# UNIT - 1st

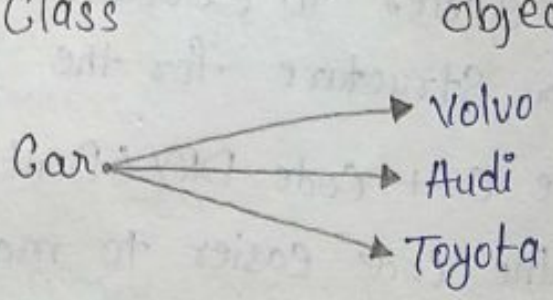
# Language Fundamental

## Advantages of OOP

* OOP is faster and easier to execute.
* OOP provides a clear structure for the program.
* OOP helps to keep the C++ code DRY (Dont Reapeat yourself) and makes the code easier to maintain, modify and debug.
* OOP makes it possible to Create full reusable applications with less code and shorter development time.
* Reuse to code through inheritance.
* Flexibility through polymorphism.
* Data Redundancy:- This is a Condition Created at the place of data storage (Data base) where the same piece of Data is held in two separate places. So the data redundancy is one of the greatest advantages of oops.
* Code maintenance:- This features is more of a necessary for any programming language, it helps user from doing re-work in many ways. It is always easy and time saving to maintain and modify the exiting codes with incorporating new changes into it.

# Charateristics of OOP

* Object :- Object is an instance of a class. When a class is defined, no memory is allocated but when it is an object is Created memory is allocated.




* Class :- A class in C++ is the data type which holds its own data member and member function, which can be accesed and used by Creating an object. A C++ class is like a blueprint for an object.

A class is user-defined data type which has data member and member function. Data member are the data variable and member function are the function used to manupulate these variable and together these data member and member function defines the property and behaviour of the object in a class.

Syntax :-
```
class classname//← User defined name
{
  Access Specifier; // access Specifier
  Data member; // Variable to be used
  member function( )// Method to access to
{                   // data member
}
};  // class name end with semicolon
```

* Abstraction:- Abstraction means displaying only essential information and hiding the detail

Data abstraction refer to providing only essential information about the data to the outside world, hiding the background details or implimentation.

* Encapsulation:- Encapsulation is a process of combining data member and function in a single unit called class. This is a to prevent the access to the data directly, the access to them is provided through the function of the class. It is one of the popular feature of oop that helps data hiding (public, private, protected).

* Inheritance:- The Capability of a class to derive properties and characteristics from another class is called inheritance.

Sub class:- The class that Inherits properties from another class is called Sub class or derived class.

Super class:- The class whose property are inherited by Sub class is called Base class or Super class.

* Polymorphism:- Polymorphism means "Many forms" and it occur when we have many Classed that are related to each other by Inheritance.

# * INTRODUCTION *

- Compilied type language.
- C++ is machine independent.
- Speedy execution are possible in C++.
- Object oriented with support for classes and object.
- Support pointers which then leads to direct memory access.
- Case insensitive.
- Used in many gaming application.
- first choise in programming competitions because of speed and easier less complex syntax.

```
// A Simple C++ program

#include<iostream.h>// include header file
int main( )                  // main function
{
  Cout<<" C++ is better than C:\n");
  return 0;
}          // end of program
```

(//) This is C++ Comment Symbol (double Slash //)
This is Called Single line comment
Multiline Comments can be written as follow

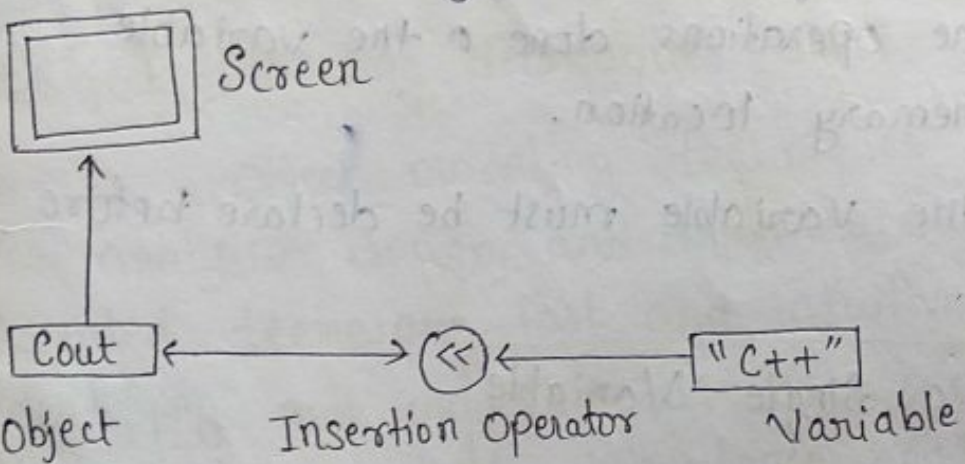
```
// This is a example of
// C++ program of illustrate
// Some of its features
```

* Output Operator

Cout << "C++ is better than C";

Causes the String in qoutation marks to be displayed on the Screen. This Statements introduces two new C++ features Cout and <<. The identifier Cout (pronounced as 'C out') is a predefined object that represent the Standard output Stream in C++.

The operator << is called the insertion or put to operator. It inserts the Contents of the variable on its right to the object on its left.




* The iostream file (header file)

We have used the following #include directives in the program

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

* Directive:- This directive causes the preproccesor to add the content of the istream file to the program. It Contain declaration for the identifier Cout and the operator <<.

The header file iostream should be included at the begining of all program that use input / output statement. Note that the naming conventions for header files may vary.

