

Chapter 2: The Simple Sequential Algorithm

1. Foreword

When the computer was invented in the 1940s, it operated using electronic tubes, and its programming (writing commands) was done by entering a series of numbers composed of zero (0) and one (1). It was difficult for programmers. However, with the creation of the transistor, computers became much smaller, and their capabilities increased. Easier-to-use programming languages were invented, which are very similar to human language. These languages are called high-level languages. Any program written in these languages can be quickly and automatically translated into machine language (0s and 1s) using a compiler.

In this chapter, we will cover the concept of language, algorithm structure, the concept of variables and constants, and also explore some types of simple instructions such as assignment, input, and output.

2. Notion of Language and Algorithmic Language

2.1. Language

Language is a means of communication and understanding among human beings. In the case of a computer, it is the way a computer understands human commands. Language consists of an alphabet, symbols, vocabulary, grammatical rules, and meanings.

- **Alphabet:** It is a set of letters, numbers, and symbols.
- **Vocabulary:** A collection of symbols and words, whether reserved words or those defined by the programmer.
- **Syntax:** Rules or laws that govern the grouping and placement of symbols and vocabulary.
- **Semantics:** Specifies the meaning of each instruction that can be constructed in the language, particularly what it will produce during execution.

2.2. Programming Language

A programming language provides us with a framework to develop algorithms and produce programs that a computer can execute. It allows us to describe the data structures that will be processed by the computer and the operations that will be performed. It serves as an intermediate language between human language and machine language. Humans can understand it, and computers can translate it into the 0s and 1s of machine language that they understand.

There are many programming languages, each with its own advantages and rules, making them suitable to varying degrees for specific types of software. Programmers must know some of these languages and know which language is appropriate for each type of application.

2.2.1. Types of Programming Languages

Programming languages can be divided into two parts:

- **Interpreted:** If the program is not fully translated into machine language, but rather translated and executed instruction by instruction. Examples: Matlab and web languages.
- **Compiled:** If the program is fully translated into machine language before execution. Example: C.

Examples of Programming Languages:

- C and C++: which will be used in this course and are considered the parent language of many others.
- Pascal: used for educational purposes and closely related to algorithms.

- Delphi and WinDev: good for developing management software.
- Java: suitable for network and mobile applications.
- C# (C-Sharp) and Visual Basic: useful for developing software specific to the Windows environment.
- Objective C (Xcode): used for software development for Apple products (Mac, iPad, and iPhone).
- PHP: specific to web development.
- Matlab and Python: specific to data analysis and used by engineers.

2.2.2. Compiler

A computer program that converts the source code written in a particular programming language into targeted code that can be executed directly by a computer. There are many programming languages that a programmer cannot know all of them. To enable programmers to work in teams and exchange solutions, they need to formulate these solutions in algorithmic language. Algorithmic language is the common language of all programmers.

2.2.3. Integrated Development Environment (IDE)

To write a program, you can use any text editor such as Notepad. However, this method makes the development process very difficult. Therefore, there is a set of programs that provide all the necessary tools for the development process called IDE (Integrated Development Environment). IDE provides all the necessary tools for designing, developing, testing, debugging, and deploying applications, which makes development easier and faster. The IDE includes all the necessary tools to start designing applications, such as:

- Code editor: for writing and editing program code. It performs automatic formatting, making the process of reading easier.
- Project manager: to manage the files that make up a single project.
- Debugger: detects and corrects errors in the code.
- Shortcuts to compile and run the program.
- Other tools...

Examples of IDEs: Dev-C++, Embarcadero, Visual Studio...

2.3. Basic Syntactic Elements

2.3.1. Reserved Words

Reserved words are words that have preexisting meaning in the programming language and cannot be used by the programmer to create new elements. Each language has its own reserved words, such as "algorithm" "begin," and "end" in the algorithm, and "if" and "while" in C.

2.3.2. Values

3. Values can be numbers, characters (always enclosed in single quotes '), strings of characters (always enclosed in double quotes ""), true, or false. See types.

example: -2, 7, 3.12, 6e-7, 'k', 'خ', '1', '!', "azerty", "سلام", true, false

3.1.1. Identifier

An identifier is the name given by the programmer to any element of the algorithm that they want to create. Examples include the name of the algorithm, variable name, type name, constant name, function name, etc. There are rules and conditions in the C language that we adopt in the algorithm for naming identifiers:

Rules for Naming Identifiers:

- The identifier name can only contain literal symbols and numerals from A to Z, a to z, and 0 to 9, as well as the underscore symbol "_".
- It must be a single word, meaning the name cannot contain spaces " ".
- It must start with a letter or "_", not with a digit.
- It must not be a reserved word.
- The identifier must be unique; it is not possible to define more than one element with the same name.
- It is recommended to use meaningful names, for example, we use "Width" instead of "x".

