C++ *language*

Introduction:

A digital computer is a useful tool for solving a great variety of **problems**. A solution to a problem is called an **algorithm**; it describes the sequence of steps to be performed for the problem to be solved. A simple example of a problem and an algorithm for it would be:

Problem: Sort a list of names in ascending lexicographic order.
Algorithm: Call the given list *list1*; create an empty list, *list2*, to hold the sorted list. Repeatedly find the 'smallest' name in *list1*, remove it from *list1*, and make it the next entry of *list2*, until *list1* is empty.

An algorithm is expressed in abstract terms. To be intelligible to a computer, it needs to be expressed in a language understood by it. The only language really understood by a computer is its own **machine language**. Programs expressed in the machine language are said to be **executable**. A program written in any other language needs to be first translated to the machine language before it can be executed.
A machine language is far too cryptic to be suitable for the direct use of programmers. A further abstraction of this language is the **assembly language** which provides mnemonic names for the instructions and a more intelligible notation for the data. An assembly language program is translated to machine language by a translator called an **assembler**.
Even assembly languages are difficult to work with. High-level languages such as C++ provide a much more convenient notation for implementing algorithms.
They liberate programmers from having to think in very low-level terms, and help them to focus on the algorithm instead. A program written in a high-level language is translated to assembly language by a translator called a **compiler**. The assembly code produced by the compiler is then assembled to produce an executable program.

Computer Program Language:

Can be divided into three levels, these levels are:
1-Low Level Language: as assembly language in which a mnemonic represents each of the machine language instructions for a particular computer.
2- Middle Level Language: as C++ and Java languages which are combination between low level language instructions and high level languages instructions.
3- High Level Language: as Basic, Pascal, Fortran,…etc which a single statement translates into one or more machine language instructions.
The high level language and middle level language are translated into machine language which is made up of binary-coded instructions, that is used directly by the computer called compiler program.

History of C++ Language:

C++ was created by Bjarne Stroustrup, beginning in 1979. The development and refinement of C++ was a major effort, spanning the 1980s and most of the 1990s. Finally, in 1998 an ANSI/ISO standard for C++ was adopted. In general terms, C++ is the object oriented version of C. It soon expanded into being a programming language in its own right. Today, C++ is nearly twice the size of the C language. Needless to say, C++ is one of the most powerful computer languages ever devised.

The Programming Process:

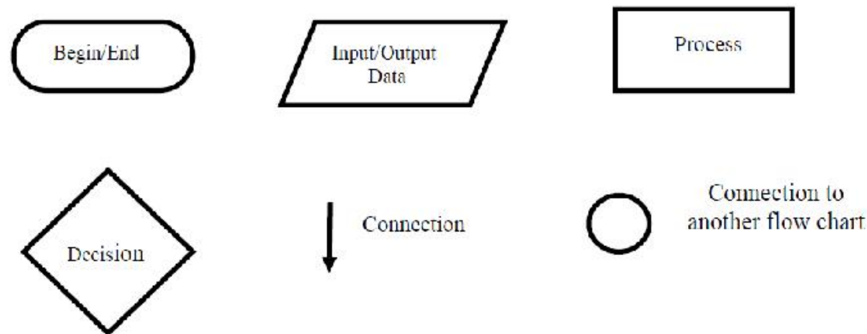
1- Specify the task.
2- Discover an algorithm for its solution.
3- Code the algorithm in C++ language.
4- Test the code.

Example: *Write algorithm to find the average of five integer numbers, and then print the positive result only?*
Answer:

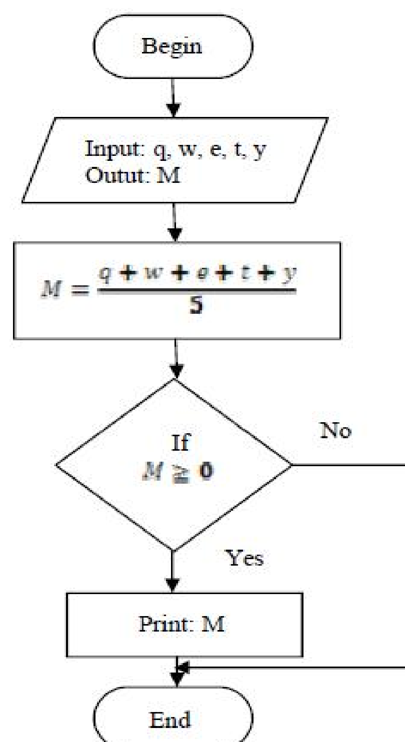
1- Start.
2- Input the numbers as q, w, e, t. y.
3- Output is M.
4- M=(q+w+e+t+y)/5
5- If (M 0) print M, otherwise finish.
6- End.

Flow Chart:

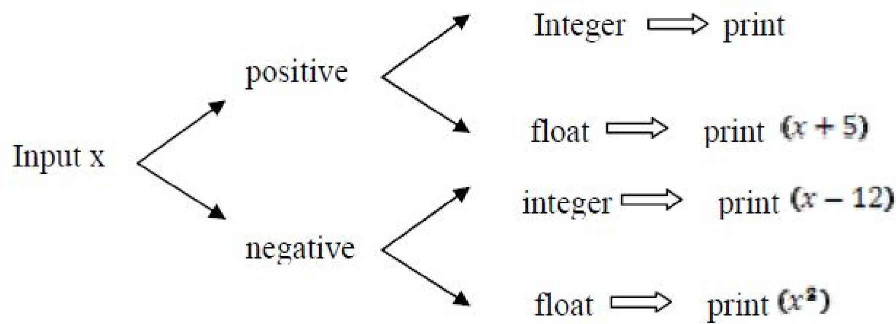
Flow chart is a set of symbols to identify both the operation required in the solution and the sequence in which they are performed. The commonly symbols are consisting of:



The flow chart with including algorithm for example above become as:



Question: Sketch flow chart including the algorithm to investigate the shape below.



The Main structure of C++ programs

The program below shows the main structure of c++ programs and when run, simply outputs the message Hello World.

```
1. #include <iostream.h>
2. int main (void)
3. {
4. cout<<"Hello World \n";
5. }
```

Line1: This line uses the **preprocessor directive** #include to include the contents of the header file iostream.h in the program. Iostream.h is a standard C++ header file and contains definitions for input and output.
Line2: This line defines a **function** called main. A function may have zero or more **parameters**; these always appear after the function name, between a pair of brackets. The word void appearing between the brackets indicates that main has no parameters. A function may also have a **return type**; this always appears before the function name. The return type for main is int (i.e., an integer number). All C++ programs must have exactly one main function. Program execution always begins from main.
Line3: This brace marks the beginning of the body of main.
Line4: This line is a **statement**. A statement is a computation step which may produce a value. The end of a statement is always marked with a semicolon (;). This statement causes the **string** "Hello World\n" to be sent to the cout **output stream**. A string is any sequence of characters enclosed in double-quotes. The last character in this string (\n) is a newline character which is similar to a carriage return on a type writer. A stream is an object which performs input or output. Cout is the standard output stream in C++ (standard output usually means your computer monitor screen). The symbol << is an **output operator** which takes an output stream as its left operand and an **expression** as its right operand, and causes the value of the latter to be sent to the former. In this case, the effect is that the string "Hello World\n" is sent to cout, causing it to be printed on the computer monitor screen.
Line5: This brace marks the end of the body of main.

