Ministry of Higher Education and
Scientific Research

Al-Mustansiriyah University
Faculty of Engineering

Computer Engineering Department

وزارة التعليم العالي والبحث العلمي
الجامعة المستنصرية
كلية الهندسة
قسم هندسة الحاسوب

Data Structure Lab.

Prepared By

Dr. Hanan Ahmed Salman

2016

2017

Experiment No. (1)

Introduction to C++ Basic Programming Language

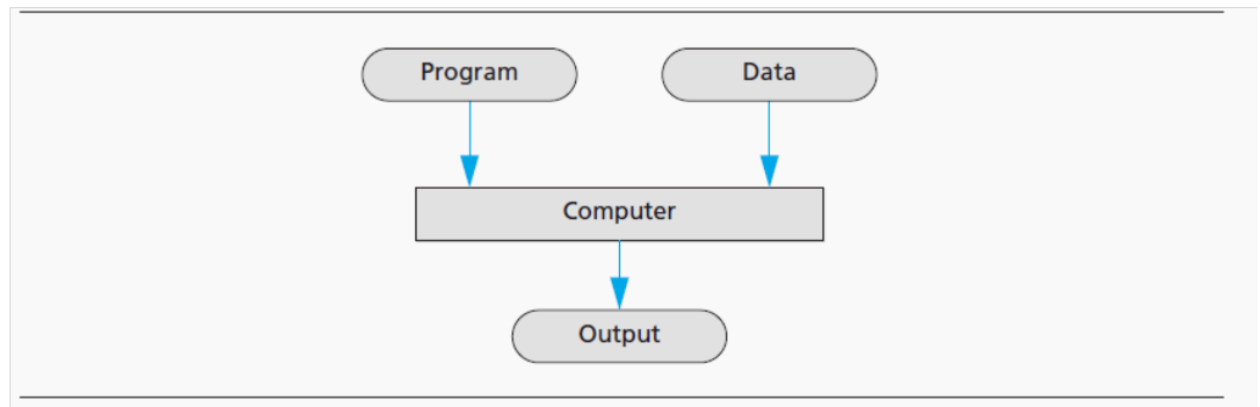
Object:

To give a general introduction to computer programming, programming languages, and simple programs in C++ language.

Theory:

A computer program is a set of instructions that a programmer writes to tell a computer how to carry out a certain task. The instructions, however, must be in a language that the computer understands. Computers only understand a binary language i.e. that composed of 1's and 0's. This is a low-level language and very hard to program in. So, humans invented higher level languages such as C++ or Pascal to make the job easier. As you will see, these languages are nearly like English but you don't have the freedom to write what you like – there are still rules you have to follow.

As shown in below, the input to a computer can be thought of as consisting of two parts, a program and some data. The computer follows the instructions in the program, and in that way, performs some process. The **data** is what we conceptualize as the input to the program. For example, if the program adds two numbers, then the two numbers are the data. In other words, the data is the input to the program, and both the program and the data are input to the computer.



Languages: High, low level and assembly

C++ is a **high-level language**, like most of the other programming languages such as C, Java, Pascal, Visual Basic, FORTRAN, COBOL, Lisp, Scheme, and Ada. **High-level languages** resemble human languages in many ways. They are designed to be easy for human beings to write programs in and to be easy for human beings to read. A high-level language contains instructions that are much more complicated than the simple instructions a computer's processor (CPU) is capable of following.

The kind of language a computer can understand is called a **low-level language.** The exact details of low-level languages differ from one kind of computer to another. A typical low-level instruction might be the following:

ADD X Y Z

This instruction might mean "Add the number in the memory location called X to the number in the memory location called Y, and place the result in the memory location called Z." The above sample instruction is written in what is called **assembly language.** Although assembly

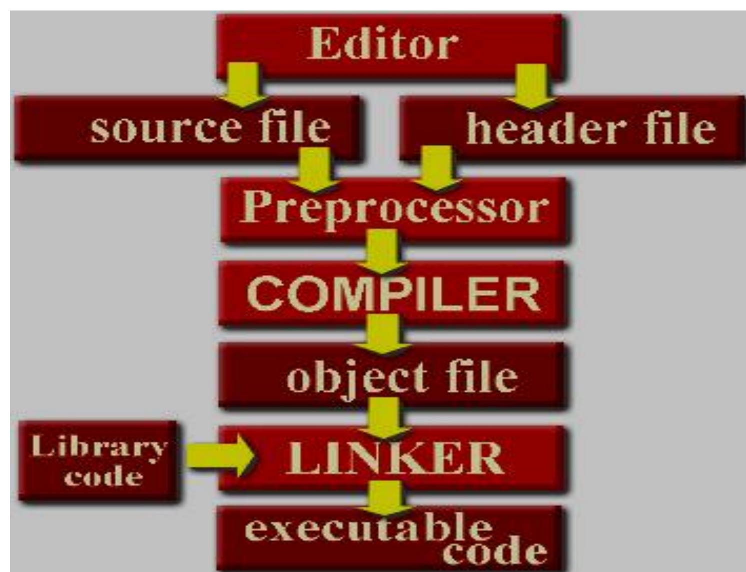
language is almost the same as the language understood by the computer, it must undergo one simple translation before the computer can understand it. In order to get a computer to follow an assembly language instruction, the words need to be translated into strings of zeros and ones. For example, the word ADD might translate to 0110, the X might translate to 1001, the Y to 1010, and the Z to 1011.

The version of the above instruction that the computer ultimately follows would then be:

0110 1001 1010 1011

Assembly language instructions and their translation into zeros and ones differ from machine to machine.

Programs written in the form of zeros and ones are said to be written in **machine language,** because that is the version of the program that the computer (the machine) actually reads and follows.



Editor: accepts the typing of the source code (or header file).

Source file: the file that contains the program you prepared in the editor after you save it. (.cpp)

Header file: header files such as iostream.h

Pre-processor: performs preliminary operations on files before they are passed to the compiler.

Compiler: translates the source code to machine language.

Object code: the file containing the translated source code. (.obj)

Linker: links the object file with additional code, such as the library codes.

Executable code: the file containing the final product. (.exe)

PROGRAMMING AND PROBLEM-SOLVING

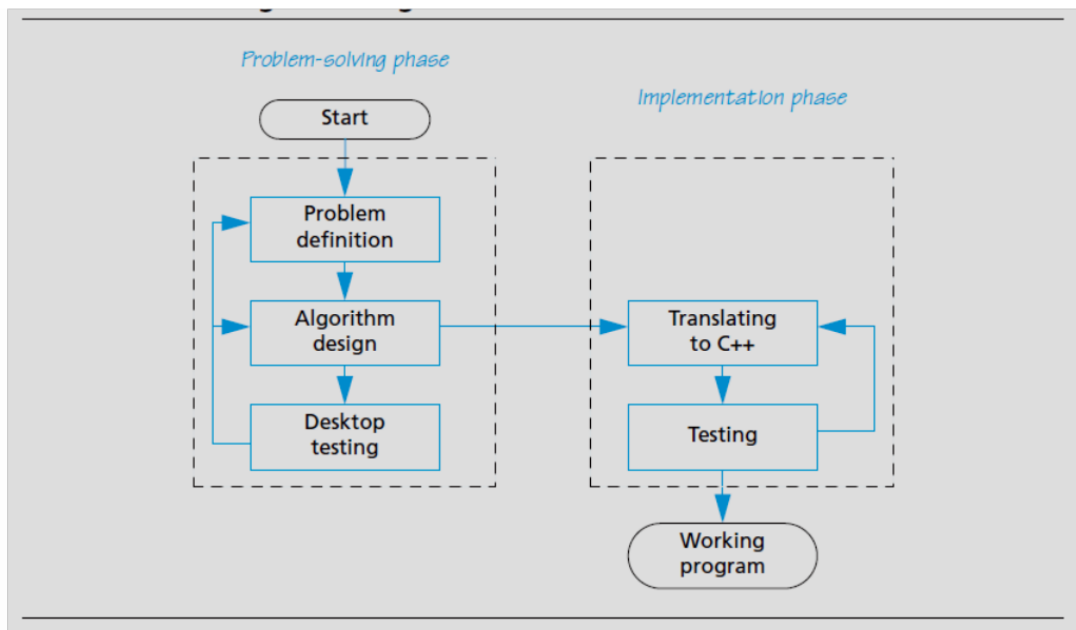
The most difficult part of solving a problem on a computer is discovering the method of solution. After you come up with a method of solution, it is routine to translate your method into the required language, be it C++ or some other programming language.