Examples of Valid Identifiers:

x, pi, Mat_info, isEmpty, n5, _if, _0a (it's better to avoid it)

Examples of Invalid Identifiers:

α , π , π , ϵ , élève (unacceptable symbols)

3a (starts with a digit)

Mat info (contains a space)

begin, end, if (reserved words)

3.1.2. Operations• **Arithmetic Operations:**

Operation	C	comment	example
- +	- +	Sign	+3 -7 -a
- +	- +	The two operands are integers, the result is an integer. one is real, result is a real number.	5+3.0 real
*	*	For the product, like addition and subtraction.	5*3 integer
/	/	for real division. The result is always a real number	5/3 or 5.0/3 in C
mod	%	To calculate the remainder of the division. Both operands are integers The result is always an integer.	5 mod 3 or 5%3 in C
div	/	To calculate the quotient (integer division). Like the remainder	5 div 3 or 5/3 in C
^		to calculate the power of a number. In C, the function pow() is used, the result is a real number	5^2 or pow(5,2) in C
√		to calculate the square root of a number. In C, the function sqrt() is used	√5 or sqrt(5) in C

• **Relational Operators:**

Algo	C	comment
>, >=, <, <=		the same in C

=	==	In C «==» twice = is read as equal, while «=» is used for assignment and read receives.
≠	!=	Not equal

The result of the comparison is always a logical **Boolean**, i.e. **true** or **false**.

- **Logical Operators:**

Operation		C	observation
not	negation	!	true if the operand is false, and false if it's true.
and		&&	true only if both operands are true, otherwise, it returns false
or			true if at least one of the operands is true, otherwise, false.
xor	exclusive OR		true if one is true and the other is false, otherwise, it returns false.
=	equivalence	==	true if both operands are equal..

- **String Operators:**

+: used to concatenate two strings. For example, "hello" + "world" gives "helloworld" without adding spaces. "hello" + " " + "world" gives "hello world".

- **Operator Priority:**

When evaluating an expression, we follow the priority summarized in the following table. If the priorities are equal, priority is given to the operation on the left.

priority	the operation
0	()
1	+ and - sign, not.
2	* / mod div
3	- +
4	>, >=, <, <=
5	≠, =
6	and
7	or

Parentheses are used to change priority (and sometimes for readability).

3.1.3. Expression

An expression is a structure of values and identifiers, connected by operations. When evaluated, it results in a single value. Expressions are created using values, variables, parentheses, and operations.

Example:

Assuming a=2, b=3, and ok=true,

expression	result	expression	result	expression	result
5	5	a+3	5	ok	True
a	2	"In"+"fo"	Info	a*(b-7)>8 et ok	False

3.1.4. Instruction

An instruction (statement) is a command or step in the solution, meaning it is the action that will be executed.

3.1.5. Block of Instructions

A block of instructions is a set of instructions that begins with the word "Begin" and ends with "End," or begins with a reserved word defining the beginning, such as "if," and ends with "End" + the corresponding starting word, e.g., "End If." In C, it starts with "{" and ends with "}".

example:

algorithm	C
Begin Instruction 1 ... Instruction n End	{ Instruction 1 ; ... Instruction n ; }

3.1.6. Comments

Comments are texts that are ignored during the translation of the program and are not part of the algorithm. They are added to programs to provide explanations and facilitate understanding.

In C, comments can be added using «//» for single-line comments. It begins with // and ends with a line break.

Comments can also be added, starting with « /* » and ending with « */ », which can extend over several lines.

example:

```
// One-line commentary
/* Comment
It can span multiple lines*/
```

3.2. Expressing the Algorithm

An algorithm can be expressed by writing it in natural language, such as Arabic, French, or English. However, natural language is ambiguous and imprecise. Therefore, we write the algorithm using Pseudocode, flowcharts, or programming languages.

- 1- **Pseudocode:** Describes the algorithm in human languages like Arabic, French, or English, in a manner similar to programming languages. Some use many details (to be closer to programming languages), while others use fewer details (closer to human language). There is no specific rule for writing this type of code.
- 2- **Flowchart:** An illustrated representation of the algorithm that shows the steps to solve the problem from start to finish while abstracting away the details to provide an overview of the solution. Arrows and agreed-upon geometric shapes are used to represent the steps.
- 3- **Code:** Where the algorithm is written in a programming language directly, such as C, so that the computer can translate it into binary language for direct execution by the processor.

