discount amount.
2. Rewrite the same algorithm using the multiple choice structure.

Solution1 :

In the following algorithm, nested **'if...then...otherwise'** structures are used.
There are five possible cases according to the 'cat' category, so there will be four **if...then...otherwise.**

```
Algorithm reduction;
Variable
mtb, red: real;
cat: integer;
begin
Display ('Enter the ticket amount:');
Input (mtb);
Display ('Enter the user category:');
Input (cat);
If (cat= 1) then
   Red ← mtb*0.1
Otherwise
   If (cat=2) then
      Red ← mtb*0.5
   Otherwise
      If (cat=3) then
         Red ← mtb
      Otherwise
         If (cat=4) then
            Red ← mtb*0.3
         Otherwise
            Red←0
         Endif
      Endif
   Endif
Endif
Display (' The amount of the discount=', red);
End.
```

Solution2 :

In the following algorithm, we use the depending case structure.
The following solution illustrates the case where the lists (case1, case2, case3, case4) are simple and contain only one constant (1, 2, 3, 4).

```
Algorithm reduction;
Variable
mtb, red: real;
cat: integer;
begin
Display ('Enter the ticket amount:');
Input (mtb);
Display ('Enter the user category:');
Input (cat);
  Depending on case cat
    1 :  Red ← mtb*0.1 ;
    2 :  Red ←  mtb*0.5 ;
    3 :  Red ← mtb;
    4 :  Red ← mtb*0.3
  otherwise
   Red ← 0 ;
  end ;
Display (' The amount of the discount=', red);
End.
```

The Pascal language translation of this algorithm is as follows:

```
program   reduction ;
Var
mtb, red : real ;
cat : integer ;
begin
 write ('Enter the ticket amount:') ;
 read (mtb) ;
 write ('Enter the user category :') ;
 read (cat) ;
  Case cat of
    1 :  Red := mtb*0.1 ;
    2 :  Red :=  mtb*0.5 ;
    3 :  Red := mtb;
    4 :  Red := mtb*0.3
  else
     Red := 0 ;
  end ;
  write ('The amount of the discount=', red) ;
end.
```

Exercise 2

Write the algorithm and program that read a character and display which set it belongs to:

1. To 'Letters' if the character belongs to 'A'..'Z', 'a'..'z'
2. To 'Numbers' if the character belongs to '0'..'9'.
3. To 'Operators' if the character belongs to '+', '-', '*', '/'
4. To 'Comparators' if the character belongs to '<','>','='
5. Otherwise belongs to 'Special characters'.

The following algorithm illustrates the case where lists are composed: each list consists of a sequence of constants, separated by commas.

```
algorithm character_set;
Variable
car : character;
Begin
Display('Enter a caractere');
input (car);
Depending on case car
  'A'..'Z', 'a'..'z' :  Display('Letters');
  '0'..'9' :            Display('Numbers');
  '+', '-', '*', '/' :  Display('Operators');
  '<','>','=' :         Display('Comparators')
otherwise
  Display(' Special characters");
end;
End.
```

The translation of the algorithm into Pascal language is as follows:

```
Program character_set;
Var
car :char;
Begin
Write('Enter a caractere');
read (car);
case car of
  'A'..'Z', 'a'..'z' :  Write('Letters');
  '0'..'9' :            Write('Numbers');
  '+', '-', '*', '/' :  Write('Operators');
  '<','>','=' :         Write('Comparators')
else
  Write(' Special characters");
end;
End.
```

Chapter 6 : Repetitive control structures (loops)

6.1 Introduction

In everyday problems, we do not only execute processes (blocks of instructions) with or without conditions, but it can often be necessary to execute a process several times.
In fact, to enter a student's N marks and calculate their average, you have to enter N variables, then add them up and divide the sum by N. This solution requires reserving space by declaring variables and a series of read and write instructions.
This problem can be solved by using repetitive control structures, also known as iterative control structures or simply **loops**. These allow you to give an order to repeat an instruction or block of instructions zero or more times.