A sequence of precise instructions which leads to a solution is called an **algorithm.**

A computer program is simply an algorithm expressed in a language that a computer can understand. Thus, the term algorithm is more general than the term program .**Designing a program** is often a difficult task. There is no complete set of rules, no algorithm to tell you how to write programs. Program design is a creative process.

The entire program design process can be divided into two phases, the *problem-solving phase* and the *implementation phase*. The result of the

problem-solving phase is an algorithm, expressed in English, for solving the problem. To produce a program in a programming language such as C++, the algorithm is translated into the programming language. Producing the final program from the algorithm is called the **implementation phase.**



Program 1:

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

Code	Output

`// my first program in C++`
`#include <iostream.h>`
`int main ()`
`{`
`cout << "Hello World!";`
`return 0;`
`}`
	Hello World!

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using.

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream.h>

Lines beginning with a pound sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case, the directive #include <iostream> tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other

functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout << "Hello World";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the **iostream** standard file within the std namespace, so that's why we needed to include that specific file

and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

return 0;

The **return statement** causes the main function to finish. **return** may be followed by a return code (in our example is followed by the return code 0). A return code of **0** for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces ({}) of the main function.

Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code. C++ supports 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.

Program 2

We are going to add comments to our second program:

```
/* my second program in C++
   with more comments */

#include <iostream.h>

int main ()
{
  cout << "Hello World! ";     // prints Hello World!
  cout << "I'm a C++ program"; // prints I'm a C++ program
  return 0;
}
```

```
Hello World! I'm a C++ program
```

If you include comments within the source code of your programs without using the comment characters combinations //, /* or */, the

compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

Questions:

1- Write a C++ program that prints a message "Welcome to the first your lesson".

2- Write a C++ program that prints your name and your class number.

For example:- Ali 1st Class

Experiment No. (2)

Variables and Data Types

Object:

Programming is not limited only to print simple texts on the screen. In order to be able to write programs that perform useful tasks that really save us work, we need to introduce the concept of variables and data types.

Theory:

Let us think that we ask you to retain the number 5 in your mental memory, and then ask you to memorize the number 2 at the same time. You have just stored two different values in your memory. Now, if we ask you to add 1 to the first number, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Now for example, we could subtract and obtain 4 as result. The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a **variable as a portion of memory to store a determined value.** Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were **a**, **b** and result, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

Identifiers

Symbolic names can be used in C++ for various data items used by a programmer. A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set. The rules for the formation of an identifier are:

- An identifier can consist of alphabets, digits and/or underscores.
- It must not start with a digit
- C++ is case sensitive that is upper case and lower case letters are considered different from each other.
- It should not be a reserved word.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language or your compiler's specific ones since they could be confused with these. The standard reserved keywords are:

asm	auto	break	case	catch
char	class	const	continue	default
delete	do	double	else	enum

extern	inline	Int	float	for
friend	goto	If	long	new
operator	private	protected	public	register
return	short	signed	sizeof	static
struct	switch	template	this	Try
typedef	union	unsigned	virtual	void
volatile	while			

When programming, we store the variables in our computer's memory. But the computer has to know what we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A **byte** is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between **0** and **255**). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next, you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2bytes	1 wide character

* The values of the columns Size and Range depend on the architecture of the system where the program is compiled and executed.

Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the desired data type (e.g. **int**, **bool**, **float**...) followed by a valid variable identifier. 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**. Once declared, the variables **a** and **mynumber** can be used within the rest of their scope in the program.

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. 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 integer data types **char**, **short**, **long** and **int** can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specified signed or the specified unsigned before the type name. For example:

```
unsigned short int Number;
signed int MyAccount;
```

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccount;
```

with exactly the same meaning (with or without the keyword signed)

An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable.

short and **long** can be used alone as type of specifies. In this case, they refer to their respective integer fundamental types: **short** is equivalent to **short int** and **long** is equivalent to **long int**. The following two variable declarations are equivalent:

```
short Year;
short int Year;
```

Finally, signed and unsigned may also be used as standalone type specifies, meaning the same as **signed int** and **unsigned int** respectively. The following two declarations are equivalent:

```
unsigned NextYear;
unsigned int NextYear;
```

Program 1

To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory proposed at the beginning of this section:

```
// operating with variables

#include <iostream.h>
int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;}
```

```
4
```

Initialization of variables

When declaring a regular local 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. There are two ways to do this in C++:

The first one, known as **c-like**, is done by appending an equal sign followed by the value to which the variable will be initialized:

type identifier = initial_value ;

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

```
int a = 0;
```

The other way to initialize variables, known as **constructor initialization**, is done by enclosing the initial value between parentheses (()):

type identifier (initial_value) ;

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

Program 2

```
// initialization of variables

#include <iostream.h>

int main ()
{
  int a=5;               // initial value = 5
  int b(2);              // initial value = 2
  int result;            // initial value undetermined

  a = a + 3;
  result = a - b;
  cout << result;

  return 0;
}
```

```
6
```

Variable Scope

Scope is a region of the program and broadly speaking there are three places, where variables can be declared:

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

We will learn what a function is and it's parameter in subsequent chapters. Here, let us explain what local and global variables are.

Local Variables:

Variables that are declared inside a function or block are 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. Following is the example using local variables.

Program 3

```
#include<iostream.h>
int main ()
{
// Local variable declaration:
int a, b;
int c;
// actual initialization
a =10;
b =20;
c = a + b;
cout << c;
return0; }
```

```
30
```

Global Variables:

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your 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. Following is the example using global and local variables.

Program 4

```
#include<iostream.h>

// Global variable declaration:
int g=10;
int main ()

{

// Local variable declaration:
int a, b, c;

// actual initialization

a =10;

b =20;

c = a + b+ g;

cout << c;

return0;

}
```

```
40
```

A program can have same name for local and global variables but value of local variable inside a function will take preference.

Program 5

Code	Output

#include<iostream.h>
 	
// Global variable declaration:
int g=20;
int main ()
 	
{
 	
// Local variable declaration:
int g =10;
 	
cout << g;
 	
return0;
 	
}
	10

Questions:

1- Write a C++ program that computes the z value from the following equations:

a- z= a(b*d)-(c*b)+d where a=10, b=20, c=14, and d= 5

b- $z=1/x+1/x^2+1/x^3$ where x=2

2- Write a C++ program that prints a temperature in Celsius when a Fahrenheit is given. Use the formula:

$$C^0=5/9(F\text{-}32)$$

Experiment No. (3)

String and Constant

Object:

How defining and using stream of characters and constant in the program.

Theory:

Variables that can store non-numerical values that are longer than one single character are known as strings.

The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: <string> and have access to the std namespace (which we already had in all our previous programs thanks to the using namespace statement).

Program 1

Code	Output

`// my first string`
`#include <iostream.h>`
`#include <string.h>`
 	
`int main ()`
`{`
`  string mystring = "This is a string";`
`  cout << mystring;`
`  return 0;`
	This is a string

}	

Strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. Both initialization formats are valid with strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution.

Program 2

```
// my first string
#include <iostream.h>
#include <string.h>

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

```
This is the initial string content
This is a different string content
```

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal. An integer literal can also have a suffix that is a

combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order. Here are some examples of integer literals:

```
212// Legal
215u// Legal
0xFeeL// Legal
078// Illegal: 8 is not an octal digit
032UU// Illegal: cannot repeat a suffix
```

Following are other examples of various types of Integer literals:

```
85// decimal
0213// octal
0x4b// hexadecimal
30// int
30u// unsigned int
30l// long
30ul// unsigned long
```

Floating-point literals:

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form. While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159// Legal
314159E-5L// Legal
510E// Illegal: incomplete exponent
210f// Illegal: no decimal or exponent
.e55 // Illegal: missing integer or fraction
```

Boolean literals:

There are two Boolean literals and they are part of standard C++ keywords:

- A value of true representing true.
- A value of false representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character literals:

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in wchar_t type of variable. Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of char type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like new line (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

\n	Newline
\r	carriage return

\t	Tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

String literals:

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \

dear"

"hello, ""d""ear"

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. Its format is:

#define identifier value

For example:

```
#define PI 3.14159265
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

Program 3

```
// defined constants: calculate circumference

#include <iostream.h>

#define PI 3.14159
#define NEWLINE '\n';

int main ()
{
  double r=5.0;             // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

  return 0; }
```

Output:

```
31.4159
```

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 NEWLINE) by the code to which they have been defined (3.14159265 and '\n' 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 within the body of the program that the preprocessor replaces.