4. Parts of an Algorithm

An algorithm consists of two main parts: data and instructions. An algorithm's structure is very similar to a recipe for cooking. It typically consists of a title, followed by ingredients, and finally, the preparation method.

An algorithm takes the following form:

```

Algorithm name
  Declaration of the data needed
Begin
  Instructions
End

```

The algorithm is composed of three basic parts:

- Header: Comprised of the word "Algorithm," followed by the name that explains the problem to be solved. The name should be a valid identifier.
- Declarations: Reserved for reserving memory space for data (constants and variables) that will be used as input and output.
- Instructions: A set of steps or commands that will be executed during the algorithm's execution. It starts with "Begin" and ends with "End." There are five main types of instructions:
 1. Assignment instruction.
 2. Read (input) instruction.
 3. Write (output) instruction.
 4. Conditional instruction.
 5. Iterative (loop) instruction.

5. Data: Variables and Constants

5.1. Constant

A constant is a value (numeric or symbolic) that has a name and cannot be changed during program execution.

Constant declaration:

```
Const Identifier = value
```

Const ou **Constant**: These are two reserved words that allow constant declaration.

Identifier: The name given to the constant.

Value: The value assigned to the constant.

Example:

```

Const
  PI = 3.1415926
  DEP = "قسم الاعلام الالى"

```

Advantages of Constants:

- Condenses the code, where a long phrase can be replaced with a short word, such as using "PI" instead of 3.1415926.
- Helps avoid errors by providing a meaningful name. For example, "PI" instead of 3.1415926.
- Simplifies code maintenance, as the value needs to be changed in one place only.

5.2. Variable

A variable is a location in memory used to store data. It has a name, a type, and a value (address in the second semester).

- **Name:** Identifier used by the programmer to refer to and manipulate the variable. For example, "weight."
- **Type:** In computers, everything is represented as 0 and 1. The type determines how it is translated, as well as the size of memory to reserve. For example, "int" (32 bits).
- **Value:** The content of the bits that make up the variable, i.e., its value. Typically, this is the part that changes during program execution. For example, 1101 represents the number 13, or -5 if we consider the leftmost 1 as the sign "-".

Variable Declaration:

Var Identifier: Type

Var ou **Variable:** These are two reserved words that allow variable declaration.

Identifier: The name given to the variable.

Type: The type of the variable.

A comma "," can be used to declare multiple variables of the same type.

Examples:

```
Var
    age: integer
    gender: character
    x, y, z: real
```

Note: By convention, constant names are written in uppercase, and variable names are written in lowercase.

6. Data Types

A data type represents the domain to which data belongs, such as numbers, text, images, audio, or video. The data type determines how the bits, which compose the variable, are translated and the size of memory to reserve, i.e., the number of bits and allowed operations. When defining a variable, its type must be specified. There are five basic data types in the algorithm:

1. Integer such as: -5, 0, 1, 13
2. Real: -7, 0, 1, 3.14, 2.7e03
3. Boolean Contains only true or false.
4. Character: Includes all symbols on the keyboard, such as digits, letters in all languages, and printed (visual) and unprinted symbols. They are always enclosed in single quotes (e.g., 'a', 'M', '1', '+', ',', 'س').
5. String: A set of symbols, with a length of 0 or more, always enclosed in double quotes (e.g., "computer," "Good luck\n," "1", "3.14").

Notes:

- We use "." instead of "," to express decimal numbers.
- 1 is not the same as 1., not the same as '1', and not the same as "1." The first is an integer, the second is a real number, the third is a character, and the last is a string.
- 'a' is not the same as "a." The first is a character, and the second is a string of length 1.

- Lowercase letters are not the same as uppercase letters. For example, 'a' is not the same as 'A.'
- Some symbols (keys) do not print. For example, space ' ' or newline '\n'.
- The backslash (\) is used to represent some invisible or special symbols visually. For example, newline '\n' and tab '\t'. To print double quotes, we use \" and to print a backslash, we use \\.
- There is an empty string, denoted as "", which contains no characters and has a length of 0.

7. Basic Instructions

7.1. Assignment

This is the process that allows us to store a value in a variable.

Syntax:

`variable ← exp`

- **variable:** This is the name of a variable.
- **exp:** It is an expression (identifiers, values, and operations, see 2.3.4), calculated to obtain a unique value placed in the variable.

← read as "gets" in English. The arrow always points to the variable.