* Variable :- A variable is a name given to a memary location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memary location all the operations done o the variable effect that memary location.
- In C++ all the variable must be declare before use.

```
// Declaring a Single Variable
Data-type Variable_name;
```

* Input Operator

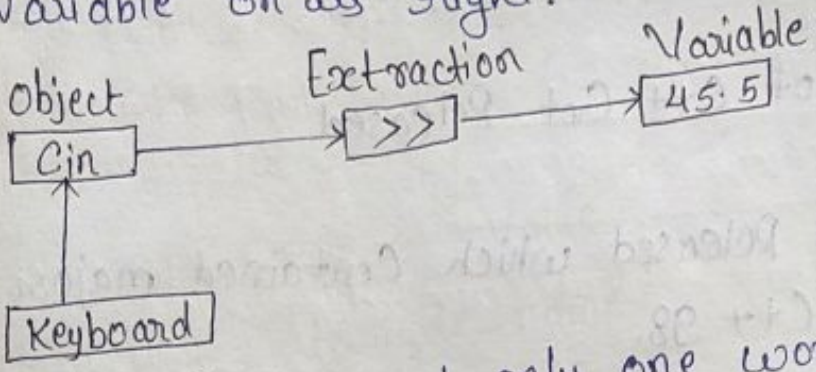
The Statement

```
Cin >> number_1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier Cin (pronounced 'c in') is a predefined object in C++ that corresponds to the standard input

Stream. Here this Stream represent the Keyboard.

The operator >> is known as extraction or get from another. It extracts (or takes) the value from the keyboard and assigned it to the variable on its right.




Note:- 'Cin' can read only one word and therefore we cannot use names with blank spaces.

* Object Modeling Techniques (OMT)

Object modeling techniques is a method for analysis, design and implimentation by an object Oriented technique. Fast and intuitive approach for identifying and modeling all object making up a system. Class attributes methods, inheritance and association can be expressed easily. Dynamic behavior of the object can be described by using the OMT dynamic model. Detailed Specification of State transitions and their descriptions within a system.

* History of C++

1979 → C with classes was first implemented

↓

1982 → C with classes was reference manual published

↓

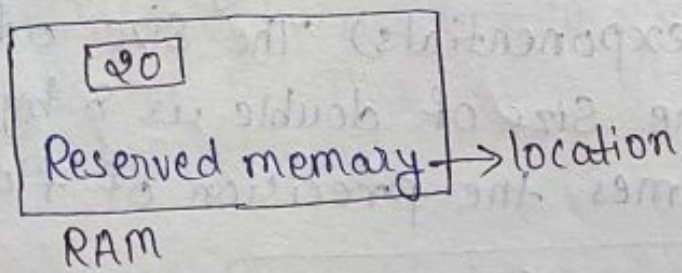
1984 → New official name C++ was given

1985 → Commercially Released - The first Edition
↓
1989 → Release of C++ 2.0
↓
1997 → C++ 98 ISO Standard Release
↓
1998 → 3rd edition of C++ Got Released
↓
2003 → C++ 0.3 Got Released which Contained major bugfixed of C++ 98.
↓
2009 → C++ 0X (New C++ // Standard and many Compilers now provide Support for Some of the Core C++ 11) get Released.
↓
2011 → C++ 11 got released which was a major revision.
↓
2014 → C++ 14 was released which was a Small improvement on C++ 11.
↓
2017 → C++ 17 released a major revision.
↓
2018 → C++ 20 is being developed.

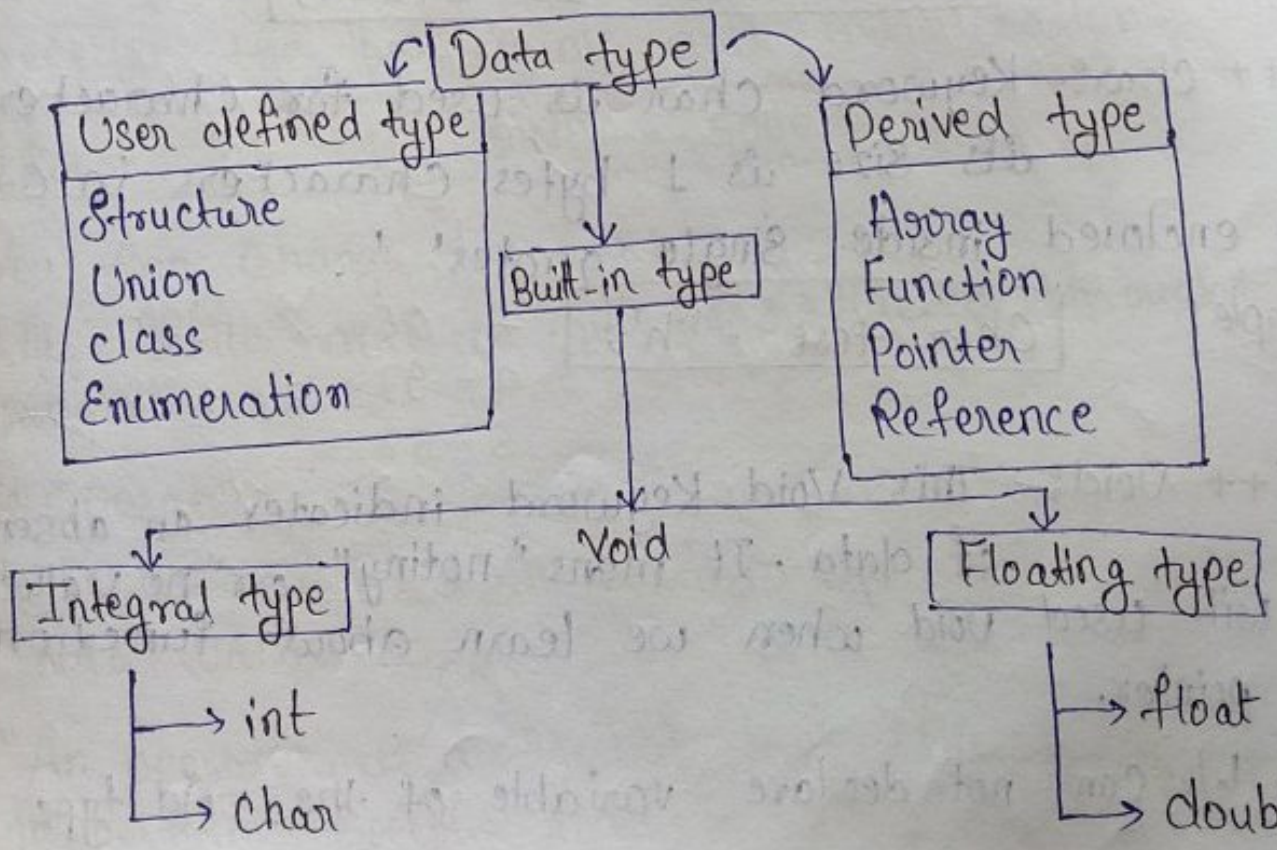
The C++ language is an object oriented language & is a Combination of both low-level & high-level language - a middle-level language. The programming language was Created, designed & developed by a Danish Computer Scientist - "Bjarne Stroustrup" at Bell telephone lab.

*Variable :- A Variable is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be changed during program execution. A Variable is only a name given to a memory location all the operation done on the variable effects that memory location.