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:

```
const type variable = value;
```

```
const int pathwidth = 100;
const char tabulator = '\t';
const zipcode = 12440;
```

In case that no type is explicitly specified (as in the last example) the compiler assumes that it is of type int.

Program 4

```
#include<iostream.h>

int main()
```

```
{
const int LENGTH =10;
const int WIDTH =5;
const char NEWLINE ='\n';
int area;
area = LENGTH * WIDTH;
cout << area;
cout << NEWLINE;
return 0;
}
```

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

```
50
```

Questions:

1- Write a C++ program to compute the volume of a sphere.

$$v = 4 * \pi * r^3/3$$

2- Write a C++ program that computes the average of two marks. For example, if 75 is the degree of Maths and 89 of Chemistry then the average is 82.

Experiment No. (4)

Operators

Object:

How Mathematical and Logical Operators in the C++ program are.

Theory:

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

Assignation (=)

The assignation operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable a. The part at the left of the assignation operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these.
The most important rule of assignation is the *right-to-left* rule: The assignation operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of **b** to **a** at the moment of the assignation. Therefore a later change of **b** will not affect the new value of **a**.

Program 1

```
// assignation operator

#include <iostream.h>
int main ()
{
  int a, b;
  a = 10;
  b = 4;
 // a:10, b:4
  a = b;
 // a:4,  b:4
  b = 7;
 // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

```
a:4 b:7
```

This code will give us as result that the value contained in **a** is **4** and the one contained in **b** is **7**. Notice how **a** was not affected by the final

modification of **b**, even though we declared **a = b** earlier (that is because of the *right-to-left rule*).

A property that C++ has over other programming languages is that the assignation operation can be used as the **rvalue** (or part of an **rvalue**) for another assignation. For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;
a = 2 + b;
```

The following expression is also valid in C++:

```
a = b = c = 5;
```

It assigns 5 to the all the three variables: a, b and c.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+	Addition
-	Subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see may be ***modulo***; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Compound assignation

(+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignation operators:

Expression	**is equivalent to**
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

Program 2

```
// compund assignation

#include <iostream.h>

int main ()
```

```
5
```

```
{
 int a, b=3;
 a = b;
 a+=2;            // equivalent to a=a+2
 cout << a;
 return 0;
}
```

Increase and decrease (++, --)

C++ provides two special operators '++' and '--' for incrementing and decrementing the value of a variable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

The syntax of the increment operator is:
Pre-increment: ++variable
Post-increment: variable++
The syntax of the decrement operator is:
Pre-decrement: – –variable
Post-decrement: variable– –

In Prefix form first variable is first incremented/decremented, then evaluated
In Postfix form first variable is first evaluated, then incremented/decremented

Program 3

```
// compund assignation

#include <iostream.h>
int main ()
{
int x,y;
int i=10,j=10;
x = ++i;    //add one to i, store the result back in
```

```
11
10
```

```
x
y= j++;     //store the value of j to y then add one
to j
cout<<x;
cout<<y;
}
```

Questions:

1- First, determine the values of the variables for each of the following C++ statements manually:

   a- z=x++*y.
   b- z=2*++x*y.
   c- y %=x.

   Second, write a C++ program to find the above values to ensure from your answer.

2- Write a C++ program for the following expressions:

   a- $x+y^2+t/z$
   b- a^3-b^2/c^2+25

For the next experiments you shall need to know the following operators:

Relational and equality 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 true or 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 is. Here is a list of the relational and equality operators that can be used in C++:

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Here there are some examples:

```
(7 == 5)     // evaluates to false.
(5 > 4)      // evaluates to true.
(3 != 2)     // evaluates to true.
(6 >= 6)     // evaluates to true.
(5 < 5)      // evaluates to false.
```

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,

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

Be careful!

The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignation 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 true.

Logical operators (!, &&, ||)

(!) Operator is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)    // evaluates to false because the expression at its right (5 ==
5) is true.
!(6 <= 4)    // evaluates to true because (6 <= 4) would be false.
!true        // evaluates to false
!false       // evaluates to true.
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

(&&) Operator

a	**b**	**a && b**
true	true	True
true	false	False

false	true	False
false	false	False

(||) Operator

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

a	**b**	**a || b**
true	true	true
true	false	true
false	true	true
false	false	false

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false ).
```

(?) 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

```
7==5 ? 4 : 3     // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3   // returns 4, since 7 is equal to 5+2.
```

```
5>3 ? a : b      // returns the value of a, since 5 is greater than 3.
a>b ? a : b      // returns whichever is greater, a or b.
```

If condition is true the expression will return result1, if it is not it will return result2.

Example:

Code	Output

<pre>// conditional operator
 	
#include <iostream.h>
 	
int main ()
{
  int a,b,c;
 	
  a=2;
  b=7;
  c = (a>b) ? a : b;
 	
  cout << c;
 	
  return 0;
}</pre>
	7

In this example a was 2 and b was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b, with a value of 7.

(,) Comma operator

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

Would first assign the value **3** to **b**, and then assign **b+2** to variable **a**. So, at the end, variable **a** would contain the value **5** while variable **b** would contain value **3**.

(&, |, ^, ~, <<, >>) Bitwise Operators

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	**asm equivalent**	**Description**
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
```

```
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

sizeof() operator

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value **1** to **a** because char is a one-byte long type. The value returned by **sizeof** is a constant, so it is always determined before program execution.

Precedence of operators

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 expression:

```
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 correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Level	**Operator**	**Description**	**Grouping**
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right

8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= !=	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression.

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs (and), as in this example:

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

might be written either as:

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

Or a = (5 + 7) % 2;

Experiment No. (5)

Control Structure (if-statement)

Object:

Learning (if condition statements) in the C++ program.

Theory:

Conditional structure:

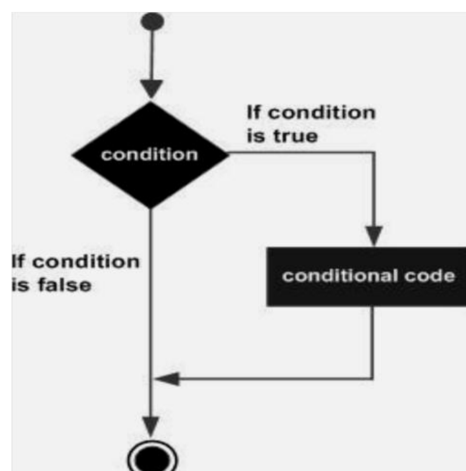
1. if statement

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 this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

Flow Diagram:



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

```
if (x == 100)
  cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
```

Program 1

```
#include <iostream.h>

int main ()

{

// local variable declaration:

int a = 10;

// check the boolean condition

if( a < 20 )

// if condition is true then print the following

cout << "a is less than 20;" << endl;

cout << "value of a is : " << a << endl;

return 0;   }
```

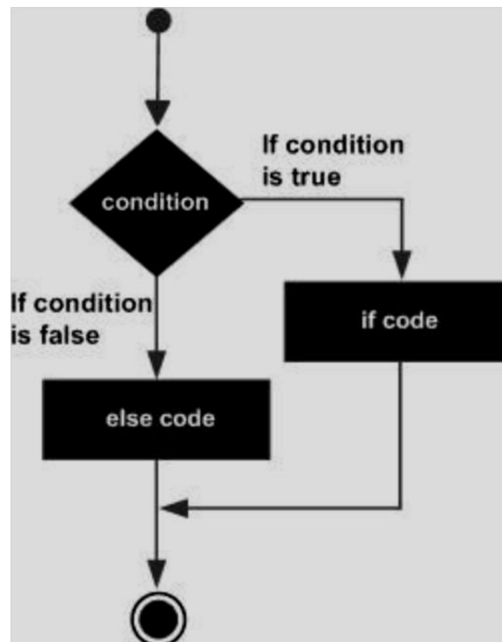
```
a is less than 20;

value of a is : 10
```

2. If else statement

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)
  cout << "x is 100";
else
  cout << "x is not 100";
```

The program 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.

The **if** + **else** structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling

if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

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 { }.

Program 2

```
#include <iostream.h>
int main ()
{
// local variable declaration:
int a = 100;
// check the boolean condition
if( a < 20 )
{
// if condition is true then print the following
cout << "a is less than 20;" << endl;
}
else
{
// if condition is false then print the following
cout << "a is not less than 20;" << endl;
}
cout << "value of a is : " << a << endl;
return 0; }
```

```
a is not less than 20;

value of a is : 100
```

Program 3

```
#include <iostream.h>

int main ()
{
// local variable declaration:
int a = 100;
// check the boolean condition
if( a == 10 )
{
// if condition is true then print the following
cout << "Value of a is 10" << endl;
}
else if( a == 20 )
{
// if else if condition is true
cout << "Value of a is 20" << endl;
}
else if( a == 30 )

{

// if else if condition is true

cout << "Value of a is 30" << endl;

}

else

{

// if none of the conditions is true

cout << "Value of a is not matching" << endl;

}

cout << "Exact value of a is : " << a << endl;

return 0; }
```

```
Value of a is not matching

Exact value of a is : 100
```

3. nested if statements: Syntax:

```
if ( boolean_expression 1)
{
// Executes when the boolean expression 1 is true
if(boolean_expression 2)
{
// Executes when the boolean expression 2 is true
}}
```

Program 4

Write a program that converts an integer number to character.