A variable can hold only one value at each point in the program's execution. When the operation is performed, only the left variable changes. It loses its old value and takes on the new one. For the assignment process to work correctly, the value of the right expression and the left variable must be of the same type or at least compatible types.

Example:

<code>a ← 5</code>	a gets 5
<code>b ← a * 2</code>	b gets 10
<code>a ← 0</code>	a gets 0
<code>b ← b - 1</code>	b gets 9
<code>c ← 'b'</code>	c gets the letter b
<code>d ← b > a</code>	d gets true
<code>s ← "name"</code>	s gets the word "name"

Before a variable can be used, it must be declared and assigned an initial value. To obtain the value of any variable or constant, simply write its name.

7.2. Input/Output Instructions

To interact with the user, the programmer has two instructions: Read() and Write().

7.2.1. Input: Read()

Read() is a ready-to-use function in algorithms. You input a value from the user, via the keyboard, and assign it to the variable inside parentheses. It is always used for data entry.

Syntax:

`Read(variable)`

- **variable:** This is the name of a variable. read() can only be used with variables.

When the program is executed and the input instruction `read()` is encountered, execution is paused until the user enters data. The input process ends by pressing the Enter key. The program will continue to execute. Several variables can be entered at once, separated by a comma ",". In this case, the user enters the value of the first variable, then presses the space key, then enters the value of the second variable, and presses the Enter key only after entering the value of the last variable.

Example:

<code>Read (name)</code>	The user enters a series of letters, for example, <Muhammad>, then presses the Enter key.
<code>Read (a, b)</code>	They enter a number, for example, "15", then press space, then enter the second number, for example, "20", then press the Enter key.

7.2.2. Output: Write()

`Write()` is a ready-to-use function in the algorithm. It displays on the screen whatever we put inside its parentheses. It is always used to print results.

Syntax:

`Write (exp)`

or

`Write ("message")`

- `exp`: This is an expression, calculated to obtain a single value, to display on the screen.
- `"message"`: Any text you want to display as is on the screen. It is not calculated. It can be in any language or set of letters. It must be enclosed in double quotes, which are not displayed on the screen.

Several values and texts can be displayed at once, separated by a comma ",".

Example:

<code>Write (name)</code>	The value of the variable name appears on the screen, for example, <Mohammed>.
<code>a←5</code> <code>Write (a+3)</code>	Displays 8 without changing the value of a.
<code>Write ("square of ", a, " is ", a * a)</code>	Displays: square of 5 is 25
<code>Write ("b=", a)</code>	Displays: b=5

Notes:

- Always before the `Read()` instruction comes the `Write()` instruction, to explain to the user what is expected to be entered.
- The user writes on the keyboard, while the program (computer) reads <Read()> from the keyboard, and the program writes <Write()> on the screen, while the user reads from the screen.
- <Read()> can be generalized for the input of all input units, and <Write()> for the output of all output units.

8. Building a Simple Algorithm

After seeing that the algorithm consists of 3 parts, namely: header, declaration, and instructions, and we have learned to declare constants and variables, and we have learned 3 types of instructions, namely: assignment, reading, and writing. Now we can write simple algorithms.

To know the variables, we ask the question: "What data is needed and what is the expected result?" The instruction part usually consists of three basic steps:

- The first step, "Inputs": the data needed for implementation is entered using the <Read()> instruction.
- The second step, "Processing": It contains a set of instructions necessary to solve the problem using the assignment instruction.
- The third step, "Outputs": the results are presented using the <Write()> instruction.

Example 1:

Write an algorithm that calculates the area of a circle.

```

Algorithm circle_area
Const
    P=3.14
Var
    r, s: integer // r is the radius and s is the surface
Begin
    Write("Enter the radius")
    Read(r)
    s ← p * r * r
    Write("The area of the circle is:", s)
End

```

Example 2:

Write an algorithm that calculates the average for ASD1.

```

Algorithm avg_ASD1
Var
    exam, td, tp, avg: real
Begin
    Write("Enter the exam score, tutorial score, and practical score")
    Read(exam, td, tp)
    avg ← (exam * 3 + td + tp) / 5
    Write("The average is:", avg)
End

```

8.1. Execution of an Algorithm

The execution of the algorithm aims to know the value of each variable after each instruction. Where execution starts from the word Begin to the word End. At the beginning, the values of the variables are undefined (empty), then after each assignment or reading, the value of the variable changes.