Data type:

Programs are written to handle data. This is why the industry as a whole is often referred to as data processing, information technology, computer information systems, and so on. That data might be information about employees, parts to a mathematical computation, scientific data, or even the elements of a game. No matter what programming language or techniques you use, the ultimate goal of programming is to store, manipulate, and retrieve data. Data must be temporarily stored in the program, in order to be manipulated. This is accomplished via variables. A variable is simply a

place in memory set aside to hold data of a particular type. It is a specific section of the computer's memory that has been reserved to hold some data. It is called a variable because its value or content can vary. When you create a variable, you are actually setting aside a small piece of memory for storage purposes. The name you give the variable is actually a label for that address in memory. When you are deciding what type of variable to create, you need to think about what kind of data you might wish to store in that variable. If you want to store a person's age, then an int is a good choice. The int data type will hold whole numbers, and will, in fact, hold numbers much larger than you might need for a person's age. The long data type will also hold whole numbers, but it is designed for even larger whole numbers. If you wished to store bank balances, grade point averages, or perhaps temperatures, you would need to consider using a float. The float data type holds decimal values.
However if you were storing astronomical data, you would still want to hold decimals, but because they might be quite large, you would have to consider storing them in doubles. Picking what data type to use is actually simple. Just give some thought to the type of data you intend to store in a variable.

Integer Numbers

Variable	Type	Memory	Example	Range
Integer	int	2 bytes	int x	-32768 → +32767
	short	2 bytes	short y	-32768 → +32767
	long	4 bytes	long voltage1	-2147483648 → +2147483647
	literal long	4 bytes	1984L or 1984l	-2147483648 → +2147483647
	unsigned int	2 bytes	unsigned int salary	0 → 65535
	unsigned short	2 bytes	unsigned short age	0 → 65535
	unsigned long	4 bytes	unsigned longprice	0 → 4294967295
	unsigned literal long	4 bytes	40000U or 40000u or 40000LU or 40000ul	0 → 4294967295

Real Numbers

Variable	Type	Memory	Example	Range
Real	float	4 bytes	float x=0.05	$10^{-38} \rightarrow 10^{+38}$
	literal float	4 bytes	Y=0.05f or 0.05F	$10^{-38} \rightarrow 10^{+38}$
	Double	8 bytes	double y2=3.141592654	$10^{308} \rightarrow 10^{+308}$
	literal double	8 bytes	y2=3.1415926541 or y2=3.141592654L	$10^{-308} \rightarrow 10^{+308}$

In addition to the decimal notation used so far, literal reals may also be expressed in scientific notation. For example, 0.002164 may be written in the scientific notation as 0.002164 = 2.164E-3 or 2.164e-3 (the letter E or e) stands for *exponent*.

Characters

A char can store a single *alpha-numeric* type. In other words, it can store a number or a letter.

Variable	Type	Memory	Example	Range
Characters	char	1 bytes	'A' , 'a', '$', '55', '+'	-128 → 127
	unsigned character	1 bytes	unsigned char x2	0-255

A Literal Character is written by enclosing the character between a pair of single quotes (e.g., 'A'). Nonprintable characters are represented using **escape sequences** or **escape code** .for example:

Escape Sequence	Represents
'\n'	New line
'\r'	Carriage return
'\t'	Horizontal tab
'\v'	Vertical tab
'\b'	Backspace
'\f '	form feed
'\a'	Bell (alert)

Strings

A string is a consecutive sequence (i.e., array) of characters which are terminated by a null character. A **string variable** is defined to be type `char*`, and contain the address (memory location) of where the first character of the string appears.

A **literal string** is written by enclosing its characters between a pair of double quotes for example: "Example to show the use of backslash for writing a long string" Or A long string may extend beyond a single line, in which case each of the preceding lines should be terminated by a backslash. "Example to show \ the use of backslash for \ writing a long string". A common programming error results from confusing a ***single-character string*** (e.g., "A") with a ***single character*** (e.g., 'A'). These two are not equivalent. The first one consists of two bytes, whereas the second one consists of a single byte.

Variable declaration:

Before a variable appears in programs, you must declare it, the general variable declaration is:-

Data type variable list;

For example, you might declare a variable in the following manner.

```
int j;
```

Then, you have just allocated four bytes of memory; you are using the variable `j` to refer to those four bytes of memory. You are also stating that the only type of data that `j` will hold, is whole numbers. (The `int` is a data type that refers to integers). Now, whenever you reference `j` in your code, you are actually referencing the contents being stored at a specific address in memory. You can declare more than one variable on a single line.

```
int x, i_age, sgnal9;
```

All three variables are of type int. All variables have two important attributes:

A type: which is established when the variable is defined (e.g., integer, real, characters). Once defined, the type of a C++ variable cannot be changed.

A value: which can be changed by assigning a new value to the variable?

When a variable is define (declare), its value is undefined until it is actually assigned one. The assigning of a value to a variable for the first time is called **initialization**.

```
int x;
float y;
x=5;
y=4.7;
```

It is possible to declare a variable and assign it at the same time.

```
int num = 1500;
float number = 0.451;
```

It is important to ensure that a variable is an initialized before it is used in any computation. A default value is simply some starting value that the variable will hold, by default, if no other value is placed into it.

Variables Names:

Programming languages use names to refer to the various entries that make up the program. For example (variables, functions, macro), C++ imposes the following rules for creating valid names (also called **identifiers**). A name should consist of one or more characters, each of which may be a letter (i.e. "A"-"Z") a digit (i.e. "0-"9), or an underscore characters ("_") except that the first character may not be a digit. Upper and lower letters are distinct. ex. (x, y1, signal, signal2, _signal, Signal).
Certain words are reserved by C++ for specific purposes and may not be used as identifiers. These are called **reserved words** or **keywords** and summarized in table below:

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

The comments

A comment which is included in a program for the benefit of a person who are read the program in the future. To advice the complier that a group of characters is a comment and should be ignored, two types of comments can be written as follow.
1. ***A single-line comment*** the comment consist of one statement use // before the statement,
2. ***C's style*** the programmer must place the comment between /* and */ when the comment consist of several statement.

Simple Input/Output

The most common way in which a program communicates with the outside world is through simple, character-oriented Input/Out (IO) operations. C++ provides two useful operators for this purpose: >> for input and << for output. When you include a header file (iostream), you have access to all the functions defined in that file. By including this file you will have access to a number of functions for input and output.
The two most important functions are `cout` **output stream** and `cin` **input stream**.
Virtually all your input and output needs (at least for keyboard input and screen output) can be handled by these two functions.
The input operator >> takes an input stream as its left operand (cin is the standard C++ input stream which corresponds to data entered via the keyboard) and a variable (to which the input data is copied) as its right operand. Both << and >> return their left operand as their result.

Example: write a program in C++ to print on the screen of computer the integer number 5.
Ans:

First program is:	Second program is:	Third program is:

#include<iostream.h>
main()
{
int x=5;
cout<<x;
}
	#include<iostream.h>
	main()
	{
	int x=5;
	cout<<"x= "<<x;
	}
		#include<iostream.h>
		main()
		{
		cout<<5;
		}

Computer screen appearing 5	Computer screen appearing x= 5	Computer screen appearing 5

Question : Write a program which inputs a temperature reading expressed in Fahrenheit and outputs its equivalent in Celsius, using the formula:

$$C° = 5/9*(F° - 32)$$

Hexadecimal and Octal Constants

It is sometimes easier to use a number system based on 8 or 16 instead of 10. The number system based on 8 is called ***octal*** and uses the digits 0 through 7. The base 16 number system is called ***hexadecimal*** and uses the digits 0 through 9 plus the letters A through F, which stand for 10, 11, 12, 13, 14, and 15. Because of the frequency with which these two number systems are used, C++ allows you to specify integer constants in hexadecimal or octal instead of decimal if you prefer. A hexadecimal constant must begin with a **0x** (a zero followed by an x) or **0X**, followed by the constant in hexadecimal form. An octal constant begins with a **zero**. Here are two examples:

```
int hex = 0x80; // 128 in decimal
int oct = 012; // 10 in decimal
```

Example: what is the result of program that appearing on the screen of computer.

```
#include<iostream.h>
main()
{
int d=0x12, s=0x11;
cout<<(d+s);
}
```

Ans: The result appearing on the screen is 35

C++ Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. There are four general classes of operators in C++: **arithmetic**, **relational, logical**, and **bitwise**. In addition, there are some special operators for particular tasks.