```
#include <iostream.h>

Void main()

{

  int number;

  cout<<"Enter an integer number:  ";

  cin>> number;

  if (number <=0)

   if (number >=127)

        cout<<"The  Character  is  :   "<<  (char)
number<<endl;

}
```

Note: the above two if statements can be replaced by the statement:
If (number>=0&&number <=127)
cout<<"The character is: " <<(char) number<<endl;

Questions:

1- Write a C++ program that computes the division for two integer numbers.

2- Write a C++ program that inputs an integer number and determines whether the number is even or odd.

3- Write a C++ program that computes the following equation:

$$y = \{\begin{cases} x+5 & x > 10 \\ 2x & x \leq 10 \end{cases}$$

Homework:

1- Write a program that reads a number and determines whether the number is positive, negative, or zero.

2- Write a C++ program that computes the following equation:

$$y = \begin{cases} 5x^2 + 6 & 0 \leq x \leq 100 \\ 0 & x \leq 0 \\ x^2 + 4x + 4 & x > 100 \end{cases}$$

Experiment No. (6)

Control Structure (switch-statement)

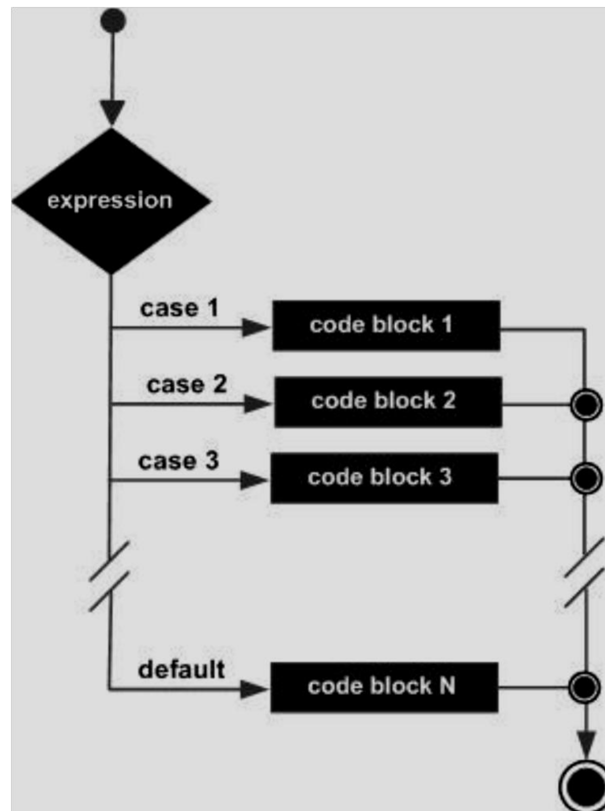
Object:

Learning (the selective structure switch) in the C++ program.

Theory:

The syntax of the switch statement is to check several possible constant values for an expression. Something similar to several if and else if instructions. Its form is the following:

```
switch (expression)
{
  case constant1:
    group of statements 1;
    break;
  case constant2:
    group of statements 2;
    break;
  .
  .
  .
  default:
    default group of statements
}
```



It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example	**if-else equivalent**

switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
	if (x == 1) {
	  cout << "x is 1";
	  }
	else if (x == 2) {
	  cout << "x is 2";
	  }
	else {
	  cout << "value of x unknown";
	  }

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.

Program 1

```
#include <iostream.h>
int main ()
{
// local variable declaration:
char grade = 'D';
switch(grade)
{
```

```
You passed
Your grade is D
```

```
case 'A' : cout << "Excellent!" << endl; break;

case 'B': cout << "good!" << endl; break;

case 'C': cout << "Well done" << endl; break;

case 'D': cout << "You passed" << endl; break;

case 'F' : cout << "Better try again" << endl; break;

default : cout << "Invalid grade" << endl;

}

cout << "Your grade is " << grade << endl;

return 0;
```

4. nested switch statements: Syntax

```
switch (ch1) {
case 'A':
cout << "This A is part of outer switch";
switch(ch2) {
case 'A':
cout << "This A is part of inner switch";
break;
case 'B': // ...
}
break;
case 'B': // ...

}
```

Questions:

1- Write a C++ program that performs the arithmetic operations (+, -, *, /) determined by a user input.

2- Write a C++ program that reads an average mark and prints the range of it. For example, if the mark is 95 then print your average between 90-100.

Experiment No. (7)

Control Structure (while-loop)

Object:

Learning iteration structures (while-loops) in the C++ program.

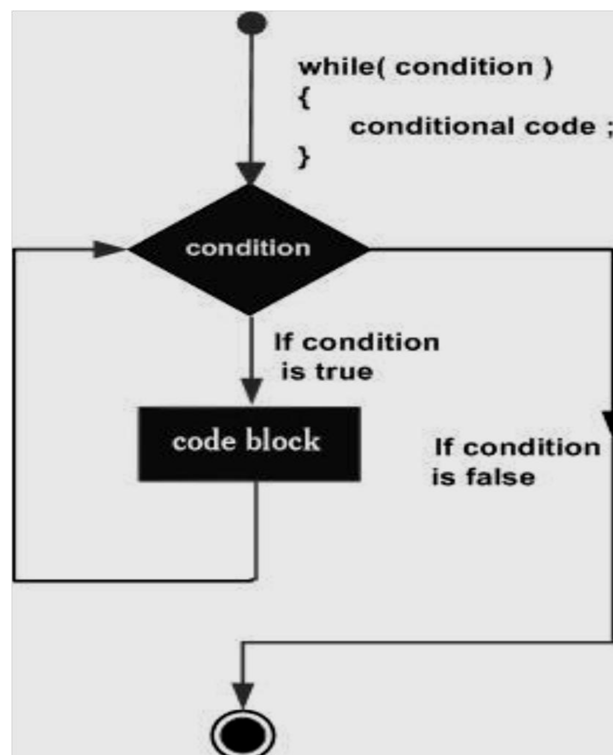
Theory:

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

1. The while loop

Its format is: **while (expression) statement**

Its functionality is simply to repeat statement while the condition set in expression is true. Flow the following diagram:



Program 1