```
int age = 20; ← Value
data type   Variable_name
```




* C++ Data type



In C++, data type are declarations for variable. This determines the type and size of data associated with variable.

1) C++ int :- The 'int' Keyword is used to indicate integer its size 4 bytes it can stores value - 2147483648 to 2147483647

Example:
```
int Salary = 85000;
```

2) C++ float and double :- Float and double are used to store floating-points number (decimal and exponentials) The size of float is 4 bytes and the size of double is 8 bytes. Hence double has two times the precision of float.

Example:
```
float area = 64.74;
double area = 134.64534;
```

3) C++ char :- Keyword char is used for characters. its size is 1 bytes characters in C++ are enclosed inside single quotes' '.

Example:
```
Char test = 'h';
```
A = 65, Z = 90
a = 97, z = 122
$ = 53

4) C++ void :- This 'Void Keyword indicates an absense of data. It means "notiny" or "no value" we will used void when we learn about function and pointer.

Note: We can not declare variable of the void type.

## * Constant in C++

A Constant like a Variable is a memory location where a value can be stored. Unlike variable, Constants never change in value you must initialize a Constant when it is Created. C++ has two type of Constant Literal and Symbolic.

A literal Constant is a value type directly into your program wherever it is needed.

Example:

```
long width = 5;
```

The Statement assign the integer variable width the value 5. This is a literal Constant value Can't be Changed.

A Symbolic Constant is Constant represented by a name. Just like a variable. The Constant keyword precedes the type. name and initialization.