1- Arithmetic Operators:

C++ provides five basic arithmetic operators. These are summarized in Table below :

Operator	Name	Example
+	Addition	12 + 4.9 // gives 16.9
-	Subtraction	3.98 - 4 // gives -0.02
*	Multiplication	2 * 3.4 // gives 6.8
/	Division	9 / 2.0 // gives 4.5
%	Remainder	13 % 3 // gives 1

Except for remainder (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are reals then the result will be a real (or double to be exact).

Example:

```
/* This program calculates the weekly gross pay for a
worker, based on the total number of hours worked and the
hourly pay rate. */
#include <iostream.h>
main ()
{
int workDays = 5; // Number of work days per week
float workHours = 7.5; // Number of work hours per day
float payRate = 33.50; // Hourly pay rate
float weeklyPay; // Gross weekly pay
weeklyPay = workDays * workHours * payRate;
cout << "Weekly Pay = " << weeklyPay << "\n";
}
```

2- Relational operators:

C++ provides six relational operators for comparing numeric quantities. These are summarized in Table below. Relational operators evaluate to 1 (representing the *true* outcome) or 0 (representing the *false* outcome).

Operator	Name	Example
==	Equality	5 == 5 // gives 1
!=	Inequality	5 != 5 // gives 0
<	Less Than	5 < 5.5 // gives 1
<=	Less Than or Equal	5 <= 5 // gives 1
>	Greater Than	5 > 5.5 // gives 0
>=	Greater Than or Equal	6.3 >= 5 // gives 1

Note that the <= and >= operators are only supported in the form shown. In particular, =< and => are both invalid and do not mean anything.

Example: Write an expression to test if a number (n) is even.
Solution: `n%2==0`

3- Logical operators:

C++ provides three logical operators for combining logical expression. These are summarized in Table below. Like the relational operators, logical operators evaluate to 1 or 0.

Operator	Name	Example
!	Logical Negation	!(5 == 5) // gives 0
&&	Logical And	5 < 6 && 6 < 6 // gives 1
||	Logical Or	5 < 6 || 6 < 5 // gives 1

Precedence Operators
Highest !
> >= < <=
= = !=
&&
Lowest ||

Note that here we talk of zero and nonzero operands (not zero and 1). In general, any nonzero value can be used to represent the logical *true*, whereas only zero represents the logical *false*. The following are, therefore, all valid logical expressions:

```
!20 // gives 0
10 && 5 // gives 1
10 || 5.5 // gives 1
10 && 0 // gives 0
```

C++ does not have a built-in Boolean type. It is customary to use the type `int` for this purpose instead. For example:

```
int sorted = 0; // false
int balanced = 1; // true
```

Example: writ a program in C++ language to test the operation of Relational and Logical Operators with printing the result appearing on the screen of computer.

Ans:

```
#include<iostream.h>
// Program to test Relational and Logical Operators
main()
{
int A=57,B=57;
char C='9';
cout<<"(A<57)="<<(A<57)<<endl;// endl is equivalent to "\n"
cout<<"(A<90)="<<(A<90)<<endl;
cout<<"(A<30)="<<(A<30)<<endl;
cout<<"(A<=57)="<<(A<=57)<<endl;
cout<<"(A>B)="<<(A>B)<<endl;
cout<<"(A>=B)="<<(A>=B)<<endl;
cout<<"(A==B)="<<(A==B)<<endl;
cout<<"(A!=B)="<<(A!=B)<<endl;
cout<<"(A==C)="<<(A==C)<<endl;
cout<<"(A&&0)="<<(A&&0)<<endl;
cout<<"(A||20)="<<(A||20)<<endl;
cout<<"(A=!C)="<<(A=!C)<<endl;
cout<<"A="<<A;
}
```

(A<57)=0 (A<90)=1 (A<30)=0 (A<=57)=1(A>B)=0 (A>=B)=1 (A==B)=1 (A!=B)=0
(A==C)=1 (A&&0)=0 (A||20)=1 (A=!C)=0A=57

4- Bitwise Operators

C++ provides six bitwise operators for manipulating the individual bits in an integer quantity. These are summarized in Table below.

Operator	Name	Example
~	Bitwise Negation	`~'\011'          // gives '\366'`
&	Bitwise And	`'\011' & '\027'  // gives '\001'`
|	Bitwise Or	`'\011' | '\027'  // gives '\037'`
^	Bitwise Exclusive Or	`'\011' ^ '\027'  // gives '\036'`
<<	Bitwise Left Shift	`'\011' << 2      // gives '\044'`
>>	Bitwise Right Shift	`'\011' >> 2      // gives '\002'`

Bitwise operators expect their operands to be integer quantities and treat them as bit sequences. Bitwise *negation* is a unary operator which reverses the bits in its operands. Bitwise *and* compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise. Bitwise *or* compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1 otherwise. Bitwise *exclusive or* compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.
Bitwise *left shift* operator and bitwise *right shift* operator both take a bit sequence as their left operand and a positive integer quantity *n* as their right operand. The former produces a bit sequence equal to the left operand but which has been shifted *n* bit positions to the left. The latter produces a bit sequence equal to the left operand but which has been shifted *n* bit positions to the right. Vacated bits at either end are set to 0.
Table below illustrates bit sequences for the sample operands and results in Table above. To avoid worrying about the sign bit (which is machine dependent), it is common to declare a bit sequence as an unsigned quantity:

```
unsigned char x = '\011';
unsigned char y = '\027';
```

Example	Octal Value	Bit Sequence							
x	011	0	0	0	0	1	0	0	1
y	027	0	0	0	1	0	1	1	1
~x	366	1	1	1	1	0	1	1	0
x & y	001	0	0	0	0	0	0	0	1
x | y	037	0	0	0	1	1	1	1	1
x ^ y	036	0	0	0	1	1	1	1	0
x << 2	044	0	0	1	0	0	1	0	0
x >> 2	002	0	0	0	0	0	0	1	0

Increment/Decrement Operators

The *auto increment* (++) and *auto decrement* (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. These are summarized in Table below; The examples assume the following variable definition:

```
int k = 5;
```

Operator	Name	Example
++	Auto Increment (prefix)	++k + 10 // gives 16
++	Auto Increment (postfix)	k++ + 10 // gives 15
--	Auto Decrement (prefix)	--k + 10 // gives 14
--	Auto Decrement (postfix)	k-- + 10 // gives 15

Precedence Operators	
Highest	++ – – – (unary minus)
	* / %
Lowest	+ –

Example: writ a program in C|++ language to test the operation of arithmetic operators with printing the result appearing on the screen of computer.