6.2 Loops (Iterations)

Iterations (loops) allow the same process described by an instruction or block of instructions to be repeated several times.
In literature, the block of instructions to be repeated is also called the body of the loop.
In a loop, the number of iterations must be finite and must be controlled by a counter or a condition.

6.3 The 'For' loop

The **For** loop uses a counter to control the number of iterations of the process.
Therefore, this number of iterations must be known (set in advance).

Syntax

The syntax in algorithmic language is as follows:

> **For** Counter **from** Initial_value **to** Final_value [**Step** Step_value] **Do**
> <Instructions block > ;
> **Enddo**

With :

Counter: is an integer variable, which counts the number of times the execution of the < Instructions block > is repeated;
Initial_value : the initial value to which the *counter* is initialized,
Final_value : the final value at which the *counter* terminates ,
Step_value : is the value that is added to the *counter* at each iteration.

Semantics

The **For loop** works as follows:

1. Initialise the *counter* with *Initial_Value* (as if we had *Counter* = *Initial_Value*)
2. Test the *counter*

- if *Counter* value exceeds the f*inal_Value* (on the upper or lower side, depending on the positivity or negativity of the step); Then the loop stops and execution continues after the **enddo.**
  Otherwise, execute the <Instruction Block>,
- Incrementing or decrementing the *counter* by the step value (depending on whether the step is positive or negative),
- Return to step 2.

Note

- **For** loop is used when the number of iterations is known in advance.
- The step value can be positive or negative, and therefore, at the start of the loop, the *initial_value* must be less than or equal to the *final_value* or the *initial_value* must *be* greater than or equal to the *final_value,* depending on whether this value is positive or negative.
- **For** loop allows the process to be repeated **zero** or more times.
  In fact, if the step value is positive and the *Initial_value* is greater than the *final_value* then no iteration will be executed.
- The step value is optional. It is equal to 1 by default.

In Pascal, the syntax of the **For** loop will be as follows:

For *counter* **:=** *initial_value* **to** *final_value* **do**
<Instruction block> ;

Reminder

The instruction block must be placed between **begin** and **end** if it contains several instructions (2 or more).

Notes

- In Pascal, it is not possible to use the **For** loop with a step other than 1 or -1.
- In Pascal, if the step value is -1, then the syntax is as follows:

For *counter* **:=** *initial_value* **downto** *final_value* **do**
<Instruction block> ;

Example 1

Write an algorithm that calculates and displays the sum of n numbers.
In this algorithm, n numbers must be read and their sum calculated. The reading operation must be repeated n times. After each reading, the algorithm must add the value of the number read to the sum of the numbers read previously.
Therefore, the process of reading the number and adding it to the sum must be repeated n times.

```
Algorithm sum ;
Var
i, n :integer ;
s, x : real ;
begin
Display ('Give the number of numbers');
input (n) ;
s←0 ;
for  i from 1 to n step 1 do
  Display ('Give a number');
  input(x);
  s←s+x ;
enddo
Display ('The sum =', s);
end.
```

The step is optional.
Since step=1. We could simply write
For i from 1 to n do

The translation into Pascal will be as follows:

```
program sum ;
Var
i, n :integer ;
s, x : real ;
begin
write ('Give the number of numbers') ;
read (n) ;
s :=0 ;
for  i := 1 to n do
begin
  write ('Give a number') ;
  read(x) ;
  s :=s+x ;
end;
write ('The sum =', s);
end.
```

Note that even if n ≤ 0, the execution will remain consistent with the value of n and s will remain equal to 0 (s=0), since no iteration will be performed.

Example 2

Write an algorithm that displays the values from *1* to *n* in reverse order, where *n* is a positive integer.
In this algorithm, the goal is to write instructions to display the numbers from *n* to *1* (n, n-1, ..., 3, 2, 1*)* . The display instruction must be repeated *n* times.

In this algorithm, we use the integer variable *i* to count the number of repetitions. This variable is initialized to the value *n* and is decremented at each iteration of the loop. Therefore, the loop step in this algorithm is equal to -1.

```
Algorithm reverse;
Variable
n, i : integer ;
begin
display ('Enter an integer') ;
input(n) ;
For  i from n to 1 step -1 do
   Display (i) ;
Enddo
End.
```

The step must be defined. Otherwise, its default value will be 1.