Example

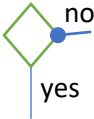
Algorithm mirror	Execution		
Var			
a, b, c: integer	a	b	c
Begin			
a←357	357	?	?
c←0	357	?	0
b←a mod 10	357	7	0

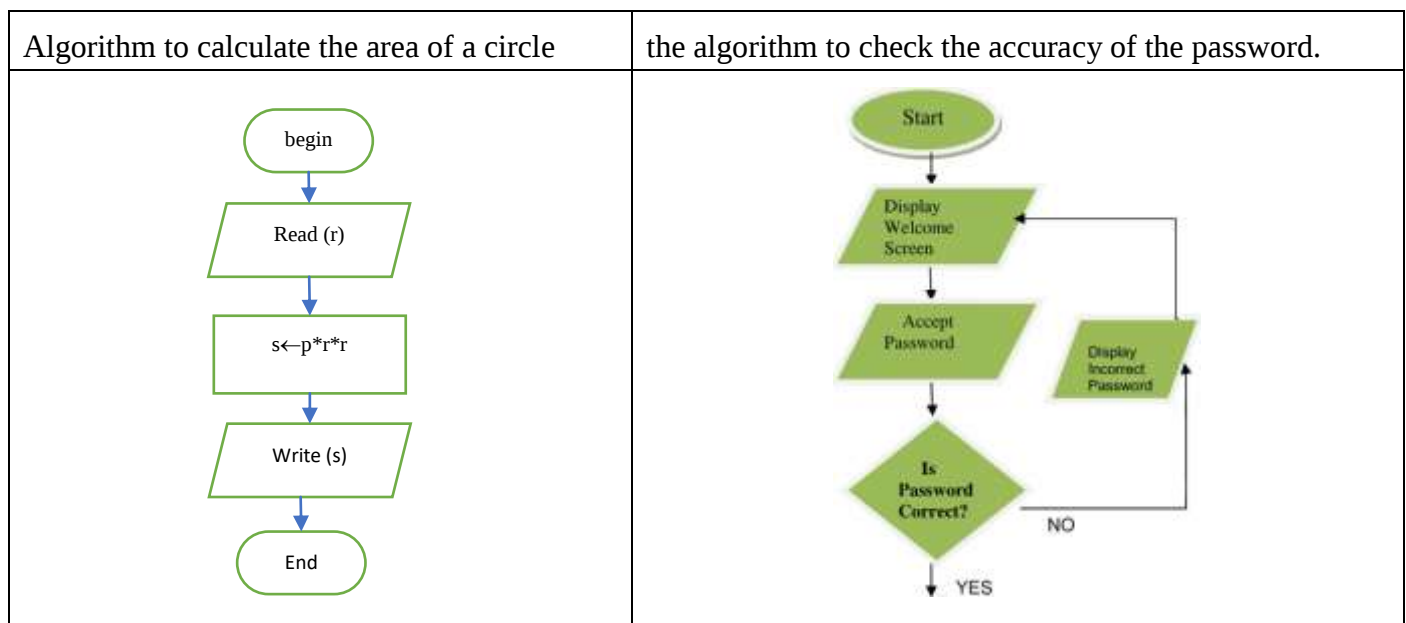
$c \leftarrow c * 10 + b$	357	7	7
$a \leftarrow a \text{ div } 10$	35	7	7
$b \leftarrow a \text{ mod } 10$	35	5	7
$c \leftarrow c * 10 + b$	35	5	75
$a \leftarrow a \text{ div } 10$	3	5	75
$b \leftarrow a \text{ mod } 10$	3	3	75
$c \leftarrow c * 10 + b$	3	3	753
End			

9. Representing an Algorithm with Flowcharts

A flowchart is a visual representation of an algorithm before its programming. It shows us the sequence of operations and gives us the overall structure of the algorithm's components. Flowcharts have many advantages: they provide a better visualization of ideas and are easily understood by everyone, facilitating teamwork.

Several shapes are used in a flowchart, with the most important being:

Symbol	Use
	Start, end, interruption
	Input - Output read()/write()
	Processing Symbols like assignment
	Logical Symbols: Choices with conditions

Example:**10. Translation into C Language**

"C" is a fully compiled high-level imperative language. It is one of the most widely used programming languages in the world and is considered the parent language of many programming languages. In this course, we will use Dev-C++ as an IDE. It's worth noting that "C" is case-sensitive, distinguishing between uppercase and lowercase letters. <main> is not the same as <Main>, <MAIN>, or <mAin>. Therefore, we recommend always writing in lowercase.

All simple statements (declaration, assignment, I/O, return) must end with a semicolon ";".

10.1. The Preprocessor

Before the program is actually compiled, the source code files are processed by a preprocessor, which resolves certain directives given to it. For example, including other files (libraries), replacing words with other phrases (macros).