Ans:

```
#include<iostream.h>
//test arithmetic operators:
main()
{
int a=13, b=5;
cout<<a<<"+"<<b<<"="<<(a+b)<<endl;
cout<<"-b="<<(-b)<<endl;
cout<<a<<"*"<<b<<"="<<(a*b)<<endl;
cout<<a<<"/"<<b<<"="<<(a/b)<<endl;
cout<<a<<"%"<<b<<"="<<(a%b)<<endl;
cout<<"a++="<<a++<<endl;     //  endl is equivalent to "\n"
cout<<"++a="<<++a<<endl;
cout<<"--a="<<--a<<endl;
cout<<"a--="<<a--<<endl;
}
```

The result appearing on the screen of computer is:

13+5=18
-b=-5
13*5=65
13/5=2
13%5=3
a++=13
++a=15
--a=14
a--=14

Question: What is the difference between ++a and a++.

Question: find the result of the following if

`int n=7, m=24;`

i: `37/(5%3)`
ii: `m-8-n`
iii: `m%n++`
iv: `++m-n—`
v : `m*=n++`
vi: `m+=--n`

Assignment Operators

The assignment operator has a number of variants, obtained by combining it with the arithmetic and bitwise operators. These are summarized in Table below. The examples assume that *n* is an integer variable.

Operator	Example	Equivalent To
=	n = 25	
+=	n += 25	n = n + 25
-=	n -= 25	n = n - 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25
&=	n &= 0xF2F2	n = n & 0xF2F2
|=	n |= 0xF2F2	n = n | 0xF2F2
^=	n ^= 0xF2F2	n = n ^ 0xF2F2
<<=	n <<= 4	n = n << 4
>>=	n >>= 4	n = n >> 4

An assignment operation is itself an expression whose value is the value stored in its left operand. An assignment operation can therefore be used as the right operand of another assignment operation. Any number of assignments can be concatenated in this fashion to form one expression. For example:

```
int m, n, p;
m = n = p = 100; // means: n = (m = (p = 100));
m = (n = p = 100) + 2; // means: m = (n = (p = 100)) + 2;
```

This is equally applicable to other forms of assignment. For example:

```
m = 100;
m += n = p = 10; // means: m = m + (n = p = 10);
```

Conditional Operator

The conditional operator takes three operands. It has the general form:

operand1? operand2: operand3

First *operand1* is evaluated, which is treated as a logical condition. If the result is nonzero then *operand2* is evaluated and its value is the final result. Otherwise, *operand3* is evaluated and its value is the final result. For example:

```
int m = 1, n = 2;
int min = (m < n ? m : n); // min receives 1
```

Note that of the second and the third operands of the conditional operator only one is evaluated. This may be significant when one or both contain side-effects (i.e., their evaluation causes a change to the value of a variable). For example, in

```
int min = (m < n ? m++ : n++);
```

m is incremented because m++ is evaluated but n is not incremented because n++ is not evaluated. Because a conditional operation is itself an expression, it may be used as an operand of another conditional operation, that is, conditional expressions may be nested. For example:

```
int m = 1, n = 2, p =3;
int min = (m < n ? (m < p ? m : p): (n < p ? n : p));
```

Comma Operator

Multiple expressions can be combined into one expression using the comma operator. The comma operator takes two operands. It first evaluates the left operand and then

the right operand, and returns the value of the latter as the final outcome. For example:

```
int m, n, min;
int mCount = 0, nCount = 0;
//...
min = (m < n ? mCount++, m : nCount++, n);
```

Here when m is less than n, mCount++ is evaluated and the value of m is stored in min. Otherwise, nCount++ is evaluated and the value of n is stored in min.

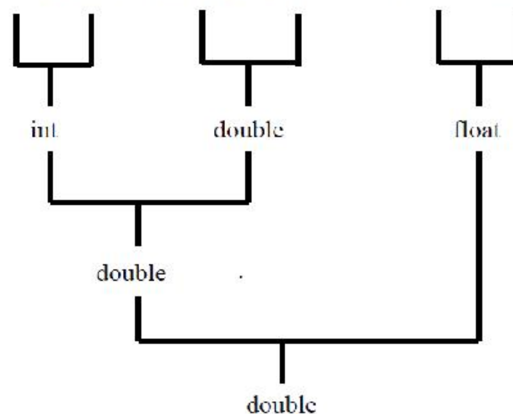
General Operator Precedence

The order in which operators are evaluated in an expression is significant and is determined by precedence rules. These rules divide the C++ operators into a number of precedence levels (see Table below). Operators in higher levels take precedence over operators in lower levels.

<table>
<tr><th>Level</th><th colspan="6">Operator</th><th>Kind</th><th>Order</th></tr>
<tr><td>Highest</td><td>::</td><td></td><td></td><td></td><td></td><td></td><td>Unary</td><td>Both</td></tr>
<tr><td></td><td>()</td><td>[]</td><td>-></td><td>.</td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>+
-</td><td>++
--</td><td>!
~</td><td>*
&</td><td>new
delete</td><td>sizeof
()</td><td>Unary</td><td>Right to Left</td></tr>
<tr><td></td><td>->*</td><td>.*</td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>*</td><td>/</td><td>%</td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>+</td><td>-</td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td><<</td><td>>></td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td><</td><td><=</td><td>></td><td>>=</td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>==</td><td>!=</td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>&</td><td></td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>^</td><td></td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>|</td><td></td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>&&</td><td></td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>||</td><td></td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
<tr><td></td><td>? :</td><td></td><td></td><td></td><td></td><td></td><td>Ternary</td><td>Left to Right</td></tr>
<tr><td></td><td>=</td><td>+=
-=</td><td>*=
/=</td><td>^=
%=</td><td>&=
|=</td><td><<=
>>=</td><td>Binary</td><td>Right to Left</td></tr>
<tr><td>Lowest</td><td>,</td><td></td><td></td><td></td><td></td><td></td><td>Binary</td><td>Left to Right</td></tr>
</table>

Type Conversion:

Let data types as: Character **char**; Integer **int**; Floating point **float**; Double float point **double.**



The C++ Mathematical Functions

The C++ originally supported by the set of 22 math functions. In C++, the math functions require the header <math.h>. All angles are in radians.

sin(x)	cos(x)	tan(x)	asin(x)	acos(x)
atan(x)	atan2(x)	sinh(x)	cosh(x)	tanh(x)
ceil(x)	floor(x)	exp(x)	fabs(x)	log(x)
log10(x)	pow(x,p)	sqrt(x)	fmod()	frexp(x)

- abs() - returns the absolute value of integer parameter.
- acos() - returns the arc cosine of the argument. The range of the argument is from -1 to 1. Any argument outside this range will result in error.
- asin()-returns the arc sine of the argument. The range of the argument is from -1 to 1. Any argument outside the range will result in error.
- atan()- The function atan() returns the arc tangent of the argument.
- atan2()-returns the arc tangent of y/x. It accepts two arguments.
- ceil()- The function ceil() returns the smallest integer which is not less than the argument.
- cos() – The function cos() returns the cosine of the argument. The value of theargument must be in radians.
- cosh() –The function cosh() returns the hyperbolic cosine of the argument.
- exp()- The function exp() returns the exponential of the argument.
- fabs()- The function fabs() returns the absolute value of floating point parameter.
- floor()- The function floor() returns the largest integer which is not greater than the argument.
- log() – The function log() returns the natural logarithm of the argument. An error is encountered if the argument is negative or zero.
- log10()- The function log10() returns base 10 logarithm of the argument. An error occurs if the argument is negative or zero.
- pow()- The function pow() returns the base raised to the power.
- sin()- The function sin() returns the sine of the argument. The value of the argument must be in radians.
- sinh()- The function sinh() returns hyperbolic sine of the argument.
- sqrt()- The function sqrt() the square root of the argument. An error occurs if the value of the argument is negative.
- tan()- The function tan() returns the tangent of the argument. The value of the argument must be in radians.
- tanh()- The function returns the hyperbolic tangent of the argument.
- *fmod(x,y)* calculates x modulo y (the remainder f, where x = ay + f for some integer a, and 0 f < y).

```
/* fmod and fmodl example */
#include <iostream.h>
#include <math.h>
main()
{
   double x = 5.0, y = 2.0;
```

```
    double result;
    result = fmod(x,y);
    cout<<"The remainder of "<< x  <<"over"<< y <<"
 is"<< result;
  }
```

