



Lebanese University
Faculty of Sciences



I1101

Imperative Programming

Department of Applied Math



2nd Semestre
2018 ~ 2019

© Copyright Reserved

Copyright Reserved for the Lebanese University ©2018

This material is only for academic use by the students at the Faculty of Sciences in the Lebanese University, it should not be distributed for purchase or photocopy anywhere under the threat of legal prosecution.

حقوق الطبع والنشر محفوظة للجامعة اللبنانية ©2018

إن هذه النسخة موضوعة بتصرف طلاب كلية العلوم في الجامعة اللبنانية و لهدف اكاڤمي فقط لا غير وعليه يمنع توزيع اي نسخة (ورقية او الكترونية) لاي جهة اخرى او التصوير والبيع في كافة المكتبات تحت طائلة الملاحقة القانونية.

1 Algorithm vs Program

1.1 Definition

An algorithm is a set of instructions (or statements) that is used to perform a certain task. In other terms, an algorithm is the description of a procedure which terminates with a result. The procedure is a set of successive treatments of given input data in order to obtain output data (Figure 1). We mean by input data, the initial data to be treated in order to obtain the result that is the output data. Once the result is achieved, then the algorithm is terminated and the task is performed. The algorithm is also known as pseudo code.

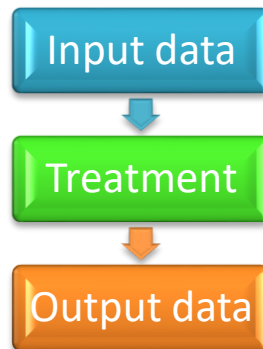


Figure 1: General outline of an algorithm

Cooking recipe is a classic example of algorithm.

Problem: Preparation of omelet

Input data: 2 eggs, 3g of Sault, 20g of margarine.

Treatment:

Crack the eggs in a bowl,
Add the salt, and then mix them,
Heat the margarine then add the mixture and cook for 2 minutes.

} statements

Output data: an omelet.

So, an algorithm is a set of instructions which, if followed, performs a particular task. On the other hand, a program is a set of instructions **written in a particular syntax (called language)**, which if followed performs a particular task. Thus, we can say that an algorithm is language independent, unlike a program.









1.2 Flowchart

Flowchart is a diagrammatic representation of an algorithm. Flowchart are very helpful in writing program and explaining program to others.

1.2.1 Symbols used in Flowchart

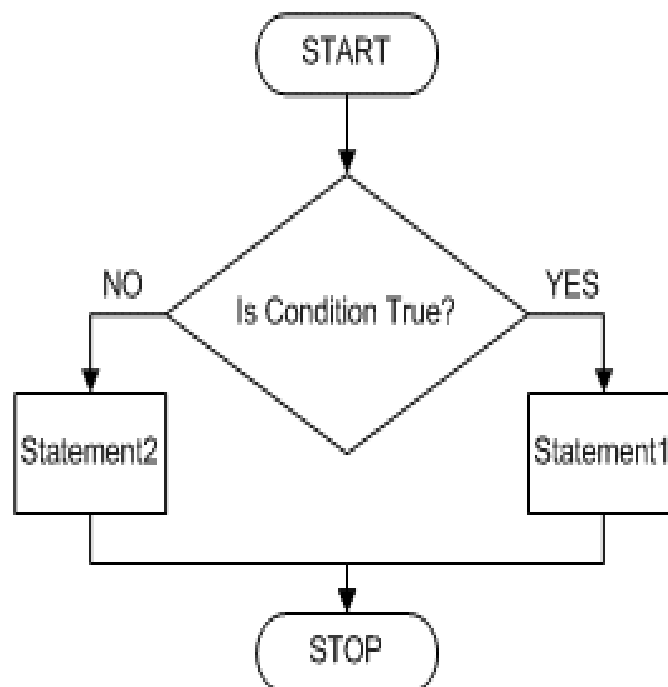
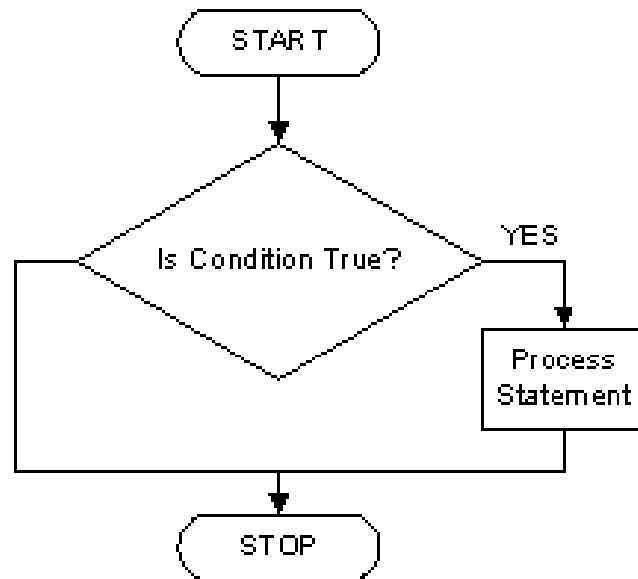
Different symbols are used for different states in flowchart. For example: Input/Output and decision making has different symbols. Table 1 describes all the symbols that are used in making flowchart:

Table 1: Symbols used in flowchart

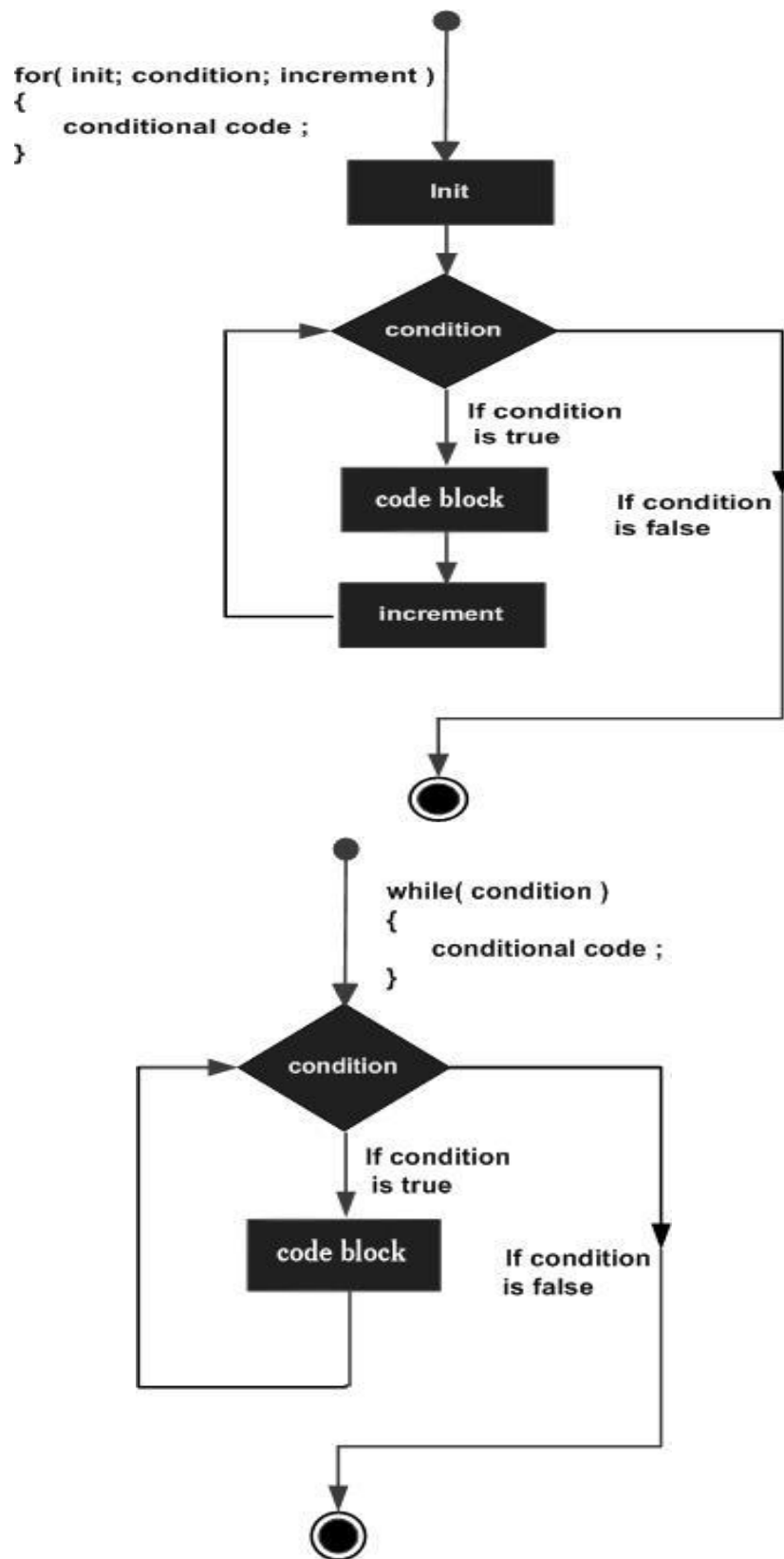
Symbol	Purpose	Description
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for airthmetic operations and data-manipulations.
	Desicion	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect flowchart portion on different page.
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