A directive given to a preprocessor always starts with a #.

10.1.1. #include

The #include directive instructs the compiler to include the content of another file into the current program's code.

```
#include < filename >
```

Generally, these files are libraries of predefined and ready-to-use functions. Example:

Example:

- To use I/O functions (scanf and printf), we use the stdio.h library.
- To use mathematical functions (sin, cos, exp, pow, sqrt, ...), we use the math.h library.
- To use string functions (strlen, ...), we use the string.h library.

```
#include <stdio.h>
```

```
#include <math.h>
```

10.1.2. Macro

A macro, in its simplest form, is defined as follows:

```
#define macro_name replacement_text
```

Example:

```
#define N 10
```

The preprocessor replaces all occurrences of the word N with 10.

10.2. Types

10.2.1. Predefined Types

The following tables summarize the basic types in algorithms and their equivalents in C.

- Integers in algorithms from $-\infty$ to $+\infty$:

Types	Size (bytes)	Size (bits)	Range
char	1	8	$-2^7, 2^7-1$
short	2	16	$-2^{15}, 2^{15}-1$
long	4	32	$-2^{31}, 2^{31}-1$
int	4	32	$-2^{31}, 2^{31}-1$

- Natural numbers in algorithms from 0 to $+\infty$. Since integers contain natural numbers, we usually use integers to express them. You can also add unsigned before a type in C to express only natural numbers.

Types	Size (bytes)	Size (bits)	Range
unsigned char	1	8	$0, 2^8-1$
unsigned short	2	16	$0, 2^{16}-1$
unsigned long	4	32	$0, 2^{32}-1$
unsigned int	4	32	$0, 2^{32}-1$

- Real numbers:

Types	Size (bytes)	Precision	Range
float	4	6	
double	8	8	
long double	10	8	

- Boolean** type: There is no boolean type in C, but int is used instead. True is represented by the number 1 and false by 0. Any number other than 0 is translated as true.
- Character type is char.
- String type: To express strings in C, we use arrays (Chapter 5) of char[] or pointers (second semester) of char*.

10.2.2. Notes

- The **int** type is the generic type for integers.
- The **char** type is used for both integers and characters, where each character is associated with a number.
- In C++, there's the **bool** type for Boolean and **string** for string.
- In this course, we use **char** for characters, **int** for integers and Boolean, and **float** for real numbers.

10.2.3. Type Conversion

- **Implicit:** This is done automatically by the compiler. It goes from a smaller type to a larger type without losing information. For example, char to int, int to float, or float to double. Converting 5 to float becomes 5.0.
- **Explicit (casting):** When the conversion could lead to losing information, it's necessary to declare that the programmer is performing the operation. You need to specify the destination type in parentheses before the expression to convert it. For example, **(int) 3.1416** converts it to 3.

10.2.4. Defining New Types

To create a new type or rename a specific type, the typedef keyword is used:

```
typedef old_name new_name;
```

Example: To declare a new type named Banane with an underlying type of int, you use:

```
typedef int Banane;
```

10.3. Declaration of Variables and Constants

10.3.1. Declaration of Variables

Syntax:

```
Type variable;
```

example:

```
int age;
char gender;
float x, y, z;
Banane b;
```

10.3.2. Declaration of Constants

Syntax:

```
const Type Identifier = value;
```

or

```
Type const Identifier = value;
```

example:

```
int const N = 10;
const float P1 = 3.1415926;
const char[] DEP = "قسم الاعلام الالي";
```

Note: Macros can be used to declare constants as well, like:

```
#define DEP "قسم الاعلام الالي"
```

10.4. Assignment

We use = instead of $\leftarrow$, and read "receives" rather than "equals".

Syntax:

```
variable = expression;
```

example:

a=5;	a gets 5
b=a*2;	b gets 10
a=0;	a gets 0
b=b-1;	b gets 9

c='b' ;	c gets the letter b
d=b>a ;	d gets 1
s="name" ;	s gets the word name

Declaration with initialization

```
float x, y=3, z; //y is initialized with 3
```

Multiple assignments with right priority

```
b=3 ;
a=b+5+3 ;
```

b takes 8, then a takes the value of b, which is 8

Assignment shortcuts in C.