```
Const int KILL_BONUS = 5000;
```

You Can Change the Constant KILL_BONUS and it will be reflacted throught the program.

## * Operator

"Operator are used to perform operations on variable and values". OR

"An operator is a Symbol that tells the Compiler to perform Specific mathemathical or logical mainpulations".

C++ divides the operators into the following groups:

- Arithmatic
- Assignment
- Comparison
- logical
- Bitwise operator

* Arithmatic operator :- Arithmatic operators are used to perform common mathematical operations.

| Operator | Name | Description | Example |
|---|---|---|---|
| + | Addition | Adds two values | X + Y |
| - | Substraction | Substract one value from another | X - Y |
| * | Multiplication | Multiplies two values | X * Y |
| / | Division | Divides one value from another | X / Y |
| % | Modulus | Return the division remainder | X % Y |
| ++ | Increment | Increases the value of a variable by 1 | ++X |
| -- | Decrement | Decreases the value of a variable by 1 | --X |

* Assignment Operator :- Assignment Operator are used to assign value to variable.

| Operator | Example | Same as |
|---|---|---|
| = | X = 5 | X = 5 |
| += | X += 3 | X = X+3 |
| -= | X -= 3 | X = X-3 |
| *= | X *= 3 | X = X*3 |
| /= | X /= 3 | X = X/3 |
| %= | X %= 3 | X = X%3 |
| &= | X &= 3 | X = X&3 |
| \|= | X \|= 3 | X = X\|3 |
| ^= | X ^= 3 | X = X^3 |
| >>= | X >>= 3 | X = X>>3 |
| <<= | X <<= 3 | X = X<<3 |

* Comparison Operator :- Comparison operator are used to Compare two value.

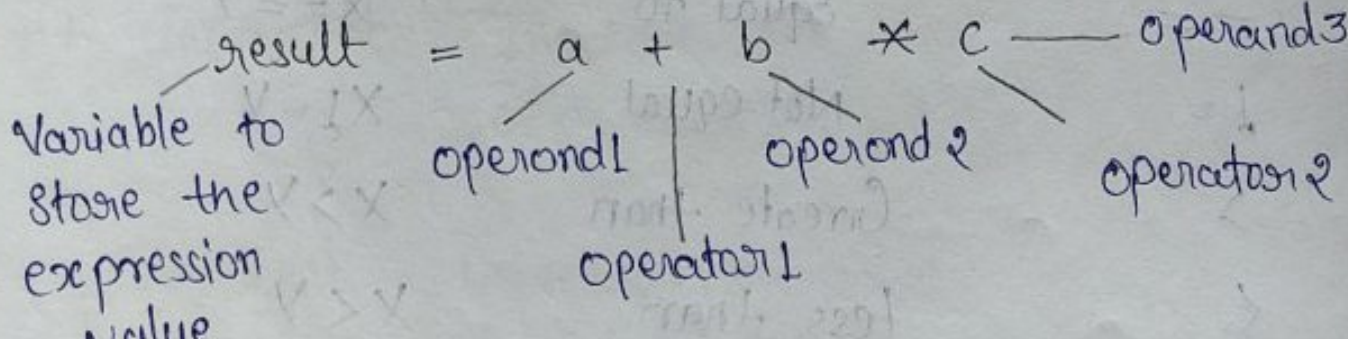
| Operator | Name | Example |
|---|---|---|
| == | Equal to | X == Y |
| != | Not equal | X != Y |
| > | Greate than | X > Y |
| < | less than | X < Y |
| >= | Greater than or equal to | X >= Y |
| <= | less than or equal to | X <= Y |

* Logical Operator :- Logical operator are used to determine the logic between variable or values.

| Operator | Name | Description | Example |
|---|---|---|---|
| && | logical and | Returns true if both Statement are true | x<5 && x<<2 |
| \|\| | logical or | Return true if one of the Statement is true | x<5 \|\| x<5 |
| ! | logical Not | Reverse the Result return false it the result is true | !(x<5 && x<10) |

* Expression in C++ :- An expression is a Combination of operator constant and variable. An expression may consist of one or more operend and zero or more operator to produce a value.

What is a expression



## Type of Expressions

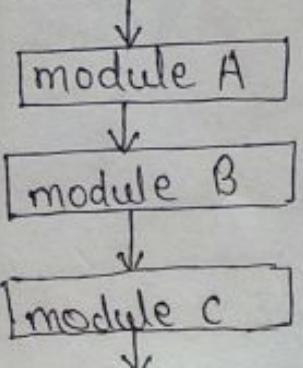
- → Integral expressions (X, X*Y, X+int(5.0))
- → Floating expressions (X+Y, 10.75)
- → Relational expressions (X<=Y, X+Y>2)
- → logical expressions (X>Y && X==10
- → Pointer expressions (&X, ptr, ptr++)
- → Bitwise expressions (X<<3, Y>>1)
- → Constant expressions (5, 10+5/6.0, 'X')

## * CONTROL STRUCTURES

Control structure are just a way to specify flow to control in program. Any algorithm or program can be more clear and understood if they use self-contained modulus called as logic or control structure It basically analyzes and chooses ~~structure~~ in which direction a program flow based on certain parameter or conditions.

There are three type are control structure.

1. Sequential logic :- Sequential logic as the name suggests follows as serial or sequential flow in which the flow in which the flow depend. on the series of instructions given to the computer. Unless new instruction are given



2. Selection logic (Conditional flow):- Selection logic simply involves a number of condition or parameter which decide one out of several written modulus.

1) Single Alternative (If)

If (condition) then:
[Module A]
[End of If Structure]

2) Double Alternative

If (Condition) then:
[Module A]
[End If Structure]

3) Multiple Alternative (If else)

If (condition). Then:
[Module A]
Else if (condition B) then:
[Module B]
Else if (condition N) then:
[Module N]
[END of Structure]

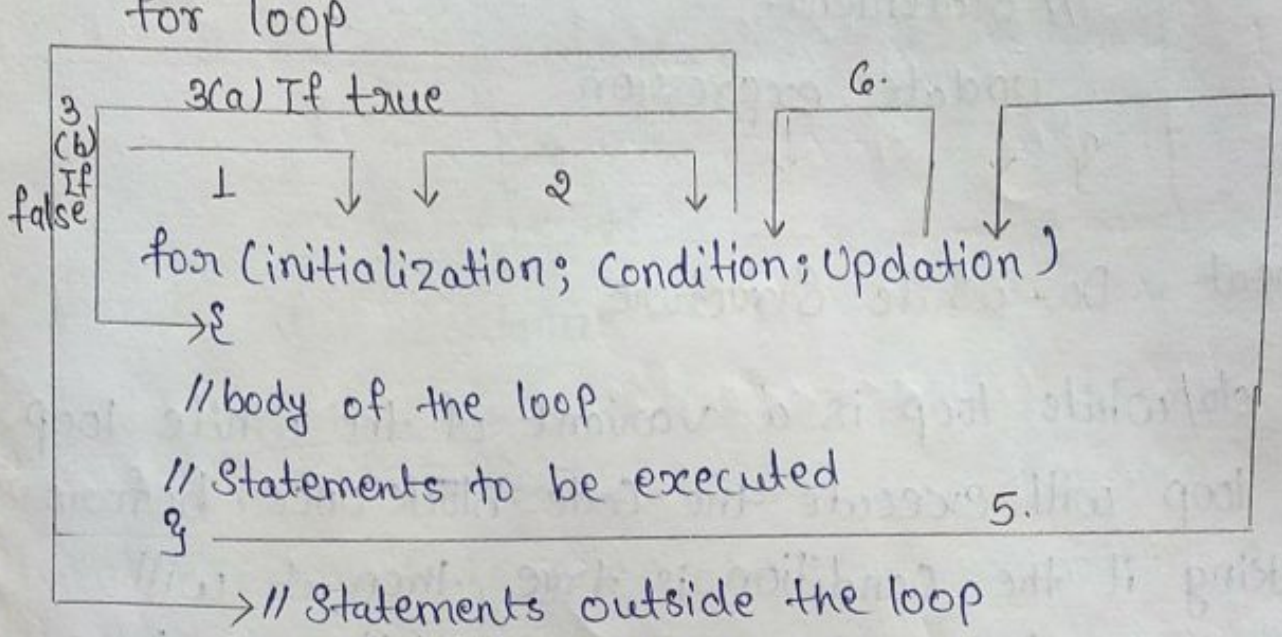
3. Iteration logic (Repetive flow):- The Iteration logic employee aloop which involves a repeat statement followed by a module known as the body of a loop.
There are two type:-

- Repeat - for Structure

This structure has the form

Repeat for i = A to N by I:
[Module]
[End of loop]

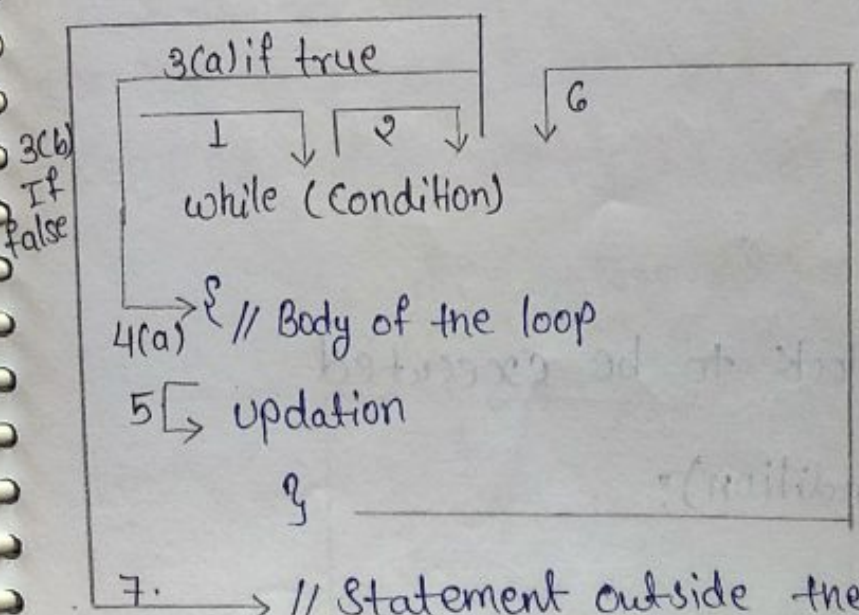
for loop is a repeatition control structure which allows us to write a loop that is executed a specific number of times.




Syntax:-

```
for (initialization ; condition ; updation)
   {
     //body of loop
     // Statement to be executed
   }
```

- Repeat - while Structure

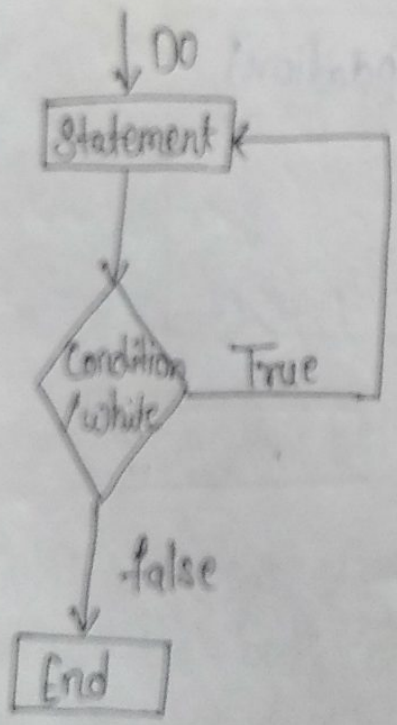


Syntax:-