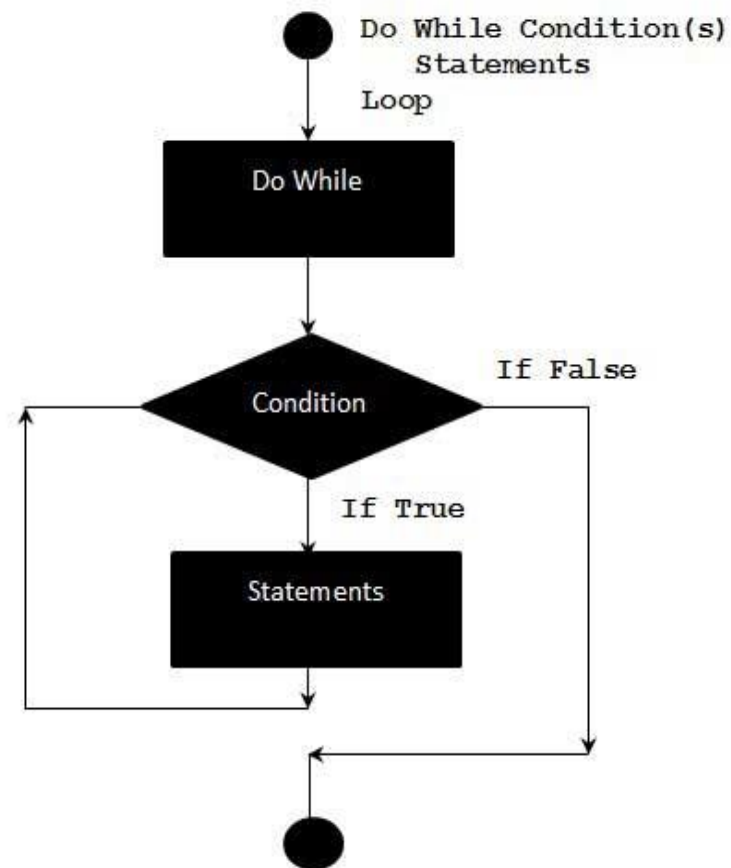
1.2.2 Useful Structures

1.2.2.1 Conditions



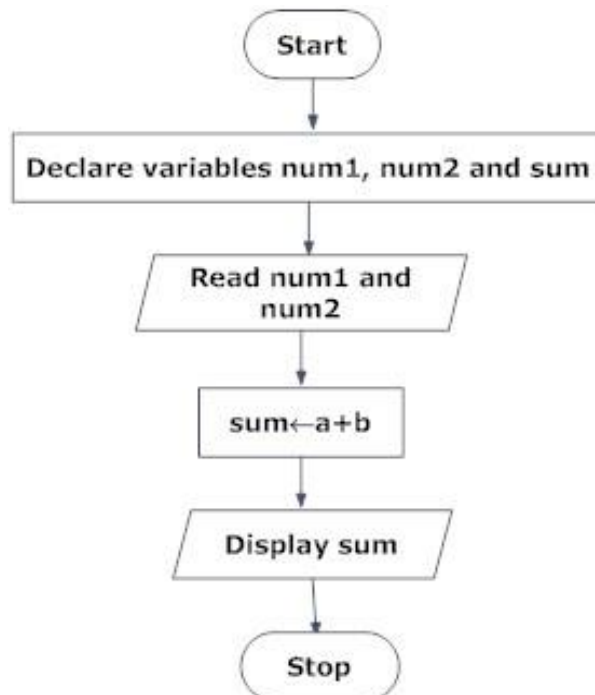
1.2.2.2 Loops



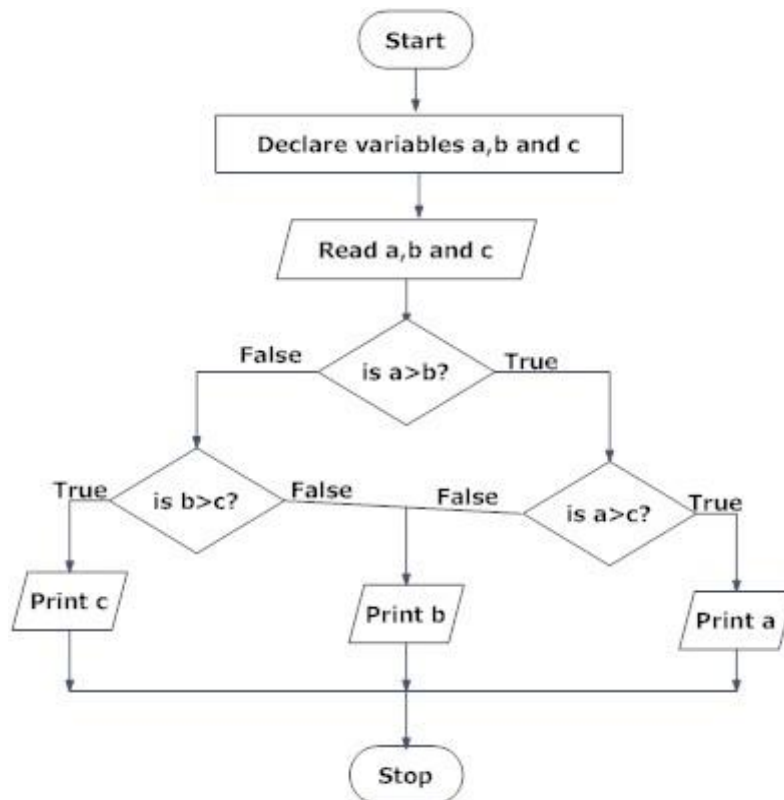


1.2.3 Examples of flowcharts in programming

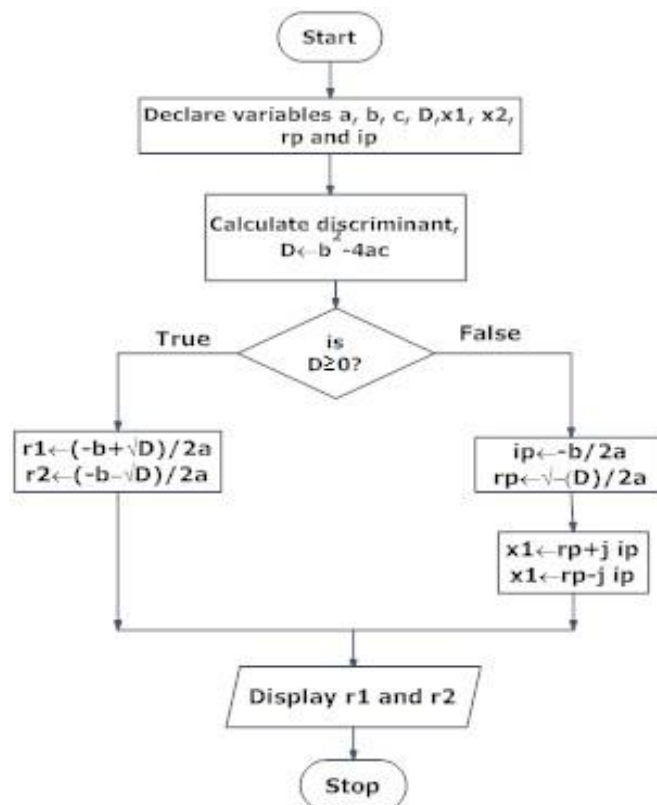
- Draw a flowchart to add two numbers entered by user.



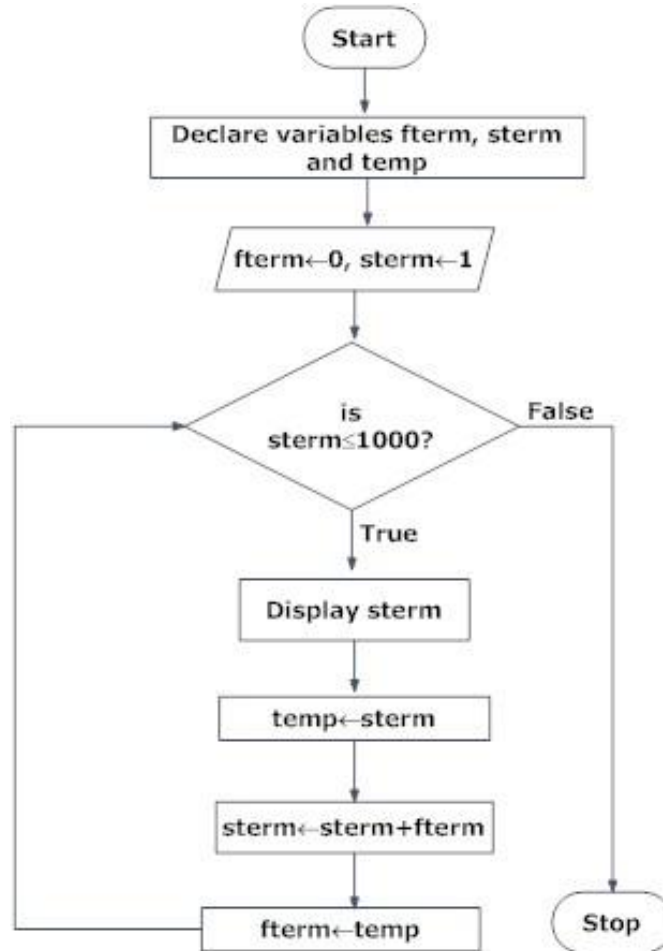
- Draw flowchart to find the largest among three different numbers entered by user.



- Draw a flowchart to find all the roots of a quadratic equation $ax^2+bx+c=0$



- Draw a flowchart to find the Fibonacci series until $\text{term} \leq 1000$.



1.3 Why use C language ?

C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems. The largest measure of C's success seems to be based on purely practical considerations:

- the portability of the compiler;
- the standard library concept ;
- a powerful and varied repertoire of operators;
- an elegant syntax ;
- ready access to the hardware when needed;
- and the ease with which applications can be optimised by hand-coding isolated procedures

C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low-level functions. Most high-level languages provide everything the programmer might want to do already built into the language. A low-level language (e.g. assembler) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably does not supply all the constructs found in high-languages but it provides you with all the building blocks that you will need to produce the results you want!

1.4 Compiler and linker

Some jargon listed for you:

- Source code: the syntax you are writing into a file.
- Compile (build): taking source code and making a program that the computer can understand.
- Executable: the compiled program that the computer can run.
- Header file: files ending by .h, which are included at the start of source code.

The compiler and the linker are necessary to make your source code (.c, .cpp, or .cc files) into an executable (running) program. Generally, compiler refers to both a compiler and a linker.

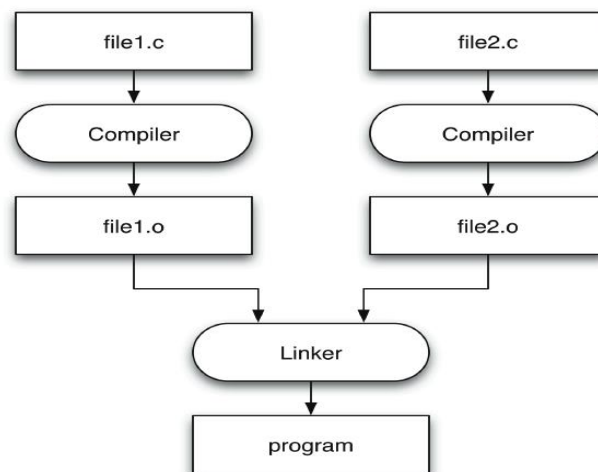


Figure 2: Compile and link process

Figure 2 shows how two C source files, file1.c and file2.c, are turned into an executable named program. The compiler turns file1.c into file1.o and file2.c into file2.o by a process known as **compiling**. The linker then combines file1.o and file2.o into the final **executable** named **program** by a process known as **linking**.