expression	comment	example
v+=exp ; $\Leftrightarrow$ v=v+(exp) ;	Parentheses are important	x=2 ; x*=5+3 ; $\Leftrightarrow$ x=x*(5+3) ; $\neq$ x=x*5+3 ; x become 16
v-=exp ; $\Leftrightarrow$ v=v-(exp) ;		
v*=exp ; $\Leftrightarrow$ v=v*(exp) ;		
v/=exp ; $\Leftrightarrow$ v=v/(exp) ;		
v%=exp ; $\Leftrightarrow$ v=v%(exp) ;		
v++ ; $\Rightarrow$ v=v+1 ;	If contained in another instruction, the calculation is performed with the current value of the variable, and after completion, 1 is added to or subtracted from the variable depending on the operation.	x=2 ; y=3+x++ ; $\Leftrightarrow$ y=3+x ; x=x+1 ; In other words, y becomes 5 and x 3
v-- ; $\Rightarrow$ v=v-1 ;		
++v ; $\Rightarrow$ v=v+1 ;	If included in another instruction, 1 is added or subtracted according to the operation before the expression is calculated, using the new variable value.	x=2 ; y=3+ ++x ; $\Leftrightarrow$ x=x+1 ; y=3+x ; In other words, y becomes 6 and x 3
--v ; $\Rightarrow$ v=v-1 ;		

Note:

- v++ is not identical to v+1, but v=v+1.
- v++, ++v, v=v+1, v+=1 are all equivalent if presented in a separate instruction.
- The difference between v++, ++v when it appears in another sentence, is the pre- or post-addition.

Empty instruction: It's an instruction that does nothing, like the semicolon instruction " ; " and instructions such as i + 1, where the result is calculated and ignored, without any change in the program state.

10.5. Input and Output**10.5.1. printf (print formatted)**

The printf function, defined in the stdio.h library, is used to write formatted data to the standard output unit, which is typically the screen.

Syntax:

```
printf (format, expression_1,... , expression_n);
```

- **format:** A text or string that is displayed as is on the screen, except for the "%" symbol to indicate expression formats and the "\" character for escape sequences.

- **expression:** Computed to obtain a single value, which will be displayed on the screen in the specified format <format>.

The format follows this structure:

% [flags] [width] [.prec] **type_char**

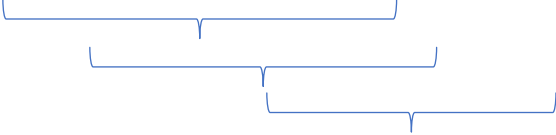
where it always starts with %.

- **flags**
 - -: Left-align the output.
 - +: Show the sign of the number.
 - Empty: Display a space instead of + for positive numbers.
- **width:** Represents the minimum number of digits to display for a specific value. If this number is greater than the required size, the difference is filled with spaces or zeros, depending on whether the number starts with 0 or not.
- **.prec:** The number of digits after the decimal point for floating-point numbers.
- **type_char:** A character representing the type of value to output.
-

format	use
%d	To input or output a number in the d ecimal system (10)
%o	To input or output a number in the o ctal system (8)
%x	To input or output a number in the hexadecimal system (16)
%u	To input or output an unsigned natural number u nsigned
%i	To input or output an integer (i nt) like %d
%f	To input or output a real number (f loat)
%c	To input or output a character (c har)
%s	To input or output a s tring (char[], char*)
%e	To input or output a nbr in scientific format such as 3 e -2

- Some characters are special, so we must use an escape technique to use them. In C, the escape character is (), backslash, and we use it to add a new line '\n' or a tabulation (a large space) '\t'. To print double quotes ("), we use ", to print a backslash () we use \, and to print % we use %%.

example:

<code>printf("Hello");</code>	displays Hello
<code>int a=13;</code> <code>printf("a=(%d)10\ta=(%o)8\ta=(%X)16\n",a ,a ,a);</code>	<code>a=(13)10 a=(15)8 a=(D)16</code>
<code>a=66;</code> <code>printf("a=%i\ta=%f\ta=%c\ta=%c\n",a ,a ,a ,a+32);</code> 	<code>a=66 a=0.000000 a=B a=b</code> Because 66 is not necessarily 66 in float And 66, if we see it as a character, represents the letter B, while 66 + 32 = 98 represents the coding of the letter b

	in lower case.
float pi=3.1415926; printf("%f\t%.4f\t%06.2f" ,pi ,pi ,pi);	3.141593 3.1416 003.14

10.5.2. scanf (scan formatted)

The **scanf** function, defined in the `stdio.h` library, is used to read formatted data from the standard input unit, typically the keyboard. The function copies the value entered by the user to the variable's memory location. Therefore, **&** is used before the variable name.

Syntax:

```
scanf(format, &variable_1, ... , &variable_n);
```

- **format:** A string representing the reading format.
- **variable:** The variable name. It's preceded by **&**, except for string variables (pointer types).