```
while (test_expression
{
  // Statements
    update expression
}
```

- Repeat - Do-while Structure

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking it the condition is true, then it will repeat the loop as long as the condition is true.




Syntax:-

```
do{
  // Code block to be executed
}
  while (condition);
```

Example :-

```
int i = 0;
do {
    cout<<i<<"\n";
    i++;
  }
  while (i<5);
```

* Break and Continue

The break in c++ is a loop Control Statement which is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iteration stops there and control returns from the loop immediately to the first statement after the loop.

Syntax :-

```
break;
```

Basically break statement are used in the situation when we are not sure about the actual number of Iterations for the loop or we want to terminate the loop based on same condition.

Example :-

```
for (init; condition; update)
{
  if (condition to break)
  {
    break;
  }
  // code
}
Outside of loop
```

```
#include <iostream>
Using namespace std;
int main()
    for(int i=1; i<=5; i++)
    {
        if(i==3)
        {
            break;
        }
        Cout<<i<<endl;
    }
    return 0;
}
```

Output:- 1
2

* Continue:- In Computer programming, the Continue Statement is used to skip the Current iteration of the loop and the Control of the program goes to the next iteration.

Syntax:-

```
Continue;
```

Working:-

```
for(initalization; Condition; updation)
{
  // code
   if (condition to break) {
     Continue;
   }
    // code
}
```

```
#include<iostream>
Using namespace std;
int main()
    for (int i = 1; i<=5; i++)
    {
      // Condition to continue:
      if (i==3)
      {
        Continue;
      }
        Cout<<i<<endl;
      }
        return 0;
    }
```

output:- 1
2 // 3 is left
3
4

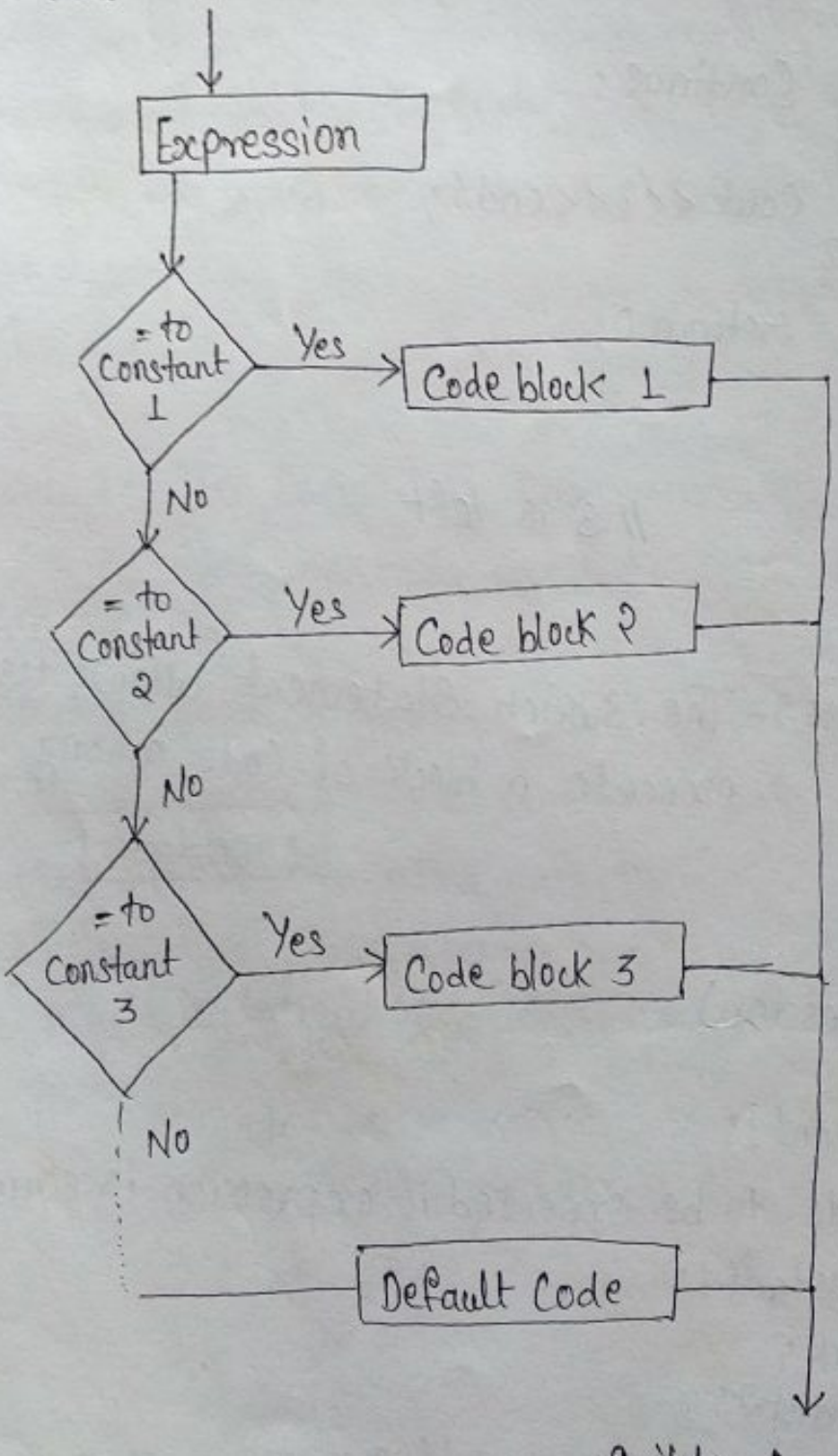
* Switch Case:- The Switch Statement allows us to execute a block of code among many alternative.

Syntax:-

```
Switch (expression)
{
  Case Constant1;
      // Code to be executed if expression is equal to
      // Constant1;
      break;
  Case Constant2;
      // Code to be executed if expression is equal to
      // Constant2;
```

```
break;
  .
  .
default:
// Code to be executed if expression doesnot match
        // any Constant
}
```

*flow chart*



Example:-