- *frexp* resolve the argument to fraction *m* (a double greater than or equal to 0.5 and less than 1) and the integer value *n*, such that *x* (the original double value) equals $m*2^n$. *frexp* stores *n* in the integer that exponent points to.

```
/* frexp and frexpl examples */
#include <math.h>
#include <iostream.h>
main()
{
   double fraction, number;
   int exponent;
   number = 8.0;
   fraction = frexp(number, &exponent);
   cout<<"The number %lf is "<< number;
   cout<<"%lf times two to the "<< fraction;
   cout<<"power of %d\n"<< exponent;
   }
```

Ex: Write a program in C++ language to test the mathematical functions.

Ans:

```
#include<iostream.h>
#include<math.h>
main()
{
float c,d,f,g,r,t,u,i,o,z,b=0.56,x=-1,y=0.33;
c=log10(1000);
d=log(1000);
f=sin(b);
g=asin(f);
r=floor(2.56);
t=ceil(2.56);
u=fabs(-78.6);
i=pow(2.71828,d);
o=sqrt(169);
z=atan(x/y);
cout<<"\t The log of base 10 for 1000 is = "<<c<<endl;
cout<<"\t The log of base 2 for 1000 is = "<<d<<endl;
cout<<"\t The sin of b in radian is = "<<f<<endl;
cout<<"\t The sin inverse of f is = "<<g<<endl;
cout<<"\t The floor of 2.56 is = "<<r<<endl;
cout<<"\t The ceil of 2.56 is = "<<t<<endl;
cout<<"\t The absolute of -78.6 is = "<<u<<endl;
cout<<"\t The power of d for 2.71828 is = "<<i<<endl;
cout<<"\t The square root of 169 is = "<<o<<endl;
cout<<"\t The tan inverse of arg(x/y) is = "<<z<<endl;
}
```

The result of this program is as:

The log of base 10 for 1000 is = 3

The log of base 2 for 1000 is = 6.90776
The sin of b in radian is = 0.531186
The sin inverse of f = 0.56
The floor of 2,56 is = 2
The ceil of 2.56 is = 3
The absolute of -78.6 is = 78.6
The power of d for 2.71828 is = 999.995
The square root of 169 is = 13
The tan inverse of arg(x/y) is = -1.25205

Example: w.p. to compute and print the distance between two point (x1,y1) and (x2,y2) in Cartesian coordinates where:

$$Dis\tan ce=\sqrt{(x1 \quad x2)^2 + (y1 \quad y2)^2}$$

Solution

```
# include <iostream.h>
# include <math.h>
main()
{
int x1,x2,y1,y2;
double distance;
cout <<"Input the value of x1,x2,y1,y2"<<endl;
cin >>x1>>x2>>y1>>y2;
distance=sqrt (pow ((x1-x2),2)+pow((y1,y2),2));
cout <<"The distance is "<<distance<<endl;
}
```

Example: w.p. to find the value of y from the expression below:

$$y=\frac{2x^2+5e^{3x}\sin 6b}{\sqrt{z}}$$

Solution

```
# include <iostream.h>
# include <math.h>
main()
{
int x,b,z;
double y;
cout <<"Input the value of x,b,z"<<endl;
cin >>x>>b>>z;
y=((2*pow(x,2))+5*exp(3*x)*sin(6*b))/sqrt(z);
cout <<"y= "<<y<<endl;
}
```

H.W: w. C++ expression for the following formula use functions from the math library described.

1. $(\sin x)^2 \times (\cos x)^2$.

2. $e^{\frac{1}{2}\sqrt{\tan \cos x}}$

3. $\dfrac{\left(\log\left(\dfrac{x^2}{1-x}\right)\right)}{\left(x^{5+x}\right)}$

4. $y = \sin x + e^{x|+x} + x^{1.5}$

Program Control Statements

Like many other procedural languages, C++ provides different forms of statements for different purposes. Declaration statements are used for defining variables. Assignment-like statements are used for simple, algebraic computations.

Branching statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition. Loop statements are used for specifying computations which need to be repeated until a certain logical condition is satisfied. Flow control statements are used to divert the execution path to another part of the program. We will discuss these in turn. A **simple** statement is a computation terminated by a semicolon. Variable definitions and semicolon-terminated expressions are examples:

```
int i; // declaration statement
++i; // this has a side-effect
double d = 10.5; // declaration statement
d + 5; // useless statement!
```

The last example represents a useless statement, because it has no side-effect (***d*** is added to 5 and the result is just discarded). The simplest statement is the null statement which consists of just a semicolon:

```
; // null statement
```

Although the null statement has no side-effect, it has some genuine uses. Multiple statements can be combined into a ***compound*** statement by enclosing them within braces. For example:

```
{ int min, i = 10, j = 20;
min = (i < j ? i : j);
cout << min << '\n';
}
```

Compound statements are useful in two ways: (i) they allow us to put multiple statements in places where otherwise only single statements are allowed, and (ii) they allow us to introduce a new **scope** in the program. A scope is a part of the program text within which a variable remains defined. For example, the scope of `min`, `i`, and `j` in the above example is from where they are defined till the closing brace of the compound statement. Outside the compound statement, these variables are not defined.

1-Selection Statements

C++ language is supported by two types of selection statements: **if** and **switch**.

A- if statements

The general form of an if statement is

```
if (condition) statement;
```

First *expression* is evaluated. If the outcome is nonzero then *statement* is executed. Otherwise, nothing happens.
For example, when dividing two values, we may want to check that the denominator is nonzero:

```
if (count != 0)
average = sum / count;
```

To make multiple statements dependent on the same condition, we can use a compound statement:

```
if (balance > 0) {
interest = balance * creditRate;
balance += interest;
}
```

Example: Write a program in C++ language to print on the screen of computer the sentence (I am Iraqi) when the number enters from keyboard is equal 100.
Solution:

```
#include<iostream.h>
main()
{
int a;
cout<<" Enter integer number:";
cin>>a;
if(a==100) cout<<" I am Iraqi";
}
```

Example: w.p. to read two integers, use *if* statement to compare between the integers if they are equal or not .
Solution:

```
# include <iostream.h>
main ( ) {
int num1, num2;
cout << "Enter two integers" <<"\n";
cin>>num1>>num2;
if (num1==num2) cout <<num1<<"is equal to "<<num2<<endl;
if (num1!=num2) cout <<num1<<" is not equal
to"<<num2<<endl;
}
```

A variant form of the `if` statement allows us to specify two alternative statements: one which is executed if a condition is satisfied and one which is executed if the condition is not satisfied. This is called the `if-else` statement and has the general form:

```
if (expression) Statement1;
else Statement2;
```