The translation into Pascal will be as follows:

```
program reverse;
Var
n, i : integer ;
begin
write ('Enter an integer') ;
read(n) ;
for  i := n downto 1 do
   write (i) ;
end.
```

Here too, if n ≤ 0, nothing will be displayed, since no iteration will be performed. This is because it is not possible to go backwards from a value less than 1 to 1.
For example : there is no iteration for the loop " **for** i := 0 **downto** 1 **do** **".**

6.4 The 'Repeat' loop

This structure allows the process to be repeated **one** or more times and to stop on a condition. When the condition is verified, the loop stops; otherwise, it re-executes the process.

Syntax

In algorithmics as in Pascal, the syntax of the Repeat loop is given by:

```
Repeat
<Instruction block> ;
Until (stopping condition);
```

Semantics

Repeat loop works as follows:

1. Executing the <Instruction Block>.
2. Test the value of the <stop condition>.
3. If it is verified, then the loop stops.
   Otherwise, return to step 1.

Notes

- In this loop, the instruction block is executed at least once before the stopping condition is evaluated.
- The condition parameters must be initialized before the loop (by reading or assignment).
- There must be an instruction in the instruction block that modifies the value of the condition, otherwise the number of iterations becomes infinite.
- In Pascal; the block of instructions should not be placed between **begin** and **end** , even if it contains several instructions, because it will be delimited by **repeat** and **until** .

Examples :

Repeat the same previous examples with the **Repeat loop** .

Unlike the For loop, in the Repeat loop, you must start by initialising the counter before the loop.
In addition, at the end of processing, this counter must be modified (incremented or decremented) in order to reach the final value.

Example 1

The algorithm for the first example is as follows:

```
Algorithm sum ;
Variable
i, n :integer ;
s, x : real ;
begin
Display ('Give the number of numbers (greater than 0)');
input (n) ;
s←0 ;
i←1 ;
Repeat
  Display ('Give a number');
  input(x);
  s←s+x ;
  i←i+1;
until (i>n);
Display ('The sum =', s);
end.
```

The translation into Pascal:

```
program sum ;
Var
i, n :integer ;
s, x : real ;
begin
write ('Give the number of numbers (greater than 0)') ;
read (n) ;
s :=0 ;
i :=1 ;
repeat
  write ('Give a number') ;
  read(x) ;
  s :=s+x ;
  i:=i+1;
until (i>n);
write ('The sum =', s);
end.
```

Unlike the **For** loop, in the **Repeat** loop n must be greater than 0 (n>0), otherwise the execution becomes inconsistent with the value of n. Note the message "Enter the number of numbers (greater than 0)".

Example 2

The algorithm for the second example is as follows:

```
Algorithm reverse;
Variable
n, i : integer ;
begin
display ('Enter a non-zero positive integer') ;
input(n) ;
i←n ;
repeat
   Display (i) ;
   i←i-1 ;
until (i<1) ;
End.
```

Here too, n must be greater than 0, otherwise the execution becomes inconsistent. Note the message " Enter a non-zero positive integer'.

For example, for n=-1, the execution begins by displaying -1, then it decrements the value of i (i=-2) and stops because i<n.

6.5 The 'While' loop

This structure allows the process to be repeated **zero** or more times depending on the execution condition. When the execution condition is verified, the process is executed; otherwise, the loop stops.

Syntax

```
While (execution condition)
Do
< Instruction block > ;
Enddo
```

Semantics

While loop works as follows:

1. Test the value of the <execution condition>
2. If it is verified then
   execution of the <Instruction block>
   Return to step 1.
   Otherwise, the loop stops.

Notes

- In this loop, the processing may never be executed; this occurs when the execution condition is false from the start.
- The condition parameters must be initialized before the loop (by reading or assignment).
- There must be an instruction in the instruction block that modifies the value of the condition, otherwise the number of iterations becomes infinite.

In Pascal, the syntax of the **While loop** is as follows:

While (execution condition) **do**
< Instruction block > ;

Reminder

The block of instructions must be placed between **begin** and **end** if it contains several instructions (2 or more).