```
#include<iostream>
Using namespace std;
int main()
{
  char oper;
  float num1 , num2;
  Cout<<" Enter on operator (+, -, *, \):";
  Cin >> oper;
  Cout<<" Enter two number :"<<endl;
  Cin>>num1>>num2;
  Switch (oper)
{
  Case '+';
      Cout <<num1<<" +"<<num2<<"="
           <<num1+num2;
      break;
   Case'-';
      Cout<<num1<<"-"<<num2<<" =" <<num1 *num2;
      break;
   Case'\';
      Cout<<num1<<"\"<<num2<<"="<<num1\num2;
     default:
      Cout<<" Error! The Operator is not Correct";
      break;
   }
      return 0;
  }
```

Output :- Enter an operator (+, -, *, /):
Enter two number:
2.3
4.5
2.3+4.5=6.8

* GOTO Statement

The goto Statement is a jump Statement which is Sometimes also refered to as unconditional jump Statement. The goto Statement Can be used to used to jump from anywhere to anywhere with a function.

Syntax:-

| | |
|---|---|
| goto label;<br>. . . . . .<br>. . . . . .<br>. . . . . .<br>label: | label:<br>. . . . . .<br>. . . . . .<br>. . . . . .<br>goto label; |

In the above Syntax the first line tells the Compiler to go to or jump to the Statement marked as a label. Here label is a user-defined identifier which indicates the target Statement. The Statement immediatly followed after 'label:' is the destination Statement. The 'label:' on also appear before the 'goto label;' Statement in the above Syntax.

Program 1 *

```
#include<iostream>
Using namespace Std;
void checkEvenoroddd(int num)
{
  if (num%.z==0)
                  // jump to even
       goto even;
  else
       // jump to odd
      goto odd;
  even:
      Cout<<num<<" is even";
  odd:
      Cout<< num<<" is odd";
  int main()
  {
     int num=26;
     checkEvenoroddd(num);
     return 0;
  }
```

Output:-

26 is even

Program 2 *

```
#include<iostream>
  Using namespace std;
  Void printNumbers()
  {
     int n=1;
     label:
          Cout<<n<<" ";
          n++
          if(n<=10)
             goto label; // jump to label
  }
     int main()
  {
     printnumber();
     return 0;
  }
```

Output:-

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|

# UNIT - 2nd

# Structure, Function & Array

## Array

* Array:- In C++, an array is a variable that can store multiple values of the same type.

For example: Suppose a class has 27 students and we need to store the grades of all of them. Instead of creating 27 separate variable, we can simply create an array.

```
double grade[27];
```

data type — array name — element of same type

In C++ the size and type of arrays can't be changed after its declaration.

- Declaration-

```
data_type arrayname[array_size];
```

For example:-

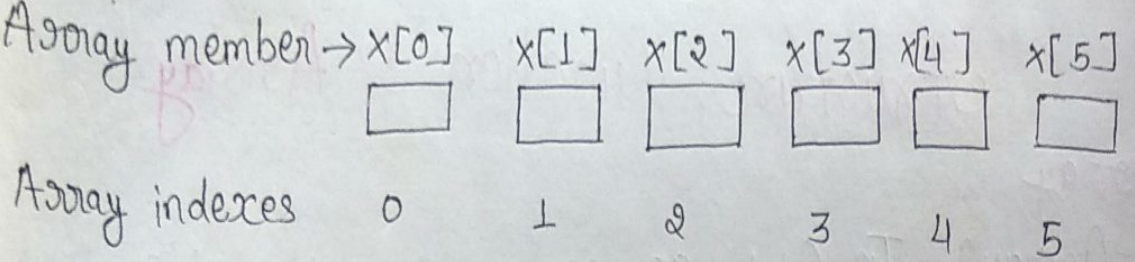
```
int x[6];
```

int - type of element to be stored.

x = name of the array

6 = size of the array

*Access element in C++ array:-




Each element in an array is associated with a number. The number is known as a array index. We can access element of an array by using those indices.

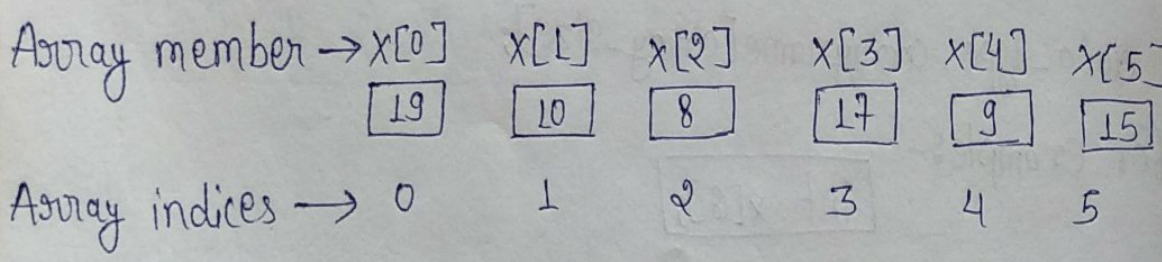
```
array[index]  // access array element
```

"If the size of an array is n the last element is stored at index (n-1) in this example, x[5] is last element. The array indices start with 0."

*Array initialization:- Its possible to initialize an array during declaration.

For example:-

```
int x[6] = {19, 10, 8, 17, 9, 15};
  // Declare and Initialize and array
```




C++ array and their data and her element.

Another method to initialize array during declaration

```
int x[] = {19, 10, 8, 17, 9, 15};
```

Here, we have not mentioned the size of the array. In such cases, the complier automatically computes

the size.

*C++ Array with empty member

In C++, if an array has a size n, we can store upto n number of element in the array. However, what will happen if we store less than n number of element.

For example:-

```
// Store only 3 element in the array
int x[6] = {9,10,8};
```

x[6] = {9,10,8};

| Array Member | x[0] | x[1] | x[2] | x[3] | x[4] | x[5] |
|---|---|---|---|---|---|---|
| | 9 | 10 | 8 | 0 | 0 | 0 |
| Array indices | 0 | 1 | 2 | 3 | 4 | 5 |

Entry array member are automatically assigned the value 0.

*How to insert and print array element?