```
// custom countdown using while

#include <iostream.h>
int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "loop is finish!";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, loop is finish!
```

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two possibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   cout << n << ", ";
   --n;
   (prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print loop is finish! and end program.

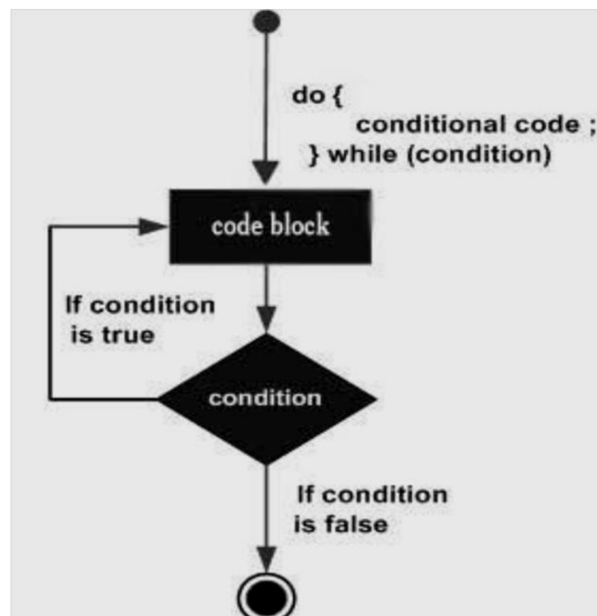
When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever.

In this case we have included **--n**; that decreases the value of the variable that is being evaluated in the condition **(n)** by one - this will eventually make the condition **(n>0)** to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.

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. Flow the following diagram:



For example, the following example program echoes any number you enter until you enter 0.

Program 2

// number echoer
 	
#include <iostream.h>
 	
int main ()
{
unsigned long n;
do {
cout << "Enter number (0 to end): ";
cin >> n;
cout << "You entered: " << n << "\n";
} while (n != 0);
return 0;
}
	Enter number (0 to end): 12345
	You entered: 12345
	Enter number (0 to end): 160277
	You entered: 160277
	Enter number (0 to end): 0
	You entered: 0

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the a value 0 in the previous example you can be prompted for more numbers forever.

Questions:

1- Write a C++ program that computes the sum of ten numbers input by the user. Use while loop.
2- Write a C++ program that computes the sum of consecutive integer number 1+2+3+….+ n. Use do/while loop.

Experiment No. (8)

Control Structure (for-loop)

Object:

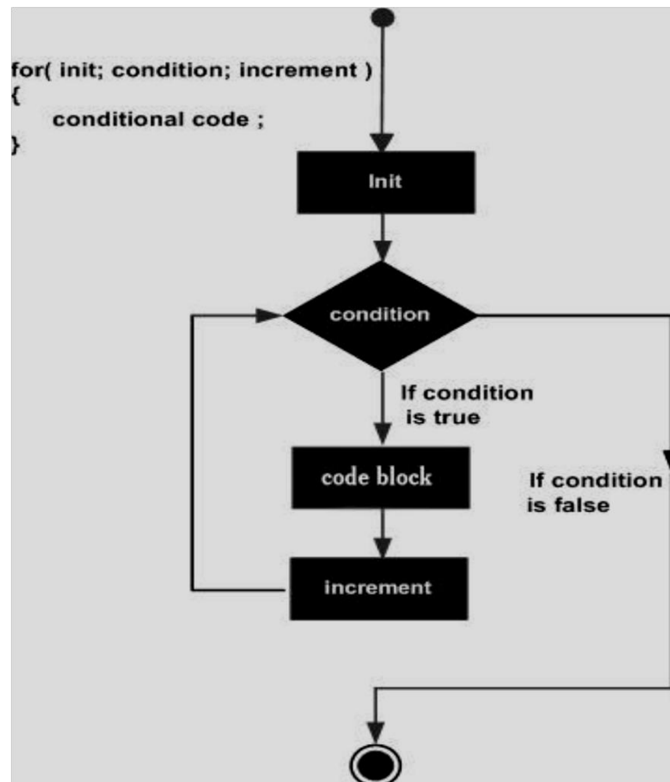
Learning Iteration structures (loops) in the C++ program.

Theory:

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:

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, otherwise the loop ends and statement is skipped (not executed).
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.



Program 1

// countdown using a for loop
#include <iostream.h>
int main ()
{
for (int n=10; n>0; n--) {
cout << n << ", ";
}
cout << "loop completed!";
return 0;
}
	10, 9, 8, 7, 6, 5, 4, 3, 2, 1, loop completed!

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include

an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
  // whatever here...
}
```

This loop will execute for 50 times if neither **n** or **i** are modified within the loop:

```
                                          → Initialization
                                          → Condition
for ( n=0, i=100 ; n!=i ; n++, i-- )
                                          → Increase
```

n starts with a value of **0**, and **i** with **100**, the condition is **n!=i** (that **n** is not equal to i). Because **n** is increased by one and **i** decreased by one, the loop's condition will become false after the **50th** loop, when both **n** and **i** will be equal to **50**.

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, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

Program 2

Code	Output

<pre>// break loop example
 	
#include <iostream.h>
 	
int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}</pre>
	10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

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.

Program 3

For example, we are going to skip the number 5 in our countdown:

Code	Output

<pre>// continue loop example
#include <iostream.h>
 	
int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;</pre>
	10, 9, 8, 7, 6, 4, 3, 2, 1, loop completed!

```
    cout << n << ", ";
  }
  cout << "loop completed!";
  return 0;
}
```

Questions:

1- Write a C++ program that prints the numbers from 1 to 20.

2- Write a C++ program that computes the sum of ten integer numbers input by the user.

3- Write a C++ program that computes the factorial of an integer number.

Homework:

1- Write a C++ program that computes the power of an integer number.

2- Write a C++ program that computes the following series:

$$z = x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \cdots + \frac{x^n}{n!}$$

Experiment No. (9)

Array (Part I)

Object:

Learning how to define and use an array one-dimension in C++ programming language.

Theory:

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays:

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type  arrayName [ arraySize ];
```

This is called a single-dimension array. The arraySize must be an integer constant greater than zero and type can be any valid C++ data type. For

example, to declare a 10-element array called balance of type double, use this statement:

```
double  balance[10];
```

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

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

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

double balance [] = {1000.0, 2.0, 3.4, 17.0, 50.0};

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements:

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays.

Program 1

Write a C++ program that calculates the sum and average of an initialized integer array.

```
#include <iostream.h>

void main ()

{

int b[ 10 ]={9, 3, 11, 7, 1};

int sum=0;

for ( int i = 0; i < 5; i++ )

 sum+=b[i];

 cout<<"Sum is " <<sum << endl;

  <<"Average is "<<sum/5.0;

}
```

Program 2

Write a C++ program that inputs ten integer values into an array and finds the maximum number in the array.

```
#include <iostream.h>

void main ()

{

   const int size=10;
```

```
int c[size], max;
cout<<"Enter ten integer values:  " ;
for (int i=0; i<10; i++)
 cin>>c[i];
 max=c[0];
for (int i=0; i<10; i++)
if (c[i]>max)
   max=c[i];
cout<<"The maximum number is "<<max;
}
```

Questions:

1. Write a C++ program that reads a one-dimensional array b and find minimum number.
2. Write a C++ program that reads a one-dimensional array b and sort its elements in ascending order.

Experiment No. (10)

Array (Part II)

Object:

To solve more problems about a one-dimension array in addition to an array of string.

Theory:

String is a character array that is terminated with null. Null is zero and can be expressed as NUL or ‘\0’. The compiler adds the null to the end of string automatically.

For example:

char name[11];//hold 10 characters plus null

char str[12]={‘H’,’e’,’l’,’l’,’o’,’ ’,’t’,’h’,’e’,’r’,’e’};

or

char str[12]=”Hello there”;//string constant plus null

str

H	e	l	l	o		t	h	e	r	e	0

Program 3

Write a C++ program that reads a string and then computes the number of capital letters in the string.

```
#include <iostream.h>
void main ()
{
    char str[30];
    int count=0
   cout<<"Enter your string:  ";
    cin>>str;
    for ( int i = 0; i < =30; i++ )
    if (str[i]>='A'&&str[i]<='Z')
      count++;
cout<<"No. of capital letters is "<< count;

}
```

Questions:

1- Write a C++ program that computes the number of even integer numbers in an array entered by the user.

2- Write a C++ program that computes the length of a string entered by the user.

Experiment No. (11)

Array (Part III)

Object:

Learning how to define and use multidimensional arrays in C++ programming language.

Theory:

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

int threedim[5][10][4];

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

type arrayName [x][y];

Where type can be any valid C++ data type and arrayName will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array a, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multi-dimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
{0, 1, 2, 3} , /* initializers for row indexed by 0 */
{4, 5, 6, 7} , /* initializers for row indexed by 1 */
{8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts ie. row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram.

Program 4

```
#include <iostream>
using namespace std;
int main ()
{
// an array with 5 rows and 2 columns.
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
// output each array element's value
for ( int i = 0; i < 5; i++ )
for ( int j = 0; j < 2; j++ )
{
cout << "a[" << i << "][" << j << "]: ";
cout << a[i][j]<< endl;
}
}
return 0;
}
```

Program 5

Write a C++ program that finds the average of each row of a 3x4 matrix input by the user.