First *expression* is evaluated. If outcome is nonzero then *statement1* is executed. Otherwise, *statement2* is executed.

```
if (balance >0)
interest =balance*creditRate;
else
```

```
interest = balance*deditRate;
balance +=interest;
```

Example: We can solve the above example by using the if-else statement.
Solution:

```
# include <iostream.h>
main ( ) {
int num1, num2;
cout << "Enter two integers" <<"\n";
cin>>num1>>num2;
if (num1==num2) cout <<num1<<"is equal to "<<num2<<endl;
else
cout <<num1<<" is not equal to"<<num2<<endl;
}
```

Or by using conditional operator

```
#include <iostream.h>
main ( )
{
int x,y;
cout <<"Enter two integers\n";
cin>>x>>y;
cout<<(x==y ? "They are equal\n":"They are not equal\n");
}
```

Example: Write a program in C++ language to print on the screen of computer the word (OKEY) when the character entering from keyboard is ‘Y’ or ‘y’ otherwise print (NO).
Solution:

```
#include<iostream.h>
main()
{
char a;
cout<<" Enter the character: ";
cin>>a;
if('Y'==a||'y'==a) cout<<"OKEY";
else cout<<"NO";}
```

Example: w.p. to test any number if it is a dividable of another number or not.
Solution:

```
# include <iostream.h>
main ( )
{
int a,b;
cout << "Enter two number" <<"\n";
cin>>a>>b;
if (a%b==0) cout <<a<<"is dividable of"<<b<<endl;
else cout <<a<<"is not dividable of"<<b<<endl;
}
```

Example: We can solve the above example by using the if statement.
Solution:

```
# include <iostream.h>
```

```
main ( )
{
int a,b;
cout << "Enter two number" <<"\n";
cin>>a>>b;
if (a%b==0) cout <<a<<"is dividable of"<<b<<endl;
if (a%b!=0) cout <<a<<"is not dividable of"<<b<<endl;
}
```

Or by using conditional operator

```
#include <iostream.h>
main ( )
{
int a,b;
cout <<"Enter two integers\n";
cin>>a>>b;
cout<<(a%b==0 ? "They are dividable \n":"They are not
dividable \n");
}
```

Example: w.p. to test if an enter integer number is even or odd and positive or negative.

Solution:

```
#include <iostream.h>
main ( )
{
int x;
cout <<"input any integer"<<endl;
cin >>x;
if (x>=0 && x%2==0)cout <<"The number is even and
positive"<<endl;
if (x<0 && x%2==0)cout <<"The number is even and
negative"<<endl;
if (x<0 && x%2!=0) cout <<"The number is odd and
negative"<<endl;
if (x>0 && x%2!=0) cout <<"The number is odd and
positive"<<endl;
}
```

The output will:

```
input any integer
-2
The number is even and negative
```

- The ***else-if*** statement is multiple-selection structure; it selects the action to perform from many different actions.

Example: W.P to read two integer numbers & compare between them if number1 > number2 print (The first number is greater) or number1 = number2 print (They are equal) or number2 > number1 print (The second is greater). Using else-if.

Solution:

```
#include <iostream.h>
main ( )
{
int a,b;
```

```
cout <<"Enter two integers\n";
cin>>a>>b;
if (a>b)cout <<"The first number is greater"<<endl;
else if(a==b)cout <<"They are equal"<<endl;
else cout <<"The second number is greater"<<endl;
}
```

B- switch statement

The switch statement is a multiway conditional statement generalizing the if-else statement. The general form of the switch statement is given by:

```
switch(expression) {
case constant1:statement sequence 1: break;
case constant2:statement sequence 2; break;
case constant3:statement sequence 3; break;
...
default: statement sequence;
}
```

Technically, the **break** statements inside the **switch** statement are optional. They terminate the statement sequence associated with each constant. If the **break** statement is omitted, execution continues on into the next **case**'s statements until either a **break** or the end of the **switch** is reached. You can think of the **case**s as labels. Execution starts at the label that matches and continues until a **break** statement is found, or the **switch** ends. Each statement sequence may be from one to several statements long. The **default** portion is optional. Both *expression* and the **case** constants must be integral types.

Note: the ***break*** statement is most importantly used within a switch statement, which can select among several cases. To interrupt the normal flow of control within a loop, the programmer can use the special statement:

```
break;
```

For example, suppose we have parsed a binary arithmetic operation into its three components and stored these in variables operator, operand1, and operand2. The following switch statement performs the operation and stored the result in result.

```
switch (operator) {
case '+': result = operand1 + operand2;break;
case '-': result = operand1 - operand2;break;
case '*': result = operand1 * operand2;break;
case '/': result = operand1 / operand2;break;
default: cout << "unknown operator: " << ch << '\n';
break;
}
```

Example: Write a program in C++ language to obtain and print on the screen of computer the (Next for N, Back for B, and Cancel for C) if the characters N. B, and C entering from keyboard.

Solution:

```
#include<iostream.h>
main()
{
char d;
cout<<"Enter the character: ";
cin>>d;
switch(d){
case 'C':cout<<" CANCEL ";break;
```

```
case ‘B’:cout<<" BACK";break;
case ‘N’:cout<<"NEXT";break;
default: cout<<"ERROR";
}
}
```

Example: Design the simple calculator program using (switch).

Solution

```
#include <iostream.h>
main ( )
{
char x;
int operand1,operand2;
cout<<"Input an arithmetic operator"<<endl;
cin>>x;
cout <<"Input two integer"<<endl;
cin>>operand1>>operand2;
switch (x)
{
case’+’: cout <<"The result is"<<operand1+operand2<<endl;
break;
case'-': cout <<"The result is"<<operand1-operand2<<endl;
break;
case’*’: cout <<"The result is"<<operand1*operand2<<endl;
break;
case’/’: cout <<"The result is"<<operand1/operand2<<endl;
break;
case’%’: cout <<"The result is"<<operand1%operand2<<endl;
break;
default:cout << "unknown character:" << x << endl;
} }
```

Example: Design the simple calculator program using (else if).

Solution

```
#include <iostream.h>
main ( )
{
char x;
int operand1,operand2 ;
cout<<"Input the Arithmetic operator"<<endl;
cin>>x;
cout <<"Input two integer"<<endl;
cin>>operand1>>operand2;
if (x == '+')
cout <<"The result is"<<operand1+operand2<<endl;
else if (x =='-')
cout <<"The result is"<<operand1-operand2<<endl;
else if (x =='*')
cout <<"The result is"<<operand1*operand2<<endl;
else if (x =='/')
cout <<"The result is"<<operand1/operand2<<endl;
else if (x =='%')
cout <<"The result is"<<operand1%operand2<<endl;
else
cout << "unknown character:" << x << endl;
}
```

Q: What happen if we remove the break from program. Try to execute the program without break statements.
Q: Write a program in C++ language to obtain and print on the screen of computer the name of day in a week if the input from keyboard the number of day in a week otherwise print ERROR.
Q: Write a program in C++ language to obtain and print on the screen of computer the name of month in a year when the number of month is input from keyboard, otherwise print ERROR.

2- Iteration Statements

C++ language is supported by two types of iteration (Loops) statements: **while** and **for**

A- while statement:

The while statement (also called **while loop**) provides a way of repeating anstatement while a condition holds. It is one of the three flavors of **iteration** in C++.The general form of the while statement is:

```
while (expression)
statement;
```

First *expression* (called the **loop condition**) is evaluated. If the outcome is nonzero then *statement* (called the **loop body**) is executed and the whole process is repeated. Otherwise, the loop is terminated.