The format takes the following form:

% [width] type_char

- **width::** A number that controls the maximum number of characters to be read in the current input field.
- **type_char::** A character representing the type of value to be entered. The same symbols in the table for `printf`.

Example:

<code>scanf("%s", name);</code>	& is not used to read a string.
<code>scanf("%d%f", &a, &b);</code>	Enter a number, then press <i>space</i> , then enter the second number, then press <i>enter</i> .

Note: It is recommended to enter only one value per `scanf` instruction.

Avoiding problems with characters and strings in scanf

When reading a character using `scanf("%c"...)`, the user enters the initial character and subsequently presses the Enter key. This action results in the creation of the '\n' character, which remains stored in memory. When the program encounters the `scanf("%c", ...)` instruction a second time, it doesn't await the user's input; instead, it directly assigns the '\n' character to the second variable.

To overcome this predicament, in the second `scanf`, we introduce a space ' ' after the % symbol, as demonstrated here: `scanf("% c"...)`. Alternatively, we can utilize the `getch` function from the `string.h` library to resolve this issue.

Example: We'll try to enter 3 letters a, b, c in the variables `c1`, `c2` and `c3`. We use:

```
scanf("%c", &c1) ;
scanf("%c", &c2) ;
scanf("%c", &c3) ;
```

The user presses a, then enter. The program assigns a to `c1` and '\n' to `c2`, and waits for the user to enter the second character to be assigned to `c3`. To avoid this problem, we use:

```
scanf("%c", &c1) ;
scanf("% c", &c2) ;
scanf("% c", &c3) ;
```

- The issue with `scanf` arises when attempting to input a string containing spaces into a variable.

For instance:

```
scanf("%s%s", v1, v2);
```

Upon entering the words "math info" and pressing Enter, the program assigns the first word to v1 and the second word to v2. However, if we intend to input both words, such as a compound name, into a variable, we use:

```
scanf("%s", v1);
```

Upon entering the words "math info" and pressing Enter, the program assigns only the first word to v1, while the second word is lost.

To circumvent this problem, we employ the **gets** function, defined in the **string.h** library:

```
#include <string.h>
gets(v1);
```

10.6. Structure of a C Program

```
1. #include <stdio.h>
2. //Public declarations (constants, types and variables)
3. int main()
4. {
5.     //Local declarations (constants and variables)
6.     //instructions
7.     return 0;
8. }
```

explanation:

1. Include the **stdio.h** library which contains **scanf** and **printf**.
2. Place for public declarations.
3. **main()**: Every program must have a starting function called **main** which indicates the entry point of the program.
4. Start of the **main** function body, corresponding to the "begin" in algorithms.
5. Place for local declarations.
6. Instructions.
7. **return**: The **main** function must return an integer (**int**). It returns 0 to the operating system to indicate successful execution.
8. End of the **main** function and the program, corresponding to the "end" in algorithms.

Observations:

- Declarations can be made either in the place of public or local declarations.
- Code formatting, alignment, margins, spaces, and line breaks after special characters ([{}]) =+, :: etc. are not significant in the program. Much of the program can be composed on a single line, with semicolons serving as separators between statements.
- Code formatting, alignment, margins, spaces, and line breaks should be used for program readability.

An example demonstrating the process of translating an algorithm into C.

algorithm	C	comment
Algorithm circle_area		The algorithm name becomes the file name circle_area.c

Const P=3.14	Const float P=3.14;	can use #define P 3.14
Var r, s:entier	int r, s;	There is no var word in C
//r radius and s area	// r the radius and s the surface	This is not part of the program, but only for explanation.
begin	int main() {	Variables can be declared after {.
Write ("Enter radius ")	printf("Enter radius \n");	\n to return to the line
Read (r)	scanf("%d", &r);	Don't forget & before the variable
s←p*r*r	s=p*r*r;	Each instruction ends with ;
Write ("The area of the circle is:" , s)	printf("The area of the circle is: %d" , s);	The format must be given
End	}	

example 2

Write a program to calculate the average for ADS1.

```
#include <stdio.h>
int main() {
    float cont, td, tp, moy ;
    printf("Enter the exam score \n") ;
    scanf("%f", &cont) ;
    printf("Enter TD score \n") ;
    scanf("%f", &td) ;
    printf("Enter TP score \n") ;
    scanf("%f", &tp) ;
    moy = (cont * 3 + td + tp) / 5 ;
    printf("The average is %.2f" , moy) ;
    return 0;
}
```