This two-step compile and link process helps scale programs to many source files. It allows developers to split up code into multiple source files in a way that make sense for the project. This not only helps from an organizational standpoint, but also helps speed up compile times. If you change one function, only the source file that contains it needs to be recompiled. All other object files can be re-used when linking the final executable.

1.5 First program in C

The C program is a set of functions. The program execution begins by executing the function **main** (). A function is simply a collection of commands that do "something". The main function is always called when the program first executes. From main function, we can call other functions, whether they are written by us or by others, or use built-in language features. To access the standard functions that comes with your compiler, you need to include a header with the `#include` directive. What this does is effectively take everything in the header and paste it into your program. Let us look at a working program:

```
#include <stdio.h>
void main()
{
    printf( "I am alive!  Beware.\n" );
    getchar();
}
```

Let's look at the elements of the program. The `#include` is a "**preprocessor**" directive that tells the compiler to put code from the header file called `stdio.h` into our program before actually creating the executable. By including header files, you can gain access to many different functions--both the `printf` and `getchar` functions are included in `stdio.h`. The **semicolon** is part of the syntax of C. It tells the compiler that you are at **the end of a statement**. You will see later that the semicolon is used to end most statements in C.

The next important line is `void main()`. This line tells the compiler that there is a function named `main`, and that the function does not return anything. The "**curly braces**" (`{` and `}`) signal the beginning and end of functions and other code blocks.

The `printf` function is the standard C way of displaying output on the screen. The quotes tell the compiler that you want to output the literal string as-is (almost). The `'\n'` sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail); for the time being, just remember that there are a few sequences that, when they appear in a string literal, are actually not displayed literally by `printf` and that `'\n'` is one of them. The actual effect of `'\n'` is to move the cursor on your screen to the next line. Again, notice the semicolon: it is added onto the end of all lines, such as function calls, in C.