```
int mark[5] = {19, 10, 8, 17, 9}
mark[3] = 9; // Change the element to 9
// take input from user
// Store the value at third position
  Cin >> mark[2];

// take the input from user;
// insert at ith position
  Cin >> mark[i-1];

// print first element of the array
  cout << mark[0];
```

```
// print ith element of the array
  Cout<< mark[i-1];
```

* Displaying array element

```
#Include<iostream>
Using namespace std;
int main()
{
  int number[5] = {7, 5, 6, 12, 35};
  Cout <<" The number are";
  //printing numbers are using range based for loop
   for (const int &n: numbers)
{
   Cout <<n<<"  ";
 }
  Cout<<" \n The number are:";
  //printing array element using
  // traditional for loop
   for (int i=0; i<5; i++)
{
   Cout << number[i]<<"  ";
 }
    return 0;
}
```

Output:-

The numbers are 7 5 6 12 35
The numbers are: 7 5 6 12 35

"Note - In our range based loop, we have used the code Const int &n instead of int n as the range declaration. However the Const int&n is more preferred because.

1. Using int n simply copies the array element to the variable n during each iteration. This is not memory - efficient.

&n, however user the memory address of the array address of the array elements to access their data without copying them to a new variable:

2. We are simply printing them (array element) not modifying them. Therefore we use const so as not to accidently change the values of the array.

A Take input From user and store them in an array

```
#include<iostream>
Using namespace std;
 int main()
{
  int number[5], int i;
  cout<<" enter the number:"<<endl;
  for (i=0; i<5; i++)
{
  cin >> number[i];
}
  cout<<" The number are";
  for(int n=0; n<5; n++)
{
  Cout <<" number[n]";
}
   return 0;
}
```

Output:-

Enter number:

11
12
13
14
15

The numbers are 11 12 13 14 15

## Function ☆

"A functions in a block of code that performs a specific task."

There are two type of function

1. Standard Library functions :- Predefined in C++
2. User defined function :- Created by users.

"C++ allows the programmer to define their own function."

"A user-defined function group code to perform a specific task and that group of code is given a name (identifier)."

When the function is involved from any part of the program, it all executes the codes defined in the body of the function.

Syntax to declare a function-

```
returntype functionname(parameter1, parameter2,.....)
{
  // Body of the function
}
```

Example:-

```
// function declaration
  void greet( )
{
  Cout<<"Hello world";
}
```

☆ greet() is a function name

☆ void is a return type

* {} is the function body is written inside

* Function Calling-

How we call the above greet() function.

```
int main()
{
   // Calling function
     greet();
}
```

Example:-

```
#include<iostream>
  void greet();<-------
{                      |
   // Code             | function
}                      | Call
   int main()          |
{                      |
   void greet();<------
}
```

```
#include<iostream>
 Using namespace std;
 void greet()
{
  Cout<<"Hello there";
}
   int main()
{
    greet();
    return 0;
}
```

Output:-

Hello there

* Function Parameter -

As mentioned above, a function can be declared with parameters (Argument). A parameter is a value that is passed when declaring a function.

```
void printnum(int num)
{
  cout << num;
}
```

function parameter

We pass the value to the function parameter while calling the function.

```
int main()
{
  int n = 7;
  printnum(n);
  return 0;
}
```

[ ∵ Calling the function n is passed to the function as - argument ]

* Function with parameter -

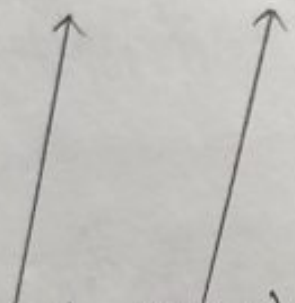
```
#Include<iostream>
Using namespace Std;
  //display a number
  void displaynum (intn1, doublen2)
{
  Cout <<" The int number is "<< n1;
  Cout <<" The double num is "<< n2;
}
  int main()
{
  int num1 = 5;
  double num2 = 6.6;
  //Calling the function.
```

```
    displaynum(num1, num2)
    return0;
}
```

Output:-

The int number is 5
The double number is 6.6

```
#include<iostream>
void displayNum(int n1, double n2)
{
  //code
}
int main()
{
  ....
  displayNum(num1, num2)
}
```



* Return Statement:- The return statement can be used to return a value from a fn.

Example:

```
int add(int a, int b)
{
  return(a+b);
}
```

Program:-

```
#include<iostream>
Using namespace std;
int add(int a, int b)
{
  return(a+b);
}
  int main()
```

```
{
    int sum;
    Sum = add (100, 78);
    Cout << " 100+78=" <<Sum << endl;
    return 0;
}
```

Output:-
100 + 78 = 178

* Function prototype

```
#include <iostream>
Using namespace std;
int add (int, int);
int main()
{
    int Sum;
    Sum = add (100, 78);
    Cout<<" 100 + 78=" << Sum <<endl;
    return 0;
}
int add (int a , int b)
{
    return (a+b);
}
```

Output:-
100 + 78 = 178

* C++ library function -

" Some Common library function in C++ are Sqrt()
, abs(), isdigit(), etc.

Program:-

```
#include <iostream>
# include< math >
Using namespace Std;
int main( )
{
   double number, SquareRoot;
   number = 250;
   SquareRoot = Sqrt(number);
   Cout<<" Square root of"<< number << "=" <<SquarRoot;
   return0;
}
```

Output:-

```
Squar root of 25 = 5
```

* Actual parameter and formal parameter.

" Actual parameter are the value that are passed to the function when it is invoked while formal argument (Parameter) are the variable defined by the function that recieves value when the function is called.

* Passing Argument to function

1. Call by Value:- The Call by value method of passing argument to a function copies the actual value of an argument into the formal parameters 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 argument.

Program:-

```
#include<iostream>
Using namespace std;
void Swap(int x, int y)
{
   int temp;
   temp = x;
    x = y;
    y = temp;
   return 0;
}
   int main()
{
    int a = 100;
    int b = 200;
   Cout<<" Before Swap, the value of a:"<<a<< endl;
   Cout<<" Before Swap, the value of b:"<< b<< endl;
   Swap(a,b);
   Cout<<"After the Swap, the value of a":<<a << endl;
   Cout<<" After the Swap, the value of b: << b<< endl;
   return 0;
}
```

Output:-

Before Swap, the value of a: 100
Before Swap, the value of b: 200
After Swap, the value of a: 100
After Swap the value of b: 200

* Call by reference:- The Call by reference method of passing argument to a function copies the reference of an argument into the formal parameter. Inside the function the reference is used to access the actual argument used in the Call. This means that change made to the parameter affect the passed argument.

To pass the value of reference argument reference is passed to the function just like any other value.

Program:-

```
#include<iostream>
Using namespace Std;
   void Swap(int&x,int&y)
{
   int temp;
   temp = x;
    x = y;
    y = temp;
    return;
}
   int main()
{
    int a= 100;
    int b = 200;
    Cout<<" Before Swap, the value of a":<<a<<endl;
    Cout <<" Before Swap, the value of b":<<b<<endl;

    Swap(a,b);
    Cout<<" After Swap, the value of a":<<a<< endl;
```

```
Cout<<" After Swap, the value of b": <<b<<endl;
return 0;
}
```

Output:-

Before Swap, the value of a: 100
Before Swap, the value of b: 200
After Swap, the value of a: 200
After Swap, the value of b: 100

* Reference Argument-

Inside the function, the reference 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 reference, argument reference is passed to the function just like any other value.

*Overloaded function-

Function overloading is a features in C++ where two or more function can have the same name but different parameter function overloading can be considered as an example of polymorphism features in C++.