Example: w.p to summation the numbers from 1 to 10.
Solution:

```
#include <iostream.h>
main ( )
{
int i=1;
int sum=0;
while(i<=10)
{
sum=sum+i;
i++;
}
cout <<"The sum is"<<sum<<endl; }
```

The output will be:

```
The sum is55
```

Example: Write a program in C++ language to obtain and print on the screen of computer the cubic of the number that entering from keyboard and terminate it with the number is less than –56.5.
Solution:

```
#include<iostream.h>
main()
{
float d;
cout<<"Enter the float number to obtain the cubic,
terminate with less -56.5\n\t";
cin>>d;
while(d>-56.5){
cout<< "The cubic of "<<d<<"is equal
"<<(d*d*d)<<endl<<'\t';
cin>>d;
}
```

```
cout<<"The number is less than -56.5";
}
```

Q: Rewrite the above program to obtain the root of the number and terminate with zero.

Q: Write a program in C++ language to print on the screen of computer the number entering from keyboard is accepted to divide by 5 without carry, and the result. Otherwise is terminated.

The do statement (also called **do loop**) is similar to the while statement, except which test the loop condition at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop always executes at least once. The general form of the **do-while** loop is

```
do {
statement sequence
} while(condition);
```

first, statement is executed, and then condition is evaluated. If it is true, control passes back to the beginning of the do statement, and the process repeat itself. When the value of condition is false, control passes to the next statement.

Example: Write a program in C++ language to obtain and print on the screen of computer the factorial of the number entering from keyboard using do…while statement.

***Solution*:**

```
#include<iostream.h>
main()
{
int n,f=1;
cout<<"Enter the positive integer number ";
cin>>n;
cout<<"The factorial of "<<n<<" is ";
do{
f*=n;
n--;
}while(n>1);
cout<<f<<endl; }
```

Q: Write a program in C++ language to print on the screen of computer the counter from 2.1 to 13.1 increments by 0.2 using do…while statement.

Q: Write a program in C++ language to find and print on the screen of computer the times of divided by 2 without fraction for number enter from keyboard using do…while statement.

B- for statement:

The general design of the **for** loop is reflected in some form or another in all procedural Programming languages. However, in C++, it provides unexpected flexibility and power. The general form of the **for** statement is

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

The **for** loop allows many variants, but there are three main parts:

1. The *initialization* is usually an assignment statement that sets the loop control variable.
2. The *condition* is a relational expression that determines when the loop exits.
3. The *update* defines how the loop control variable changes each time the loop is repeated. These three major sections must be separated by semicolons. The **for** loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the **for**.

Example: Write a program in C++ language to print on the screen of computer the numbers from 1 to 50 increments by1.
***Solution*:**

```
#include<iostream.h>
main()
{
for(int I=1;I<=10;I++) cout<<I<<'\t';
}
```

The output will be:

```
1    2    3    4    5    6    7    8    9    10
```

Example: w.p using **for** structure to sum all the even integers from 2 to 10.
Solution:

```
#include <iostream.h>
main ( )
{
int sum=0;
for (int number=2 ; number<=10; number+=2)
sum+=number;
cout <<"The sum of even number from 2 to 10 is
"<<sum<<endl;
}
```

The output will be:

```
The sum of even number from 2 to 10 is 30
```

Q: Write a program in C++ language to print on the screen of computer the numbers obtain from 2i where the i is integer number enter from keyboard.
Q: Write a program in C++ language to execute and print counter. However, the beginning and ending and update number entering from keyboard.

One of the most interesting uses of the **for** loop is to create an ***infinite loop***. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty, as here:

```
for(;;) statement;
```

Example: Write a program in C++ language to print the message ”you are in outside of loop” when you enter the right password, otherwise print “you are inside the loop” take one character as the password.
Solution:

```
#include<iostream.h>
main()
{
char f;
for(;;){
```

```
cout<<" Enter The Password:";
cin>>f;
if('D'==f)break;
cout<<"you are in loop:\n";
}
cout<<"you are in outside of loop";
}
```

Q: Rewrite the above program for the password is consists of five equal number.
Q: Rewrite the above program for three times trying otherwise print"The system is closed"
Q: Rewrite the above program for the password is as string.

3- Jump Group

A- goto statement:

The goto keyword causes program execution to jump to the label specified in the goto statement. The general form of goto is

```
goto label;
.
.
.
label: statement;
```

B- return statement:

The return statement enables a function to return a value to its caller. It has the general form:

```
return expression;
```

where *expression* denotes the value returned by the function. The type of this value should match the return type of the function. For a function whose return type is void, *expression* should be empty:

```
return;
```

The only function we have discussed so far is main, whose return type is always int. The return value of main is what the program returns to the operating system when it completes its execution. Under UNIX, for example, it its conventional to return 0 from main when the program executes without errors. Otherwise, a non-zero error code is returned. For example:

```
int main (void)
{
cout << "Hello World\n";
return 0;
}
```

When a function has a non-void return value (as in the above example), failing to return a value will result in a compiler warning. The actual return value will be undefined in this case (i.e., it will be whatever value which happens to be in its corresponding memory location at the time).

C- continue Statement:

The *continue* statement terminates the current iteration of a loop and instead jumps to the next iteration. It applies to the loop immediately enclosing the *continue* statement. It is an error to use the *continue* statement outside a loop.

```
do {
cin >> num;
if (num < 0) continue;
// process num here....
} while (num !=0);
```

It will read num rather than the num is zero.

Example: what will be the output of the following program?

```
#include <iostream.h>
main ( )
{
for (int x=1; x<=10;x++)
{
if (x==5)
continue;
cout <<x<<" ";
}
cout <<"\nUsed continue to skip printing the value 5
"<<endl;
}
```

The output will be:

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

D- break Statement:

A break statement may appear inside a loop (while, do, or for) or a switch statement. It causes a jump out of these constructs, and hence terminates the. Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it. It is an error to use the break statement outside a loop or a switch.

Example:

```
for (i=0; i < attempts; ++i)
{
cout << "please enter your password";
cin >> password;
if (verify(password)) // check password for correctness
break; // drop out of the loop
cout << "incorrect\n";
}
```

It will read a user password, but would like to allow the user a limited number of attempts, and we have assumed that there is function called `verify` which checks a password and returns true if it correct, and false otherwise.

Example: Write a program in C++ language to obtain and print on the screen of computer, when the number input from keyboard proves that:

First: Accept divided by 2 without carry to continue.

Second: Accept divided by 3 without carry to break and print “outside of loop”.

Third: Otherwise print “bottom of loop”.

Solution:

```
#include<iostream.h>
main()
{
int n;
for(;;){
```

```
cout<<"Enter integer number";
cin>>n;
if(n%2==0)continue;
if(n%3==0)break;
cout<<"bottom of loop";
}
cout<<"outside of loop";
}
```

Example: Write a program in C++ language to obtain and print on the screen of computer , method of using goto and return statements.

Solution:

```
#include<iostream.h>
main()
{
int n=5;
s1: cout<<" Now at step 1 with n="<<n<<endl;
--n;
if(n<2)return 0;
s2: cout << " Now at step 2 with n="<< n<<endl;
if(n<7) goto s4;
s3: cout<<" Now at step 3 with n="<<n<<endl;
if(n%2==0) goto s1;
s4: cout<<" Now at step 4 with n="<<n<<endl;
n-=2;
if(n>4) goto s1;
else goto s3;
}
```

The output will be:

Now at step 1 with n=5
Now at step 2 with n=4
Now at step 4 with n=4
Now at step 3 with n=2
Now at step 1 with n=2

Example: w.p. to read seven student marks then prints the number of pass and fail student.