The next statement is `getchar()`. This is another function call: it reads in a single character and waits for the user to hit enter before reading the character. This line is included because many compiler environments will open a new console window, run the program, and then close the window before you can see the output. This command keeps that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

The final brace closes off the function. You should try compiling this program and running it. You can cut and paste the code into a file, save it as a `.c` file, and then compile it. Compiling and running should be as simple as clicking a button with your mouse (perhaps the "build" or "run" button) if you use a software for this purpose. You might start playing around with the `printf` function and get used to writing simple C programs.

Fundamental elements of a C program

There are two main elements of the C program: data and instructions (or statements). Data can be represented as variables or constants. Instructions are consisting of operands and operators. Operands can be a constant or a variable. Operators can be arithmetic and relational. In this section, we will describe all those items.

2.1 Comments

Comments are parts of the disregarded treatment. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the program. There are two ways to insert comments :

```
// line comment
/* block comment */
```

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, known as block comment, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line. We are going to add comments in our programs later in the document.

2.2 Variables

A variable is a storage location (memory location) and has an associated type. In a program, we may need to store temporarily some values (data). Values are stored in variables (memory locations). In other terms, a variable is a way of referring to a memory location used in a computer program. This memory location holds values : numbers, text or more complicated types (Figure 3).

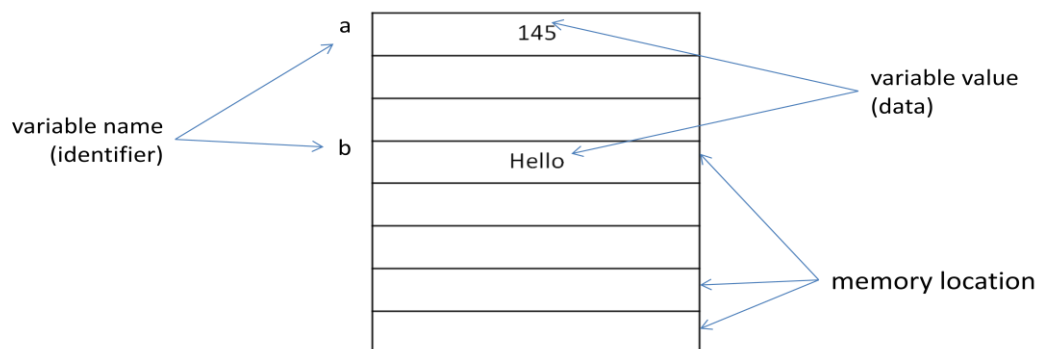


Figure 3: variables in the memory

In order to manipulate data, we must first define what it is that we are manipulating. Generally, programmers tend to talk about the *type* and the *value* of data.

A data *type* is the specific "kind of thing" that the data is. It could be a number, or a string (sequence of characters). The *type* constrains the possible *values* of the data, by defining what values are "valid".

A data *value* is, well, the actual value of the data. An element of data may be any value that the *type* of the data allows. For example, 5 and -7.2 are valid values for numeric data.

Note that the variable type determines not only its size but also the operations we can make on this variable. For example, we can add two numerical values, but we can't make an addition of two texts.

Therefore, we can define a variable as a portion of memory to store a determined value. A variable has three main characteristics: **identifier**, **type** and **value**.

2.2.1 Identifier

Each variable needs an identifier that distinguishes it from the others. A valid identifier is a sequence of one or more letters, digits or underscores characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underscore character (_). In no case can they begin with a digit.

Examples of valid identifiers:

HI H1 abc x A_B MyName x1 Y_100

Examples of **invalid** identifiers :

X Y ?Y 1W X!Y a3# A' XY" a1!3

Very important: Most programming languages are "case sensitive" languages. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. **That is the case of C language.** Thus, for example, the *RESULT* variable is not the same as the *result* variable or the *Result* variable. These are three different variable identifiers. Therefore, It is better to go native with this constraint during writing programs.

2.2.2 Keywords

C keeps a small set of keywords for its own use. These keywords **cannot** be used as identifiers in the program — a common restriction with modern languages. Here is the list of keywords used in Standard C; you will notice that none of them uses upper-case letters.

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

2.2.3 Values and standard data types

In this section, we will describe the main useful types in C language.

2.2.3.1 int

A variable is of type int if its values are integers.

Examples: -100 -5 0 1 13 199 ... are values of type int.

2.2.3.2 float

A variable is of type float if its values are real (floating point values). The float data type is used to store fractional numbers (real numbers) with 6 digits of precision.

Examples: -1.43 -0.713 1.9 27.394

2.2.3.3 double

When the accuracy of the floating point number is insufficient, we can use the double to define the number. The double is same as float but with longer precision.

2.2.3.4 char

A variable is of type char if its values is only **one** character.

Examples: 'A' '1' 'R' '4' " (empty char).

Notes: Use single quote to assign a value to a variable of type char. All mathematical operators may be USED with character variables because each character is represented as an ASCII code in the range 0 to 255 as listed in Table 2 .

Table 2. code ASCII table

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

The digits at the left of the table are the left digits of the decimal equivalent (0-127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for "F" is 70, and the character code for "&" is 38.

Table 3: Data Types and Storage Size in C Language

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 bytes	-32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
short	2 bytes	-32,768 to 32,767
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	1.2E-38 to 3.4E+38 (6 decimal places)
double	8 bytes	2.3E-308 to 1.7E+308 (15 decimal places)
Long double	10 bytes	3.4E-4932 to 1.1E+4932 (19 decimal places)

2.2.4 Declaration of variables

In order to use a variable, we **must first declare** it specifying which data type we want it to be. The way to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. The basic form for declaring variables is:

```
data_type    var_id;
```

For example :

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type `int` with the identifier `a`. The second one declares a variable of type `float` with the identifier `mynumber`.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas:

```
data_type    var_id1, var_id2, var_id3, ...;
```

For example:

```
int a,b,c ;
```

This declares three variables (`a`, `b` and `c`), all of them of type `int`, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

The declaration of three variables reserved three locations in the memory called `a`, `b` and `c` (see Figure 4). The values of the variables are until now undetermined.

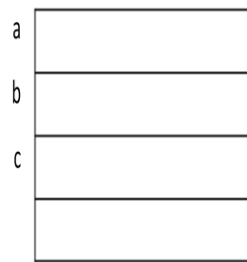


Figure 4: reservation of three locations in the memory

IMPORTANT: the declaration statements must be at the head of the program. Therefore, we must declare all variables before beginning the treatment.

2.2.5 Initialization of variables

When declaring a variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable by appending an equal sign followed by the value to which the variable will be initialized:

```
type var_id = initial_value ;
```

For example, if we want to declare an int variable called an initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a =0;
```

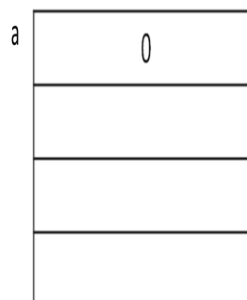


Figure 5 : variable with initial value

2.3 Constants

Constants are expressions with a **fixed** value (you cannot modify its value in the program).

2.3.1 Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the `#define` preprocessor directive (at the top of the source file). Its format is:

```
#define identifier value
```

Example :

```
#include <stdio.h>
#define PI 3.14159
#define C ':'
void main()
{
    int r = 20;
    float area = PI*r*r;
    char v = C; //the variable v gets the value ':'
    printf( "The area of the circle %c %d", C, PI );
    getchar();
}
```

In fact, the only thing that the compiler preprocessor does, when it encounters `#define` directives, is to literally replace any occurrence of their identifier (in the previous example, these were *PI* and *C*) by the code to which they have been defined (3.14159 and ':' respectively).

The `#define` directive is not a C statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and **does not require** a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences of the identifier within the body of the program that the preprocessor replaces.

2.3.2 Declared constants (const)

With the `const` prefix you can declare constants with a specific type in the same way as you would do with a variable. The previous example will be :