Examples

Repeat the same previous examples using the **While loop.**

Unlike the **Repeat loop** , in the **While loop** , the condition is evaluated at the beginning. Unlike the **For loop** , in the **While loop** , the counter must be initialized before the loop begins. Furthermore, at the end of the process, this counter must be updated to reach its final value.

Example 1

The algorithm for the first example is as follows:

```
Algorithm sum ;
Variable
i, n :integer ;
s, x : real ;
begin
Display ('Give the number of numbers');
input (n) ;
s←0 ;
i←1 ;
while (i≤n) do
  Display ('Give a number');
  input(x);
  s←s+x ;
  i←i+1;
enddo
Display ('The sum =', s);
end.
```

The translation into Pascal:

```
program sum ;
Var
i, n :integer ;
s, x : real ;
begin
write ('Give the number of numbers') ;
read (n) ;
s :=0 ;
i :=1 ;
while (i<=n) do
begin
  write ('Give a number') ;
  read(x) ;
  s :=s+x ;
  i:=i+1;
end;
write ('The sum =', s);
end.
```

As with the **For** loop, with the **While** loop, execution will remain consistent regardless of the value of n (for i<=0, s will remain equal to 0).

Exemple 2

The translation into Pascal will be as follows:

```
Algorithm reverse;
Variable
n, i : integer ;
begin
display ('Enter an integer') ;
input(n) ;
i←n ;
while (i≥1) do
   Display (i) ;
   i←i-1 ;
enddo
End.
```

The translation into Pascal will be as follows:

```
program reverse;
Var
n, i : integer ;
begin
write ('Enter an integer') ;
read(n) ;
i :=n ;
while (i>=1) do
begin
   write (i) ;
   i:= i-1 ;
end;
end.
```

Here too, if n ≤ 0, then no iteration will be performed.

Notes

1. Any problem that can be solved with the **For loop** can also be solved with the **Repeat** and **While loops** , but the reverse is not true.
2. If the number of iterations is known in advance, it is better to use the **For loop** .
3. If the process is likely not to execute (0 iterations), it is recommended to use the **While loop** .

6.6 Application exercise

Write an algorithm that allows you to:

- Read a series of letters. The reading ends with the input of a period ' **.** '.
- Display the number of consonants and the number of vowels.

In this example, the number of letters to read is not known in advance. The reading operation is repeated until the period (.) is introduced. Therefore, it is not possible to use a **For loop** .

Solution 1

This solution uses the **While** loop.

```
Algorithm  example ;
Variable
letter: character;
nbv, nbc: integer;
begin
Nbv←0 ;
Nbc←0 ;
Display('enter an alphabetical letter) ;
input(letter) ;
while (letter <> '.') do
  if (letter='a') or (letter= 'e') or (letter='i') or (letter='o') or (letter='u') or (letter='y') then
     nbv←nbv+1
  otherwise
     nbc ←nbc+1 ;
  endif
  Display('enter other alphabetical letter') ;
  input(letter) ;
enddo
Display ('number of vowels=',nbv) ;
Display ('number of consonants=',nbc) ;
end.
```

The translation of the algorithm into Pascal language is as follows:

```
program  example ;
Var
letter : char ;
nbv, nbc : integer ;
begin
Nbv :=0 ;
Nbc :=0 ;
Write('enter an alphabetical letter) ;
read(letter) ;
while (letter <> '.') do
begin
  if (letter='a') or (letter= 'e') or (letter='i') or (letter='o') or (letter='u') or (letter='y') then
     nbv :=nbv+1
  else
     nbc :=nbc+1 ;
  Write('enter other alphabetical letter') ;
  read(letter) ;
end;
Write ('number of vowels=',nbv) ;
Write ('number of consonants=',nbc) ;
end.
```

The first operation in this algorithm is to initialize the vowel and consonant counters (*Nbv* and *Nbc,* respectively). To test the loop condition, an initial read is required before the loop. Another read is necessary inside the loop to allow the process to be re-executed. The loop may not execute if a period ('.') is entered as the first letter. In this case, the algorithm displays a value of zero for the number of vowels and consonants.