Solution:

```
#include <iostream.h>
main()
{
int i=0,x,p=0,f=0;
cout <<"Input seven mark"<<endl;
again: cin >>x;
if (x>=50)
p+=1;
else f+=1;
i++;
if (i<7) goto again;
cout <<"The number of pass mark is "<<p<<endl;
cout <<"The number of fail mark is "<<f<<endl;
}
```

The output will be:

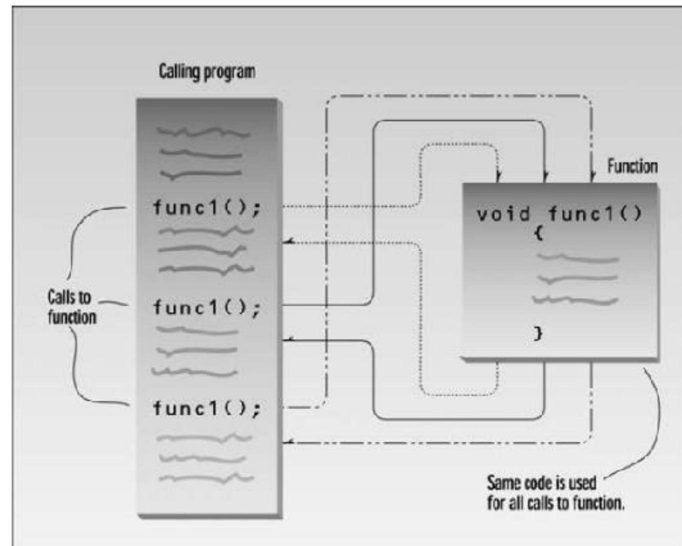
```
Input seven mark
```

```
66
25
95
66
80
70
56
The number of pass mark is6
The number of fail mark is1
```

Functions

A function provides a convenient way of packaging a computational recipe, so that it can be used as often as required. A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program as shown in the figure below. Two reasons to use functions:

1. To aid in the conceptual organization of a program. (However, object-oriented programming is more powerful).
2. To reduce program size.



To add a function to the C++ program, we need to add three components to the program:

- function declaration (prototype)
- calls to the function
- function definition.

The structure of program will be as follows:

```
#include <filename.h>
function declaration;
void main ( ) {
 ... ... call of function;
 ... ... call of function;
... ... }
type name (parameter1,parameter2,…….)  // function
definition
{
Statements// function body ...
... }
```

A **function definition** consists of two parts: interface and body. The **interface** of a function (also called its **prototype**) specifies how it may be used. It consists of three entities:

- The function **name**. This is simply a unique identifier.
- The function **parameters** (also called its **signature**). This is a set of zero or more typed identifiers used for passing values to and from the function.
- The function **return type**. This specifies the type of value the function returns.

A function which returns nothing should have the return type void. The **body** of a function contains the computational steps (statements) that comprise the function.

```
int addition (int a, int b)
{
int r;
r=a+b;
return (r);
}
```

Using a function **involves calling it**. A function call consists of the function name followed by the call operator brackets ( ), inside which zero or more comma-separated **arguments** appear. The number of arguments should match the number of function parameters. Each argument is an expression whose type should match the type of the corresponding parameter in the function interface.

```
main ( )
{
int z;
z=addition (5,3);
cout <<"The result is "<<z<<endl;
}
```

When a function call is executed, the arguments are first evaluated and their resulting values are assigned to the corresponding parameters. The function body is then executed. Finally, the function return value (if any) is passed to the caller. This example will be written as:

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

Scope of variable:

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variable a,b or r directly in function main since they were variables **local** to function addition. Also, it would have been impossible to use the variable z directly within function addition, since this was a variable local to the function main.

Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare **global** variables; these are visible from any point of the code, inside and outside all functions. In order to declare the variable outside any function or block; that means, directly in the body of the program. And here is another example about functions:

```
#include <iostream.h>
```

```
int subtraction (int a, int b)
{
int r;     // r is Local variable
r=a-b;
return (r);
}
main ( )
{
int x=5 , int y=3, z;  //  Global variable
z=subtraction (7,2);
cout <<"The first result is "<<z<<endl;
cout <<"The second result is "<< subtraction(9,2)<<'\n';
cout <<"The third result is "<< subtraction(x,y)<<'\n';
z=4+subtraction(x,y);
cout <<"The first result is "<<z<<endl;
}
```

The output will be:

```
The first result is 5
The second result is 7
The third result is 2
The first result is 6
```

The example shows a simple function which raises an integer to the power of another, positive integer.

```
1. int Power (int base, unsigned int exponent)
2. {
3. int result = 1;
4. for (int i=0; i < exponent; ++i)
5. result *= base
6. return result;
7. }
```

1. This line defines the function interface. It starts with return type of the function (int in this case). The function name appears next followed by its parameter list. Power has two parameters (base and exponent) which are of type int and unsigned int, respectively.

Note that the syntax for parameters is similar to the syntax for defining variables: type identifier followed by the parameter name. However, it is *not possible* to follow a type identifier with multiple comma-separated parameters:

```
int power (int base, exponent) // wrong
```

2. This brace marks the beginning of the function body.
3. This line is a local variable definition.
4,5. This for-loop raise base to the power of exponent and stores the outcome in result.
6. This line returns result as the return value of the function.
7. This brace marks the end of function body.

Example: Write a C++ program that computes the square of an integer number using the function **sqr()**.

```
#include<iostream>
```

```
int sqr(int);
main()
{
 int n;
cout<<"Enter an integer number: ";
cin>>n;
 cout<<"The square of "<< n <<" is "<<sqr(n)<<endl;
 }
 int sqr(int x)
{
 return (x*x);
}
```

Example: Write a C++ program that finds the maximum of two entered numbers using the function **max( )**.

Solution:

```
#include<iostream.h>
int max(int , int);
main() {
 int x , y;
cout<<"Enter two integer numbers: ";
cin >> x >> y;
cout<<"The maximum number is " << max(x,y) <<endl;
 }
int max(int a , int b)
{
if(a > b) return a;
 else return b; }
```

Example: Write a C++ program that finds the result of function:

$$S = \sum_{i=1}^{10} \frac{i\,!}{i}$$

Solution:

```
#include <iostream.h>
#include <math.h>
long int factorial(int a)
{
long int y=1;
for (int i = 0; i < a; ++i)
y*=x;
return y;
}
main ()
{
int i;
double sum;
for (i = 1; i < =10; ++i)
sum+=factorial(i)/i;
cout<< sum;}
```

The output will be:

409114

Recursive Functions:

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculates the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

n! = n * (n-1) * (n-2) * (n-3) ... * 1

more concretely, 5! (factorial of 5) would be:

5! = 5 * 4 * 3 * 2 * 1 = 120

and a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream.h>
long int factorial (long a)
{
if (a > 1)
return (a * factorial (a-1));
else
return (1);
}
 main ()
{
long int number;
cout << "Please inter a number: ";
cin >> number;
cout << number << "! = " <<
factorial (number);
}
```

The output will be:

```
Please inter a number: 9
9! = 362880
```

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 tasks. These overloaded functions have the same function name but with different argument lists (i.e. different number and/or different data types of arguments). An overloaded function appears to perform different activities depending on the kind of data sent (passed) to it. It performs one operation on one kind of data but another operation on a different kind.

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

Solution:

```
#include<iostream.h>
int area(int);
int area(int , int);
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); }
```