```
#include <stdio.h>

void main()
{
    const float PI = 3.14159;
    const char C = ':';
    int r = 20;
    float area = PI*r*r;
    char v = C; //tha variable v gets the value ':'
    printf( "The area of the circle %c %d", C, PI );
    getchar();
}
```

2.4 Operands, operators and expressions

An operand is an entity on which an operator acts. An expression is a sequence of operators and operands that computes a value. Operands include constants, variables, and complex expressions formed by combining operands with operators or by enclosing operands in parentheses. Most expressions have a value based on their contents (values of the variables). Example of expressions:

```
a + b
a - c*0.35
s + (a-b)*(c-a)
a + c/4
a - (c*0.35 + b/8) (2*a + c)
```

A statement (or instruction) is just an expression terminated with a **semicolon**. For example:

```
sum = x + y + z ;
```

Multiple statements can be on the same line. White space is ignored. Statements can continue over many lines. For example :

```
sum = x + y; printf("Hello!");    q=j*2;
```

are the same of :

```
sum = x + y;
printf("Hello!");
q=j*2;
```

2.4.1 The Assignment operator (=)

The assignment operator is the equal sign = (called *gets*) and is used to give a variable the value of an expression. An assignment operator assigns a value to its left operand based on the value of its right operand. The assignment operator has right-to-left associativity (the order in which the operands are evaluated). For example :

```
i=0;
x=34.8;
sum=a+b;
myChar='J';
j=j+3;
```

When used in this manner, the equal sign should be read as "*gets*". Note that all variables must be declared before use and when assigning a character value the character should be enclosed in single quotes.

2.4.2 Arithmetic Operators

2.4.2.1 Multiplicative operators

The multiplicative operators take operands of arithmetic types. These operators are :

- Multiplication (*)

The multiplication operator (*) yields the result of multiplying the first operand by the second. It has left-to-right associativity (the order in which the operands are evaluated).

```
operand1 * operand2
```

- Division (/)

The division operator (/) yields the result of dividing the first operand by the second. It has left-to-right associativity (the order in which the operands are evaluated).

```
operand1 / operand2
```

The division operator is used to perform **integer division if and only if both numerator and denominator are integers**. the resulting integer is obtained by discarding (or truncating) the fractional part of the actual floating point value. For example :

```
(1/2)      // evaluates to 0
(3/2)      // evaluates to 1
(9/4)      // evaluates to 2
(12 /30)   // evaluates to 0
```

- Modulus (remainder from division) (%)

The modulus operator (%) has a stricter requirement in that its operands must be of integral type. It has left-to-right associativity (the order in which the operands are evaluated).

```
operand1 % operand2
```

The above expression is read as "*operand1 modulus operand*" and evaluates to the **remainder** obtained after dividing operand1 by operand2. For example :

```
(7 % 2)      // evaluates to 1
(12 % 3)     // evaluates to 0
```

Note that division by 0 in either a division or a modulus expression is undefined and causes an error.

2.4.2.2 Additive operators

The additive operators take operands of arithmetic. These operators are:

- Addition (+)

The result of the addition (+) operator is the sum of the operands. It has left-to-right associativity (the order in which the operands are evaluated).

```
operand1 + operand2
```

- Subtraction (-)

The result of the subtraction (-) operator is the difference between the operands. It has left-to-right associativity (the order in which the operands are evaluated).

operand1 - operand2

2.4.3 Compound assignment (+=, -=, *=, /=, %=)

The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and store the result of that operation into the left operand, **which must be a modifiable value**. Table 4 lists the compound assignment operators:

Table 4: the compound assignment operators

Operator	Example	Equivalent expression
+=	index += 2	index = index + 2
-=	index -= 2	index = index - 2
*=	bonus *= increase	bonus = bonus * increase
/=	time /= hours	time = time / hours
%=	allowance %= 1000	allowance = allowance % 1000

2.4.4 Increase and decrease (++ , --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus :

```
c++;
c+=1;
c=c+1;
++c;
```

are all equivalent in its functionality: the three of them increase by one the value of c.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning. In the case that the increase operator is used as a **prefix** (++a), the value is increased **before** the result of the expression is evaluated. Therefore, the increased value is considered in the outer expression; in case that it is used as a **suffix** (a++), the value stored in a is increased after being evaluated. Therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1	Example 2
<pre>B=3; A=++B; //A contains 4, B contains 4</pre>	<pre>B=3; A=B++; // A contains 3, B contains 4</pre>

2.4.5 Relational and equality operators (comparison operators)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a boolean value that can **only** be 1 (i.e. true) or 0 (i.e. false), according to its boolean result. We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other.

2.4.5.1 Equality operators

The binary equality operators compare their operands for strict equality or inequality. The result type for these operators is boolean.

- The equal-to operator (==)

It returns 1 if both operands have the same value; otherwise, it returns 0.

```
operand1 == operand2
```

- The not-equal-to operator (!=)

It returns 1 if the operands do not have the same value; otherwise, it returns 0.

```
operand1 != operand2
```

The equality operators have left-to-right associativity.

2.4.5.2 Relational operators

The binary relational operators determine the following relationships:

- Less than (<)

```
operand1 < operand2
```

- Greater than (>)

```
operand1 > operand2
```

- Less than or equal to (<=)

```
operand1 <= operand2
```

- Greater than or equal to (>=)

```
operand1 >= operand2
```

The relational operators have left-to-right associativity. Both operands of relational operators must be of arithmetic. They yield values of type bool. The value returned is 0 if the relationship in the expression is false; otherwise, the value returned is 1.

Here are some examples :

```
(7 == 5)    // evaluates to 0
(5 > 4)     // evaluates to 1
(3 != 2)    // evaluates to 1
(6 >= 6)    // evaluates to 1
```

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that $a=2$, $b=3$ and $c=6$.

```
int a=2, b=3, c=6;
(a == 5)      // evaluates to 0 since a is not equal to 5
(a*b >= c)    // evaluates to 1 since (2*3 >= 6) is true
(b+4 > a*c)   // evaluates to 0 since (3+4 > 2*6) is false
((b=2) == a) // evaluates to 1.
```

Be careful! The operator $=$ (one equal sign) is not the same as the operator $==$ (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one ($==$) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression $((b=2) == a)$, we first assigned the value 2 to b and then we compared it to a , that also stores the value 2, so the result of the operation is 1.

2.4.6 Logical operators

The logical operators, logical AND ($\&\&$) and logical OR ($\|\|$), are used to combine multiple conditions formed using relational or equality expressions.

- the logical operator AND ($\&\&$)

The logical AND operator ($\&\&$) returns the boolean value **1** if both operands are evaluated to 1 (i.e. true) and returns **0** otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical AND has left-to-right associativity.

operand1 $\&\&$ operand2

$\&\&$ OPERATOR

a	b	a $\&\&$ b
1	1	1
1	0	0
0	1	0
0	0	0

- The logical OR operator ($\|\|$)

The logical OR operator ($\|\|$) returns the boolean value **1** if either or both operands is **1** and returns **0** otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical OR has left-to-right associativity.