Solution 2

In this solution, we use the **Repeat** loop.

```
Algorithm  example ;
Variable
letter: character;
nbv, nbc: integer;
begin
Nbv←0 ;
Nbc←0 ;
Display('enter an alphabetical letter) ;
input(letter) ;
if (letter ≠'.') then
repeat
  if (letter='a') or (letter= 'e') or (letter='i') or (letter='o') or (letter='u') or (letter='y') then
     nbv←nbv+1
  otherwise
     nbc ←nbc+1 ;
  endif
  Display('enter other alphabetical letter') ;
  input(letter) ;
until (letter='.') ;
endif
Display ('number of vowels=',nbv) ;
Display ('number of consonants=',nbc) ;
end.
```

The translation of the algorithm into Pascal language is as follows :

```
program  example ;
Var
letter : char ;
nbv, nbc : integer ;
begin
Nbv :=0 ;
Nbc :=0 ;
Write('enter an alphabetical letter) ;
read(letter) ;
if (letter<>'.') then
repeat
  if (letter='a') or (letter= 'e') or (letter='i') or (letter='o') or (letter='u') or (letter='y') then
     nbv :=nbv+1
  else
     nbc :=nbc+1 ;
  Write('enter other alphabetical letter') ;
  read(letter) ;
until (letter='.') ;
Write ('number of vowels=',nbv) ;
Write ('number of consonants=',nbc) ;
end.
```

The same principle applies to the **Repeat loop.**
The (if…then) test before the loop allows us to handle the case where we start directly with the input of the '. '.
If this test is not used, the loop runs even if we start by entering '.' as the first letter and the algorithm displays in this case the number of consonants = 1 (instead of displaying 0).

Bibliography

- Djamel eddine Zegour, *Structures de données et de fichiers. Programmation Pascal et C*, édition CHIHAB,

- John Paul Mueller et Luca Massaron, Les algorithmes pour les Nuls grand format, 2017.

- Serigne Bira Gueye, Algorithmique, Structures des Données et Programmation Pascal et C++ – Tome 1 : Types de Données, Structures de Contrôle, Sous-programmes et Fichiers Externes (2e éd. revue et augmentée, 2022). Editions L'Harmattan.

- Pascal Lienhardt, Bases en algorithmique et en programmation. Cours et exercices corrigés (publié en 2022).

- Djelloul Bouchiha, Algorithmique et programmation en Pascal. Cours avec 190 exercices corrigés (2020).

- Serigne Bira Gueye, Algorithmique, Structures des Données et Programmation Pascal et C++ – Tome 2 : Pointeurs, Listes Chaînées, Arbres, Tris et Programmation Orientée Objet (2019).

- M. BELAID, Algorithmique & Programmation en PASCAL, Cours, Exercices, Travaux Pratiques Corrigés. Pages Bleues Internationales, 2008.

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. Introduction to Algorithms. MIT Press et McGraw-Hill edition. 2001.

- Thomas H. Cormen, Algorithmes - Notions de base. Collection : Sciences Sup, Dunod, 2013.

- http://zegour.esi.dz/Cours/Algo1/Plan1%20.htm

- http://zegour.esi.dz/Publication/Livre3/Livre3.htm

Appendix

Instructions and keywords in algorithmic language	**Instructions and equivalent keywords in Pascal language**
algorithm	program
begin	begin
end	end
variable	var
constant	const
display	write
input	read
integer	integer
real	real
character	char
character string	string
logical (boolean)	boolean
x ← 1	x :=1
x=1	x=1
x≠1	x <>1
x ≤1	x<=1
x ≥1	x >=1
if	*if*
then	*then*
otherwise	*else*
endif	*may be end (after begin)*
and	*and*
or	*or*
not	*not*
while	*while*
do	*do*
enddo	*may be end (after begin)*
repeat	*repeat*
until	*until*
for i from v_1 to v_2 do	*for i :=v_1 to v_2 do*
for i from v_2 to v_1 step -1 do	*for i :=v_2 downto v_1 do*