```
#include <iostream>
void main ()
{
   int a[3][4];
    int sum;
cout<<"Enter 3x4 integer matrix:  ";
for (int i=0; i<3; i++)
  for (int j=0; j<4; j++)
    cin>>a[i][j];
cout<<"Average of each row : "<<endl;
for (i=0; i<3; i++)
{
  sum=0;
   for (int j=0; j<4; j++)
   sum+=a[i][j];
   cout<<sum/4.0<<endl;
   }
}
```

Questions:

1- Write a C++ program that adds two initialized 3x4 matrices A and B and then stores the result in a matrix C.

2- Write a C++ program that exchanges row3 with row1 in 4x4 integer matrix input by the user.

Experiment No. (12)

Array (Part IV)

Object:

To solve more problems about a Two-dimension array in addition to an array of string.

Theory:

To create an array of strings, we use a two-dimensional character array. The number of rows determines the number of strings and the number of columns specifies the maximum length of each string.

For example:

char str array[30][80];

char day[7][10]={“Sunday”, Monday”, ”Tuesday”, ”Wednesday”, ”Thursday”, “Friday”, “Saturday”};

To access an individual string, we simply specify only the row index. For example: cout<< day[1];

Questions:

1- Write a C++ program that inputs a 4x4 integer matrix and finds the maximum value in the primary diagonal and the minimum value in the secondary diagonal.

2- Write a C++ program that converts a two dimensional array into one dimensional array. Then print the 1D.

3- Use array of string to write a C++ program that prints the week days.

Experiment No. (13)

Function (Part I)

Object:

Learning how to define and use a Function in C++ programming language.

Theory:

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++. A function is a **group of statements that is executed when it is called from some point of the program.** The following is its format:

type name (parameter1, parameter2, ...) { statement }

where:

- Type is the data type.

- Name is the identifier by which it will be possible to call the function.
- Parameters (as many as needed): Each parameter consists of a data type followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- Statements are the function's body. It is a block of statements surrounded by braces { }.

Program 1

Code	Output

`// function example`
`#include <iostream.h>`
 	
`int addition (int a, int b)`
`{`
`  int r;`
`  r=a+b;`
`  return (r);`
`}`
`int main ()`
`{`
`  int z;`
`  z = addition (5,3);`
`  cout << "The result is " << z;`
`  return 0;}`
	The result is 8

A C++ program always begins its execution by the main function. So we can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

```
int addition (int a, int b)
                  ↑      ↑
z = addition (    5  ,   3   );
```

The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.

At the point at which the function is called from within main, the control is lost by main and passed to the function addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables **int a** and **int b** within the function.

Function addition declares another local variable (**int r**), and by means of the expression **r=a+b**, it assigns to r the result of a plus b. Because the actual parameters passed for **a** and **b** are **5** and **3** respectively, the result is **8**.

The following line of code:

```
return (r);
```

Finalizes function addition, and returns the control back to the function that called it in the first place (in this case, main). At this moment the program follows it regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (**return (r);**), which at that moment had a value of **8**. This value becomes the value of evaluating the function call.

```
int addition (int a, int b)
 | 8
 v
z = addition (  5  ,   3  );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable **z** will be set to the value returned by addition **(5, 3)**, that is **8**. To explain it another way, you can imagine that the call to a function (**addition (5,3)**) is literally replaced by the value it returns (**8**).

The following line of code in main is:

```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.

Program 2

Code	Output

// function example
#include <iostream.h>
int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return (r);
}
	The first result is 5
	The second result is 5
	Enter your fun parameters
	8 6
	The third result is 2
	The fourth result is 6

```
int main ()
{
  int x, y, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction
(7,2) << '\n';
  cout<<"enter your fun parameters<< '\n';
  cin>>x>>y;
  cout << "The third result is " << subtraction (x,y)
<< '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
  return 0;
}
```

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```

The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to function subtraction are the values of x and y, that are 8 and 6 respectively, giving 2 as result.

Functions with no type (using void)

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement

You will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the data that will be returned by the function with the return statement). But what if we want to return no value? Imagine that we want to make a function just to show a message

on the screen. We do not need it to return any value. In this case we should use the void type for the function. This is a special specifier that indicates absence of type.

Program 3

```
// void function example
#include <iostream.h>

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
  return 0;
}
```

```
I'm a function!
```

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.

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

1. Call by value

The **call by value** method of passing arguments to a function 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.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() **definition as follows.**

Program 4

```
#include <iostream.h>

// function declaration

void swap(int x, int y);

int main ()

{

// local variable declaration:

int a = 100;

int b = 200;

cout << "Before swap, value of a :" << a << endl;

cout << "Before swap, value of b :" << b << endl;

// calling a function to swap the values.

swap(a, b);

cout << "After swap, value of a :" << a << endl;

cout << "After swap, value of b :" << b << endl;

return 0;

}
```

```
// function definition to swap the values.
void swap(int x, int y)
{
int temp;
temp = x; /* save the value of x */
x = y; /* put y into x */
y = temp; /* put x into y */
return; }
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Questions:

1- Write a C++ program that finds the max value in an integer array a[10] using the function maxa().

2- Write a C++ program that computes the factorial of an integer number using the function factorial().

3- Write a C++ program that computes the power of an integer number using the function power().

Experiment No. (14)

Function (Part II)

Object:

To learn passing an array to a function.

Theory:

C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1
Formal parameters as a pointer as follows:

```
void myFunction(int *param)
{ . . . . }
```

Way-2
Formal parameters as a sized array as follows:

```
void myFunction(int param[10])
{ . . . }
```

Way-3
Formal parameters as an unsized array as follows:

```
void myFunction(int param[])
{ . . . }
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows.

Program 1

```
#include <iostream.h>

// function declaration:
double getAverage(int arr[], int size);

int main ()
{
// an int array with 5 elements.
int balance[5] = {1000, 2, 3, 17, 50};
double avg;
// pass pointer to the array as an argument.
avg = getAverage( balance, 5 ) ;
// output the returned value
cout << "Average value is: " << avg << endl;
return 0;
}

double getAverage(int arr[], int size)
{
int i, sum = 0;
double avg;
for (i = 0; i < size; ++i)
{
sum += arr[i];
}
avg = double(sum) / size;
return avg;
}
```

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

Average value is: 214.4

As you can see, the length of the array doesn't matter as far as the function is concerned because C++ performs no bounds checking for the formal parameters.

Return array from functions

C++ does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example.

Program 2

// main function to call above defined function.

```
#include<iostream.h>
int main ()
{
// a pointer to an int.
int *p;
p = getRandom();
for ( int i = 0; i < 10; i++ )
{
cout << "*(p + " << i << ") : ";
cout << *(p + i) << endl;
}
return 0;
}
```

Program 3

Write a C++ program that reads, sorts and prints an integer array a[10] using three functions setarray(), sortarray, and putarray().

```
#include <iostream.h>

void setarray(int b[10]);
void sortarray(int b[10]);
void putarray(int b[10]);
void main()
{
```

```
    int a[10];
    setarray(a);
    sortarray(a);
    cout<<"Sorted array is ";
    putarray(a);

}
void setarray(int b[10])
{ cout<<"Enter ten integer array: ";
          for (int i=0;i<10;i++)
          cin>>b[i];
}

void sortarray(int b[10])
{     for (int i=0;i<9;i++)
        for (int j=i+1;j<10;j++)
         if (b[i]>b[j])
           { int temp=b[i];
             b[i]=b[j];
             b[j]=temp;  }
}

void putarray(int b[10])
{
   for (int i=0;i<10;i++)
          cout>>b[i];
          cout<<endl;
}
```

Experiment No. (15)

Function (Part III)

Object:

To learn passing an array to a function by reference.

Theory:

- When we need the function returns more than one value, we can pass arguments by reference to the function.
- A reference provides an another name for a variable (It's actually the memory address of the variable that is passed).
- An important advantage of passing by refrefence is that the function can access the actual variables in the calling program. This provides a mechanism for passing more than one value from the function back to the calling program.

Program 1

A simple program to understand the reference.

```
#include<iostream.h>
void main()
{  int y=10;
   int&w=y;
   cout<<y<<endl;
   w++;
cout<<y<<endl;
}
```

Program 2

Write a C++ program that computes the square of an integer number using the function sqr() but the argument is passed by reference.

```
#include <iostream.h>

void sqr(int&)

void main()

{  int n;

   cout<<"Enter an integer number:  ";
```

```
 cin>>n;
sqr(n);
cout<<"The square is "<<n<<endl;
}
```

Experiment No. (16)

Function (Part V)

Object:

Learning how to define and use a recursive and overloading function in C++ programming language.

Theory:

Recursive Function

A recursive function is a function that calls itself in order to perform a task of computation. There are two basic components of a recursive solution:

- Termination step, stating the solution when the process comes to an end.
- Inductive step, calling the function itself with a renewed parameter value.

Program 1

Write a C++ program that computes the factorial of a positive integer number using the recursive function factorial().