```
operand1 || operand2
```

OPERATOR		
a	b	a b
1	1	1
1	0	1
0	1	1
0	0	0

For example:

```
( (5 == 5) && (3 > 6) ) // evaluates to 0 ( 1 && 0 )
( (5 == 5) || (3 > 6) ) // evaluates to 1 ( 1 || 0 )
```

2.4.7 Conditional operator (?)

The conditional operator ? evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

```
condition ? result1 : result2
```

If condition is true, the expression will return result1, if it is not it will return result2. Here are some examples :

```
b = ((7==5+2) ? 4 : 3) // b gets 4, since 7 is equal to 5+2.
d = ((a>b) ? a : b)    // d gets whichever is greater, a or b.
```

2.4.8 Evaluation and operator precedence

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this statement :

```
a = 5 + 7 % 2
```

We may doubt if it really means:

```
a = 5 + (7 % 2) // with a result of 6, or
a = (5 + 7) % 2 // with a result of 0
```

The precedence of operators determines the order in which operations are performed in an expression. Operators with **higher precedence** are employed first. If two operators in an expression have the same precedence, **associativity** determines the direction in which the expression will be evaluated.

Operator	Associativity
() []	left-to-right
++ --	right-to-left
* / %	left-to-right
+ -	
< > <= >=	
== !=	
&&	
? :	right-to-left
= *= /= %= += -=	

Table 5: Operator precedence chart

Table 5 summarizes the precedence and associativity (the order in which the operands are evaluated) of the operators, listing them in order of precedence from **highest to lowest**. Where several operators appear together, they have equal precedence and are evaluated according to their associativity. All these precedence levels for operators can be manipulated by removing possible ambiguities using parentheses signs (and), as in this example:

```
a = 5 + 7 % 2;
```

It might be written either as:

```
a = 5 + (7 % 2);
```

Or

```
a = (5 + 7) % 2;
```

It is depending on the operation that we want to perform. So if you want to write complicated expressions and you are not completely sure of the precedence levels, always include **parentheses**. It will also make your code easier to read.

3 Input/output statements

Using the input/output statements, we will be able to interact with the user by printing (displaying) messages on the screen and getting the user's input from the keyboard. Using variables in C for input or output can be a bit of a hassle at first, but bear with it and it will make sense.

3.1 Output statement (printf function)

The general forms of the output statement is by callin the printf function:

```
printf("control string",arguments list);
```

where the control string consists of

- literal text to be displayed,
- format specifiers,
- special characters.

surrounded by double quotes "".

Format specifier	Data type
%c	character
%d	integer
%f	float
%lf	double

Table 6: Format specifiers for each data type

The arguments list can be variables, constants, expressions, or function calls seperated by commas. **Number of arguments must match the number of format specifiers.** Table 6 shows what format specifiers should be used with what data types.

Table 7 shows some common special characters for cursor control used in printf function.

Special characters	Output on screen
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\\	backslash (\)

Table 7: Some special characters used in printf function

Example:

```
#include <stdio.h>
void main()
{
    int a,b,c ;
    a=8;
    b=6;
    c=b*4;
    printf("The value of b = %d \n", b);
    printf("The value of c = \n");
    printf("%d",c);
    printf("\n");
    printf("Value of a =  %d Value of b = %d Value of c =  %d \n",a,b,c);
    printf ("The end..");
}
```

Figure 6 shows the outputs on the screen using printf function and the value of each variable in the memory.

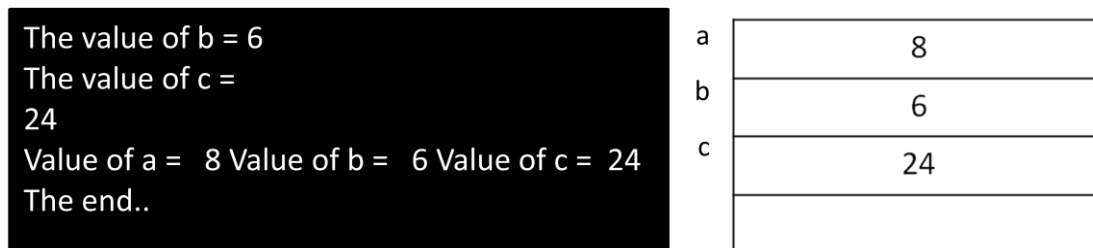


Figure 6 : the output on the screen and the variables' values in the memory

3.2 Input statement (scanf function)

The general form of the statement:

```
scanf ("format specifiers", &variable1, &variable2, ...);
```

This statement gets (reads) the user's input and stores it in the listing variables: variable1, variable2, ... one after another. The format specifiers surrounded by double quotes "" are the same shown in Table 6. **You must use the symbol & (called reference) before each variable.**

For example :

```
int a,b,c ;
scanf ("%d", &a);      // suppose the user fills 3
scanf ("%d%d", &b, &c); // suppose the user fills 5 then 7
```

Figure 7 shows the values of the variable a, b and c in the memory.

a	3
b	5
c	7

Figure 7: variables' values in the memory after filling

4 Control Structures

A program is usually not limited to a linear sequence of instructions. During its process, it may bifurcate, repeat code or take decisions. For that purpose, control structures were provided.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements that are separated by semicolons (;) like all statements, but grouped together in a block enclosed in braces { }. Its general structure is:

```
{
    statement1;
    statement2;
    statement3; ...
}
```

A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({ }). However, in the case that we want the statement to be a compound statement it must be enclosed between braces ({ }), forming a block.

4.1 Conditional structure: if, else and switch

The **if** keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

```
if (condition)
statement
```

where `condition` is the expression that is being evaluated. If `condition` is true (evaluated to 1), `statement` is executed. If it is false (evaluated to 0), `statement` is ignored (not executed) and the program continues right after this conditional structure.

For example, the following programs fragment prints `x is 100` only if the value stored in the `x` variable is indeed 100:

```
if (x == 100)
    printf("x is 100");
```

If we want more than a single statement to execute in case that the condition is true (evaluated to 1) we can specify a block using braces { }:

```

if (x == 100)
{
    printf("x is ");
    printf("%d",x);
}

```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword **else**. Its form used in conjunction with if is:

```

if (condition)
    statement1
else
    statement2

```

For example:

```

if (x == 100)
    printf ("x is 100");
else
    printf ("x is not 100");

```

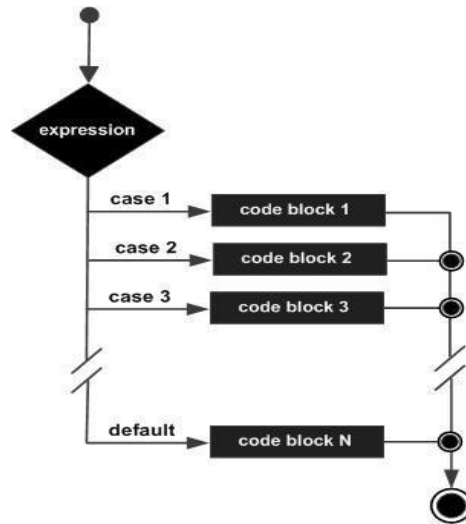
This example prints on the screen x is 100 if indeed x has a value of 100, but if it has not - and only if not- it prints out x is not 100.

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**. The syntax for a switch statement in C programming language is as follows:

```

switch(expression){
case constant-expression :
    statement(s);
    break; /* optional */
case constant-expression :
    statement(s);
    break; /* optional */
/* you can have any number of case statements */
default : /* Optional */
    statement(s);
}

```



The following rules apply to a switch statement:

- The expression used in a switch statement must have an integral.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

4.2 Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

4.2.1 The while loop

Its format is:

```
while (condition)
    statement
```

Its functionality is simply to repeat `statement` (loop body) while `condition` is true. The *while* statement works as follows:

1. `condition` is evaluated (named entry condition)
2. If it is FALSE, skip over the loop
3. If it is TRUE, `statement` is executed
4. Go back to step 1

Example :

```
int i =1;
while ( i < 7 )
{
    printf("%d\n",i) ;
    i = i+1;
}
```

Here is the output on the screen :

```
1
2
3
4
5
6
```

IMPORTANT : When creating a while-loop, we must always consider that it has to end at some point; therefore, we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever (infinite loop).

4.2.2 The do-while loop

Its format is:

```
do
    statement
while (condition);
```

Its functionality is exactly the same as the while loop, except that `condition` in the *do-while* loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if `condition` is never fulfilled.

It works as follows :

1. The statement of the loop is executed.
2. The condition is evaluated.
3. If it is TRUE, go back to step 1. If it is FALSE, exit loop.

4.2.3 The for loop

Its format is :

```
for (initialization; condition; increase)
    statement
```

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the *for* loop provides specific locations to contain an initialization statement and an increase statement. So, this loop is specially designed to perform a repetitive action with a **counter** which is initialized and increased on each iteration.

It works in the following way (see Figure 8):

1. Initialization is executed. Generally it is an initial value setting for a counter variable. **This is executed only once.**
2. Condition is checked. If it is *true* the loop continues (to step 3), otherwise the loop ends and statement is skipped (step 5).
3. Statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. Finally, whatever is specified in the increase field is executed and the loop gets back to step 2.
5. End of the *for* loop

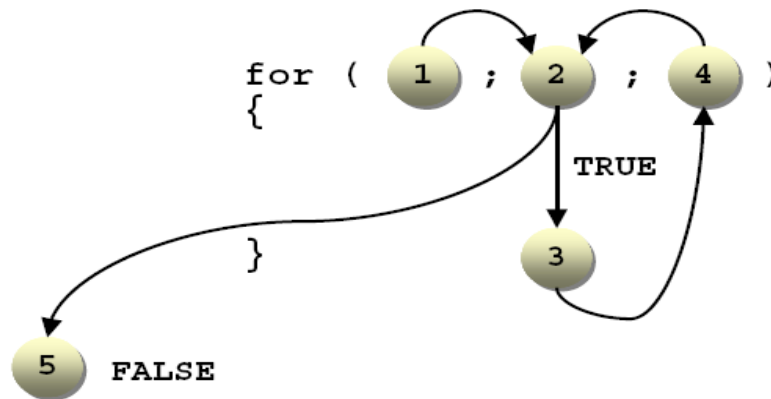


Figure 8: diagram illustrates the operation of a for loop

Example :

```

int i;
for (i=1; i<7; i=i+1)
    printf("%d\n",i) ;

```

Here is the output on the screen which is the same output of the above example using *while* loop :

```

1
2
3
4
5
6

```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

4.3 Jump statements

4.3.1 The break statement

Using *break*, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For Example :

```
int i;
for (i=1; i<7; i=i+1)
{
    if (i == 5) break;
    printf("%d\n",i) ;
}
printf("Out of for loop");
```

Here is the output on the screen:

```
1
2
3
4
Out of for loop
```

4.3.2 The continue statement

The *continue* statement causes the program to skip the **rest** of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.

Example:

```
int i;
for (i=1; i<7; i=i+1)
{
    if (i == 5) continue;
    printf("%d\n",i) ;
}
printf ("Out of for loop");
```

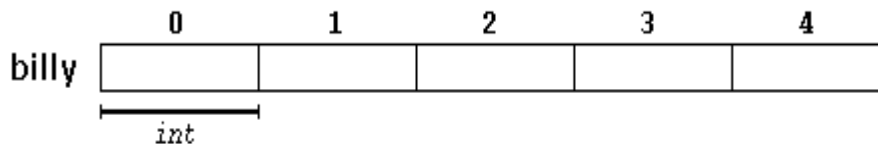
Here is the output on the screen (the number 5 was not displayed):

```
1
2
3
4
6
Out of for loop
```

5 Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, `int` for example, with a unique identifier.

For example, an array to contain 5 integer values of type `int` called `billy` could be represented like this :



where each blank panel represents an element of the array, that in this case are integer values of type `int`. These elements are numbered from 0 to 4 since in arrays **the first index is always 0**, independently of its length (size).

5.1 Array declaration

Like a regular variable, an array must be declared before it is used. A typical declaration for an array is:

```
type array_name [size];
```

where `type` is a valid type (like `int`, `float`...), `array_name` is a valid identifier and the `size` field (which is always a positive integer enclosed in square brackets `[]`), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called `billy` as the one shown above it is as simple as:

```
int billy [5];
```

NOTE: The `size` field within brackets `[]` which represents the number of elements the array is going to hold, must be a **positive constant** value.

5.2 Initializing Array

You can initialize an array in C either one by one or using a single statement as follows:

```
double billy[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces `{ }` cannot be larger than the number of elements that we declare for the array between square brackets `[]`.

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double billy[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

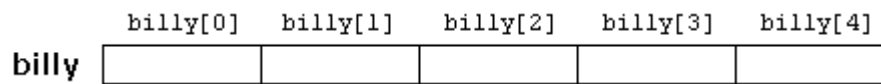
You will create exactly the same array as you did in the previous example.

5.3 Accessing the values of an array

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as :

```
array_name [index]
```

Following the previous examples in which *billy* had 5 elements and each of those elements was of type *int*, the name which we can use to refer to each element is the following:



For example, to store the value 75 in the third element of *billy*, we could write the following statement:

```
billy[2] = 75;
```

and, for example, to pass the value of the third element of *billy* to a variable called *a*, we could write:

```
a = billy[2];
```

Therefore, the expression `billy[2]` is for all purposes like a variable of type *int*.

Notice that the third element of *billy* is specified `billy[2]`, since the first one is `billy[0]`, the second one is `billy[1]`, and therefore, the third one is `billy[2]`. By this same reason, its last element is `billy[4]`.

At this point it is important to be able to clearly distinguish between the two uses that brackets `[]` have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets `[]` with arrays.

```
int billy[5];           // declaration of a new array
billy[2] = 75;         // access an element of the array
```

If you read carefully, you will see that a data type always precedes a variable or array declaration, while it never precedes an access.

Some other valid operations with arrays:

```
billy[0] = a;
billy[i] = 75;
b = billy[i+2];
billy[i+1] = billy[i/2] + 5;
billy[i*2] = billy[i] + 1;
billy[1] = 2*billy[i-2] / 3;
```

5.4 Bidimensional array (two-dimensional array)

A bidimensional array can be imagined as a bidimensional table (matrix) made of elements, all of them of a same uniform data type. A typical declaration for a bidimensional array is:

```
type array_name [HEIGHT][WIDTH];
```

where `HEIGHT` is the number of lines (rows) and `WIDTH` is the number of columns.

For example, `jimmy` represents a bidimensional array of 3 per 5 elements of type `int`. The way to declare this array in C would be:

```
int jimmy [3][5];
```

		0	1	2	3	4
jimmy {	0					
	1					
	2					

and, for example, the way to reference (access) the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```

		0	1	2	3	4
jimmy {	0					
	1					
	2					

↓
jimmy[1][3]

Some other valid operations with bidimensional arrays:

```
jimmy[0][4] = a;
jimmy[i][j] = 75;
b = jimmy[i+2][j+1];
jimmy[i+1][j] = jimmy[i][j] + 5;
jimmy[i*2][j/2] = jimmy[i][j] + 1;
jimmy[1][1] = 2*jimmy[i-1][j-2] / 3;
```

Remember that array indices always begin by **zero**.