```
#include<iostream.h>
Long factorial(int);
Void main()
{ int number;
```

```
cout<<”Enter a positive integer number: “;
cin>>number;
cout<<”The factorial is “<<factorial(number);
}
long factorial (int n)
{ if (n==0)
    return 1;
else
return (n*factorial(n-1)); }
```

Function overloading

Overloading refers to the use of the same thing for different purposes. Function overloading means that we can use the same function name to create functions that perform a variety of different task.

Program 2

Write a C++ program that computes the area of square and the area of rectangle using the overloaded function area().

```
#include<iostream.h>
int area(int);
int area( int, int);
void main()
{ int length, width;
  cout<<”Enter a length of square: ”;
   cin>>length;
cout<<”The area of square is “<<area(length)<<endl;
cout<<”Enter a length and width of rectangle : “;
cin>>length>>width;
cout<<”The area of rectangle is “<<area(length, width)<<endl;
}
int area(int a)
{
  return (a*a);
}
int area(int a, int b)
{
return (a*b);
}
```

Questions:

1- Write a C++ program that computes the power of an entered integer number using the recursive function power().

2- Write a C++ program that adds two numbers of different numeric data types(e.g. integer, float, double) using the overloaded function add().

Experiment No. (17)

Structure (Part I)

Object:

Learning how to define and use the structure in C++ programming language.

Theory:

When dealing with the students in a school, many variables of different types are needed. For example, structure STUDENT which includes name, age, and marks.

Program 1

```
#include <iostream.h>
struct STUDENT
{
        char name[80];
        int, age;
        float marks;
} ;
int main()
{       // declare two variables of the new type
      STUDENT s1, s3;
```

```
    //accessing of data members
    cin>> s1.name>> s1.age>>s1.marks;
    cout <<s1.name<<s1.age <<s1.marks;
    //initialization of structure variable
    STUDENT s2 = {"Aniket",17,92};
    cout <<s2.name<<s2.age <<s2.marks;
    //structure variable in assignment statement
    s3=s2;
    cout <<s3.name<<s3.age <<s3.marks;
    return 0;
}
```

Arrays of Structure
Program 2

Write a C++ program that inputs and prints the personal details (e.g. name, age, weight, phone_no) for 10 users using the structure Person.

```
#include<iostream.h>
struct Person
{       char name[10];
        int age;
        float weight;
        char phone_no[20];
};
void main()
{ const int size=10;
   Person p[size];//Define array of structure variables
    for (int i=0; i<size; i++)
       {  cout<<"Person   "<<i+1<<endl;
          cout<<"Enter name:  ";  cin>>p[i].name;
          cout<<"Enter age:  ";  cin>> p[i].age;
          cout<<"Enter weight:  ";  cin>>p[i].weight;
          cout<<"Enter phone_number:  ";   cin>>p[i].phone_no;
          cout<<endl;
        }
    for (int i=0; i<size; i++)
       {  cout<<"Person   "<<i+1<<endl;
          cout<<"Name:  ";  <<p[i].name<<endl;
          cout<<"Age:  ";  <<p[i].age<<endl;
          cout<<"Weight:  "<<p[i].weight<<endl;
          cout<<"Phone_number:  ";  <<p[i].phone_no<<endl;
        }   }
```

Questions:

1- Write a C++ program that inputs and prints the name and date of birth using the structure Birth.

2- Write a C++ program that inputs and prints hours, minutes and seconds using the structure Time.

Experiment No. (18)

Structure (Part II)

Object:

Learning how to define and use the structure in C++ programming language.

Theory:

Passing Structure to Functions

1- Passing structure by value

Program 1

Write a C++ program that computes the area of rectangle using the function area() and the structure Rectangle.

```
#include <iostream.h>

struct Rectangle
{ int length;
  int width;
};
int area (Rectangle);
void main ()
{ Rectangle rect;
   cout<<"Enter the length and width of rectangle: ";
   cin>>rect.length>>rect.width;
```

```
  cout<<"The area of rectangle is "<<area(rect);
}
int area(Rectangle  r)
{
return(r.length*r.width);
}
```

2- Passing structure by reference

Program 2

Write a C++ program that inputs and prints the date (day, month, and year) of birth using the structure Date and the functions getdate() and putdate().

```
#include <iostream.h>

struct Date
{ int day;
  int month;
  int year;
};
void getdate(Date&);
void putdate(Date&);

void main ()
{   Date d;
    getdate(d);
    putdate(d);
}
void getdate(Date& dd)
{  cout<<"Enter your date of birth in day, month, year: ";
   cin>>dd.day>>dd.month>>dd.year;
}
void putdate(Date& dd)
{   cout<<"You were born in "<<dd.day<<"/"<<dd.month<<"/"<<dd.year;
}
```

Question:

Write a C++ program that inputs a time in hours, minutes and seconds, and then converts and prints the overall time in seconds, then in minutes and then in hours using the functions seconds(), minutes(), hours() and the structure Time.

Experiment No. (19)

Pointer (Part I)

Object:

Learning how to define and use Pointer variables in C++ programming language.

Theory:

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

type *var-name;

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int *ip;          // pointer to an integer
double *dp;       // pointer to a double
float *fp;        // pointer to a float
char *ch          // pointer to character
```

Using Pointers in C++:

There are few important operations, which we will do with the pointers very frequently.

(a) we define a pointer variables

(b) assign the address of a variable to a pointer

(c) finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

Program 1

```
#include <iostream.h>

int main ()
{
int var = 20; // actual variable declaration.
int *ip; // pointer variable
ip = &var; // store address of var in pointer variable
cout << "Value of var variable: ";
cout << var << endl;
// print the address stored in ip pointer variable
cout << "Address stored in ip variable: ";
cout << ip << endl;
// access the value at the address available in pointer
cout << "Value of *ip variable: ";
cout << *ip << endl;
return 0;
}
```

Program 2

The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream. Consider the following program:

```
#include <iostream.h>
int main ()
{
int *ptr = NULL;
cout << "The value of ptr is " << ptr ;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:
The value of ptr is 0

C++ Pointer Arithmetic

As you understood pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and –

The arithmetic update will depend on type of pointer.

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmatic operation on the pointer:

ptr++

The **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer.

Program 3

```
#include <iostream.h>
const int MAX = 3;
int main ()
{
int var[MAX] = {10, 100, 200};
int *ptr;
// let us have array address in pointer.
ptr = var;
for (int i = 0; i < MAX; i++)
{
cout << "Address of var[" << i << "] = ";
cout << ptr << endl;
cout << "Value of var[" << i << "] = ";
cout << *ptr << endl; // point to the next location
ptr++;
}
return 0; }
```

Decrementing a Pointer:

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

Program 4

```
#include <iostream.h>
const int MAX = 3;
int main ()
{
int var[MAX] = {10, 100, 200};
int *ptr;
// let us have address of the last element in pointer.
ptr = &var[MAX-1];
```

```
for (int i = MAX; i > 0; i--)
{
cout << "Address of var[" << i << "] = ";
cout << ptr << endl;
cout << "Value of var[" << i << "] = ";
cout << *ptr << endl;
// point to the previous location
ptr--;
}
return 0;
}
```

Experiment No. (20)

Pointer (Part II)

Object:

Learning how to define and use pointers and array in C++ programming language.

Theory:

C++ Pointers vs Arrays

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Consider the following program.

Program 1

```
#include <iostream.h>
void main()
{  int a[5]={31, 54, 77, 52, 93};
   for (int i=0; i<5; i++)
   cout<<a[i]<<endl;
```

```
   cout<<endl;
 // Or use pointer
     for (int i=0; i<5; i++)
     cout<<*(a+i)<<endl; }
```

Program 2

```
#include <iostream.h>
void main()
{  int a[5]={31, 54, 77, 52, 93};
   int *ptr;
   ptr=a;
   for (int i=0; i<5; i++)
   cout<<*(ptr++)]<<endl;
}
```

Question:

Write a C++ program that uses pointer to read an integer of ten marks a[10] and compute the average using pointer and print the result.

Experiment No. (21)

Pointer (Part III)

Object:

Learning how to define and use pointers and function in C++ programming language.

Theory:

Pointer to Function

The **call by pointer** method of passing arguments to a function 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 passed argument.

To pass the value by pointer, argument pointers are passed to the functions just like any other value. Accordingly, you need to declare the function parameters as pointer types as in the following function swap()**, which exchanges the values of the two integer variables pointed to by its arguments.**

Program 1