6 Strings

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus, a null-terminated string contains the characters that comprise the string followed by a null. The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above-defined string in C:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above-mentioned string:

```
#include <stdio.h>
int main ()
{
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Greeting message: Hello

6.1 Scanf , printf , gets and puts functions with strings

A string can be printed by using various functions such as printf, puts. To input a string we use scanf or gets function. **Note** that scanf can only be used to receive strings without any spaces as input from a user, to receive strings containing spaces use gets function. Here are some examples.

Example I:

```
#include <stdio.h>

int main()
{
    char array[20] = "C programming language";
    printf("%s\n", array);
    return 0;
}
```

Example II:

```
#include <stdio.h>

int main()
{
    char array[100];
    printf("Enter a string\n");
    scanf("%s", array);
    printf("You entered the string %s\n", array);
    return 0;
}
```

Example III:

```
#include <stdio.h>

int main()
{
    char a[80];
    gets(a);
    printf("%s\n", a);
    return 0;
}
```

Example IV:

We can print a string using a for loop by printing individual characters of the string.

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

int main() {
    char s[100];
    int c, l;
    gets(s);
    l = strlen(s);
    for (c = 0; c < l; c++)
        printf("%c", s[c]);
    return 0;
}
```

6.2 Functions used with strings

In the library <string.h>, C supports a wide range of functions that manipulate null-terminated strings:

- **strcpy**(s1, s2): Copies string s2 into string s1.
- **strcat**(s1, s2): Concatenates string s2 onto the end of string s1.
- **strlen**(s1): Returns the length of string s1.
- **strcmp**(s1, s2): Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
- **strchr**(s1, ch): Returns a pointer to the first occurrence of character ch in string s1.
- **strstr**(s1, s2): Returns a pointer to the first occurrence of string s2 in string s1.

In order to use these functions, you should include the C-string library using the following directive:

```
#include <string.h>
```


7 Functions

7.1 Why functions ?

The most important question is why do we need a function? Functions have many uses. For example, a programmer may have a block of code that he has repeated forty times throughout the program. A function to execute that code would save a great deal of space, and it would also make the program more readable. Also, having only one copy of the code makes it easier to make changes. Would you rather make forty little changes scattered all throughout a potentially large program, or one change to the function body? So would I.

Another reason for functions is to break down a complex program into logical parts. For example, take a menu program that runs complex code when a menu choice is selected. The program would probably best be served by making functions for each of the actual menu choices, and then breaking down the complex tasks into smaller, more manageable tasks, which could be in their own functions. In this way, a program can be designed that makes sense when read. In addition, it has a structure that is easier to understand quickly. The worst programs usually only have the required function, `main`, and fill it with pages of jumbled code.

7.2 Functions in C

Every C program consists of one or more functions. A function is a group or sequence of C statements that are executed together. Each C program function performs a specific task. The `main()` function is the most important function and **must** be present in every C program. The execution of a C program begins in the `main()` function. The code below shows the **general structure** of a C program that will be explained later in more detail in the document :

```
#include <stdio.h>
//prototype of each function
return-type function_name1 ( arg_type arg1, ..., arg_type argN );
return-type function_name2 ( arg_type arg1, ..., arg_type argN );
// the main function
void main()
{ //statements body of the main function
}
//definition of each function
return-type function_name1 ( arg_type arg1, ..., arg_type argN )
{
// statements - body of the function
}
return-type function_name2 ( arg_type arg1, ..., arg_type argN )
{
// statements - body of the function
}
```

Now that you should have learned about variables, loops, and conditional statements it is time to learn about functions. You should have an idea of their uses as we have already used them and defined one in the guise of `main()`. `getchar()` is another example of a function. In general, functions are **blocks** of code (statements) that perform a number of pre-defined commands to accomplish something productive. You can either use the built-in library (e.g. `printf`, `scanf`, etc..) functions or you can create your own functions. **In fact, a C program is a set of functions including the main function.**

7.3 Function prototype

Functions that a programmer writes will generally require a prototype. Just like a blueprint, the prototype gives basic structural information: it tells the compiler what the function will return, what the function will be named, as well as what arguments the function can be passed. Function names follow the same rule as variable names.

The general format for a prototype is simple:

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

`arg_type` just means the type for each argument -- for instance, an `int`, a `float`, or a `char`. It's exactly the same thing as what you would put if you were declaring a variable. There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value. Functions that **do not** return values have a return type of **void**. Let's look at a function prototype:

```
int mult (int x, int y);
```

This prototype specifies that the function `mult` will accept two arguments, both integers, and that it will return an integer. **Do not** forget the trailing semi-colon. Without it, the compiler will probably think that you are trying to write the actual **definition** (implementation) of the function.

7.4 Definition of the function

When the programmer actually defines the function, it will begin with the prototype **minus** the semi-colon (named head of the function). Then there should always be a block (surrounded by curly braces) with the code that the function is to execute, just as you would write it for the main function. Any of the arguments passed to the function can be used as if they were declared in the block. General form for a function **definition** is:

```
return-type function_name ( arg_type arg1, ..., arg_type argN )
{
    // statements - body of the function
}
```

7.5 Function call

Calling (or invoking) a function is using this function in the main (or in another function). General format to call a function :

```
function_name (var1, ..., varN )
```

where `var1`, ..., `varN` are **values or variables** that must match the **types** (not necessarily the names) of the arguments in the function prototype `arg_type arg1`, ..., `arg_type argN`. For example :

```
int u = 2;
int a = mult (u,12); //call the function mult, then a gets 24
int b = mult (a,u); //call the function mult, then b gets 48
multi(a,b); //call the function mult BUT the returned value is lost
```

IMPORTANT : For a given function that returns such value, we should keep this value somewhere (usually in a variable). Otherwise, when calling the function, the returned value is lost.

Let's look at an example program:

```
#include <stdio.h>
int mult ( int x, int y ); //prototype of the function
void main()
{
    int x;
    int y;
    int z;
    printf( "Please input two numbers to be multiplied: " );
    scanf( "%d", &x );
    scanf( "%d", &y );
    //Call the function mult in printf
    printf( "The product of your two numbers is %d\n", mult( x, y ) );
    //Call the function mult and assign the return value to the variable z
    z = mult(x,y);
    printf( "The product of your two numbers is %d\n", z );
    getchar();
}
// function definition
int mult (int x, int y)
{
    return x * y;
}
```

This program begins with the only necessary include file. Next is the prototype of the function (it has the final semi-colon). In the main (), the user fills two integer values. Notice how printf actually takes the value of what appears to be the mult function. What is really happening is printf is accepting the value **returned** by mult. The result would be the same as if we assign the returned value of mult to a variable z and then display it via printf.

The mult function is actually defined below main. Because its prototype is above main (), the compiler still recognizes it as being declared, and so the compiler will not give an error about mult being undeclared.

Prototypes are **declarations** of the function, but they are only necessary to alert the compiler about the existence of a function if we don't want to go ahead and fully define the function. If mult were defined before it is used, we could do away with the prototype, the definition basically acts as a prototype as well.

`return` is the keyword used to force the function to return a value **AND** exit of the function. If a function returns void, the return statement is valid, but only if it does not have an expression. In other words, for a function that returns void, the statement `return;` is legal, but usually redundant. (It can be used to exit the function before the end of the function).

7.6 Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function:

- **Call by value:** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- **Call by reference:** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

7.7 Recursive Function in C

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
    recursion(); /* function calls itself */
}

int main() {
    recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

The following example calculates the factorial of a given number using a recursive function.

```
#include <stdio.h>

unsigned int factorial(unsigned int i) {
    if(i <= 1) {
        return 1;
    }

    return i * factorial(i - 1);
}

int main() {
    int i = 12;

    printf("Factorial of %d is %d\n", i, factorial(i));

    return 0;
}
```

8 Scope Rules of variables

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block which is called local variables,
- Outside of all functions which is called global variables.
- In the definition of function parameters which are called formal parameters.

Let us understand what are local and global variables, and formal parameters.

8.1 Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

8.2 Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. A program can have same name for local and global variables but the value of local variable inside a function will take preference.

8.3 Formal Parameters

Formal parameters are treated as local variables with-in a function and they take precedence over global variables.