```
#include <iostream.h>
// function declaration
void swap(int *x, int *y);
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;
/* calling a function to swap the values.
* &a indicates pointer to a ie. address of variable a and
* &b indicates pointer to b ie. address of variable b.
*/
swap(&a, &b);
cout << "After swap, value of a :" << a << endl;
cout << "After swap, value of b :" << b << endl;
return 0;

}

// function definition to swap the values.
void swap(int *x, int *y)
{
int temp;
temp = *x; /* save the value at address x */
```

```
*x = *y; /* put y into x */
*y = temp; /* put x into y */
return;

}
```

Question:

Write a C++ program that finds the maximum value in an entered integer a[10] using the function max(int*, int).

Experiment No. (22)

Pointer (Part IV)

Object:

Learning how to define and use pointers and Structure and string in C++ programming language.

Theory:

Pointers and Structures

Program

```
include <iostream.h>
struct Rectangle
{  float length;
   Float width;
};
void main()
{  Rectangle *rect;
   float a;
   cout<<"Enter the length and width of rectangle: ";
   cin>> rect->length*rect->width;
   a=rect->length*rect->width;
   cout<<"The area of rectangle is "<<a;
}
```

Pointers and String

Program 1

```
#include <iostream.h>
void main()
{
   char str1[]=”Defined as an array”;
   char* str2=”Define as a pointer”;
   cout<<str1<<endl;
   cout<<str2<<endl;
// str1++; // cannot do this because str1 is a constant
  str

}
```

Program 2

```
#include <iostream.h>
void dispstr(char*);
void main()
{
   char str1[]=”Idle people have the least leisure.”;
   dispstr(str);
}
void disptr (chr* ps)
{  while (*ps)
      cout<<*ps++;
      cout << endl;
}
```

Experiment No. (23)

Files

Object:

Learning how to write and read files in C++ programming language.

Theory:

Sometimes our program can generate a large amount of data as output, or it requires a large amount of data as input. It is not feasible either to print large amount of data to the screen or to read it from the keyboard. In these cases, we use data files to store and/or retrieve the data.

Writing Data to a File

Program 1

```
#include <fstream.h>
#include <iostream.h>
void main()
{
   char ch='x';
   int j= 77;
   double d= 6.02;
   char str1[] = "Ahmed"; //string without embedded spaces
   char str2[] ="Hassan";
fstream myfile;  //creat a file object
myfile.open("fdata.txt", ios::out); //open file for writing
myfile <<ch<<j<<' '<<d <<str1<<' '<<str2;  //write data to file
myfile.close();  //close file
cout<<"File written\n";
}
```

Reading Data from a File

Program 2

```
#include <fstream.h>
#include <iostream.h>
void main()
{
   char ch;
   int j;
   double d;
   char str1[10];
   char str2[10];
   fstream myfile;  //creat a file object
   myfile.open("fdata.txt", ios::in); //open file for reading
myfile >>ch>>j>>d >>str1>>str2;  //read data from file
cout<<ch<<endl<<j<<endl<<d<<endl<<str1<<str2<<endl;
myfile.close();  //close file
}
```

Adding Data to a File

Program 3

```
#include <fstream.h>
#include <iostream.h>
void main()
{
   char ch='y';
   int j=88;
   double d=7.02;
   char str1[] = "Mohammad";
   char str2[] ="Ali";
```

```
fstream myfile;
myfile.open("fdata.txt", ios::app); //open file for adding data
myfile <<endl<<ch<<j<<' '<<d <<str1<<'  '<< str2;
myfile.close();  //close file
cout<<"File is written\n";
}
```

End-of-file eof()

Program 4

```
#include <fstream.h>
#include <iostream.h>
void main()
{
    char ch;
    int j;
    double d;
    char str1[10];
    char str2[10];
    fstream myfile;
   myfile.open("fdata.txt", ios::in);
   while(!myfile.eof())
   {
      myfile >>ch>>j>>d >>str1>>str2;  //read data from file
     cout<<ch<<endl<<j<<endl<<d<<endl<<str1<<str2<<endl;
    }
myfile.close();  //close file
}
```

Experiment No. (24)

Stack

Object:

Learning how to define and use Stack in C++ programming language.

Theory:

A stack is a data structure that stores and retrieves items in a last-in-first-out (LIFO) manner.

Static and Dynamic Stacks:

Static Stacks

• Fixed size

• Can be implemented with an array

Dynamic Stacks

• Grow in size as needed

• Can be implemented with a linked list

Stack Operations:

Push causes a value to be stored in (pushed onto) the stack.

Pop retrieves and removes a value from the stack.

```
#include <fstream.h>
#include <iostream.h>
#define max 5  // size of the stack
int top, a[max];
void push(void)
{
   int x;
   if (top==max-1)  // condition for checking if Stack is Full
   {
     cout<<"stack is overflow\n";
     return;
    }
```

```
    cout<<"Enter a No.\n";
    cin>>"%d",&x;
    a[++top]=x; //increment the top and inserting element
    cout<<"%d succ.pushed\n",x;
 return;
}
void pop(void)
{
    int y;
    if (top==-1)// Condition for checking if Stack is Empty
    {
        cout<<"stack is underflow\n";
        return;
     }
     y=a[top];
    a[top--]=0;
    //insert 0 at place of removing element and decrement the top
    cout<<"%d succ.poped\n",y;
 return;
}
void display(void)
{  int i;
    If (top==-1)
{
    cout<<"stack is empty\n";
return;
}
cout<<"elements of Stack are :\n";
for (i=0;i<=top;i+=)
{
     cout<<"%d\n", a[i];
}
return;
}
void main (void)
{
   int c, top=-1;
   do
   {
      cout<<"1:Push\n2:Pop\n3:Display\n4:Exit\nChoice:  ";
```

```
      cin>>"%d",&c;
      switch (c)
     {
        case1: push(); break;
        case2: pop(); break;
        case3: display(); break;
        case4: cout<<"program ends\n"; break;
        default: cout<<"wrong choic\n"; break;
      }
    }while(c!=4);
}
```

Questions:

1- Write a C++ program to transfer elements from stack S1 to stack s2 so that the elements from S2 are in the same order as on S1. ***(You can use one additional stack).***

2- Write a C++ program to put the elements on the stack S in ascending order using two functions: ***sort_array and swap.***

Experiment No. (25)

Queue

Object:

Learning how to define and use Queue in C++ programming language.

Theory:

A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front. These restrictions ensure that the data is processed through the queue in the order in which it is received. In other words, a queue is a first in, first out (FIFO) structure.

Program

```
include <fstream.h>
#include <iostream.h>
#define MAX 50
int queue_array[MAX];
int rear=-1;
int front=-1;
void main()
{ int choice;
  while (1)
  {
     cout<<"1. Insert element to queue\n";
     cout<<"2. Delete element from queue\n";
     cout<<"3. Display elements of queue\n";
     cout<<"4. Quit\n";
     cout<<"Enter your choice: ";
     cin>>"%d", &choice;
     switch (choice)
     {
       case 1: insert(); break;
       case 2: delete(); break;
```

```
        case 3: display(); break;
        case 4: exit(1);
        default: cout<<"Worng choice \n";
}/*End of switch*/
}/*End of while*/
}/*End of main*/

insert()
{  int add_item;
   if (rear==MAX-1)
   cout<<"Queue Overflow \n";
   else
   {   if (front==-1)/*if queue is initially empty */
         front=0;
         cout<<"insert the element in queue: ";
         cin>>"%d", &add_item;
         rear=rear+1;
         Queue_array[rear]=add_item;
     }
 } /*End of insert()*/
delete()
{
  if (front==-1|| front>rear)
  { cout<<"Queue is Underflow\n";
    return;
   }
  else
  {
  cout<<"Element deleted from queue is : %\n", queue_array[front];
  front=front+1;
  }
}/*End of delete()*/
display()
{  int i;
   if (front==-1)
   cout<<"Queue is empty\n";
   else
   {
      cout<<"Queue is :\n";
      for(i=front; i<= rear; i++)
```

```
        cout<<"%d", queue_array[i];
        cout<<"\n";
      }
}/*End of display*/
```