```
int test() {}
int test (int a) { }
float test (double a) { }
int test (int a, double b) { }
```

Program:-

```
#include<iostream>
Using namespace std;
int over (int a, int b)
{
   return (a+b);
}
   float over (float a, int b)
{
   return (a+b);
}
   float over (int a)
{
   return (a);
}
   int main()
{
   Cout<<" Sum of int and int is -" <<over (10, 20) <<endl;
   Cout<<" Sum of float and int is-" <<over( 10, 50)<<endl
   Cout <<" Show single number"<< over (88) <<endl;
   return 0;
}
```

Output:-

Sum of int and int - 30
Sum of float and int - 60
Show Single number - 88

* Inline Function -

C++ inline function is powerfull Concept that is Commonly used with classes. If a function is inline, the Compiler places a copy of the code of that function at each point where the function is called at compile time.

Any changes to an inline function Could require all clients of the function to be recompiled because Compiler would need to replace all the code once again otherwise it will Continue with old functional inline is a Keyword.

```
inline return_type function_name(parameter);
```

Program:-

```
#include<iostream>
 Using namespace std;
 inline floatmul(float x, floaty)
{
   return(x*y);
 }
 inline double div(double p, doubleq)
{
   return(p/q);
 }
   int main()
{
    float a = 12.345;
    float b = 9.82;
```

```
    Cout<<"mul(a,b)"<<endl;
    Cout<<"div(a,b)"<<endl;
    return 0;
}
```

Output:-

```
121.228
1.25713
```

* Default Argument -

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.

Example of default argument -

```
#include<iostream.h>
Using namespace std;
// A function with default argument, it can be
// called with 20 argument or 3 argument or
// 4 argument
   int sum(int x, int y, int z=0, int w=0)
{
   return(x+y+z+w);
}
   int main()
{
   Cout<< sum(10,15)<<endl;
   Cout<< sum(10,15,25)<<endl;
   Cout<< sum(10,15,25,30)<<endl;
   return 0;
}
```

Output:-

25
50
80

## *STRUCTURE

A Structure is a user-defined data type in C++. A Structure Creates a data type that can be used to group items of possibly different type into a single type.

Structure keyword
↓
Struct class ← Structure tag
{
  char name[10];
  int id[5];
  float salary;
};

} Member or field of Structure

Syntax:-

Struct Structure_name
{
  member1:
  member2:
  .
  .
  membern:
};

Structure in C++ can contain two type of member-

- Data member: These members are normal C++ Variable. We can Create a Structure with variable of different data type in C++.
- Member function: These member are normal C++ functions. Along with variable, we can also include functions inside a Structure declaration.

Example:

```
// Data member
int roll;
int age;
int marks;

// Member function
Void printDetails()
{
Cout << " Roll = " << roll << " \n";
Cout << " age = " << age << " \n";
Cout << " marks = " << marks << " \n";
}
```

§ How to declare Structure Variable

```
// A variable declaration with Structure
// declaration
Struct Point
{ int x,y;
} P1; // The variable P1 is declared with 'Point'.

// A Variable declaration like basic data type.
Struct Point
{ int x,y;
};
int main()
{ Struct Point P1;
```

}

// Variable declared like a normal Variable.

Note- "In C++, the Struct keyword is optional before in declaration of a variable. In C, it is mandatory."

§ How to initialize Structure members?

Structure member cannot be initialized with declaration. For example the following C++ program fails in compilation But is considered Correct in C++ 11 and above.

```
Struct Point
{
   int x=0;  // ERROR;
   int y=0;  // ERROR;
```

The reason for above error is Simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variable are created.

§ How to access Structure element.

Structure member are accessed using dot (.) operator.

```
#include<iostream.h>
Using namespace std;
Struct Point
{
    int x, y;
}
    int main()
```

```
{
    Struct Point P1 = {0,1};
// Accessing member of point P1.
    P1.x = 20;
    Cout<<"x = "<< P1.x<<"y = "<< P1.y << endl;
    return 0;
}
```

Output :-

x = 20; y = 1;

" C allows programmers to pass a single or entire structure information to or from a function. A structure information can be passed as a function argument. The structure variable may be passed as a value or reference. The function will return the value by using the return statement."

// Passing Structure member to function.

```
#include<iostream.h>
#include<conio.h>
 int add (int, int)
     void main()
{
    Struct addition
    {
      int a, b;
      int c;
```

```
}
  Sum;
  Cout<<" Enter the value of a=";
  Cin>>Sum.a;
  Cout<<" Enter the value of b=";
  Cin>>Sum.b;
  Sum.C = add(Sum.a, Sum.b)
  Cout<<" The Sum of two no."<<Sum.C;
  getch();
}
  int add(int x, int y)
{
  int Sum1;
  Sum1= x+y;
  return(Sum1);
}
```

Output:- Enter the value of a = 34
Enter the value of b = 67

The Sum of two value is = 101

§ Passing the Entire Structure to function.

```
#include<iostream.h>
#include<conio.h>
  typedef Struct
{
    int a,b,c;
}
```

```
Sum;
void add(Sum);
Void main()
{
  Sum S1;
  Cout<<" Enter the value of a = "<<endl;
  Cin>> S1.a>>endl;
  Cout<<" Enter the value of b = "<<endl;
  Cin>> S1.b>>endl;
  add(S1);
  getch();
}
  void add(Sum x)
{
  int Sum1;
  Sum = x.a + x.b
  Cout<<" In the Sum of two values are: "<<Sum1;
}
```

Output:-

Enter the value of a = 34

Enter the value of b = 89

the Sum of two values are: 123

*Class:- A class in C++ is the building block that leads to object oriented programming. It is a user-defined data type, which holds its own data member and member functions, which can be accessed and used by creating an instance of the class. A C++ class is like a blueprint for an object.

- A class is a user defined data-type which has data member and member function.

- Data member are the data variables and member function are the functions user to mainpulate these variable and together these data member and member functions defines the properties and behavior of the object in a class.

*Object:- An object is an instance at a class. When a class is defined, no memory is allocated but when it is instantiated (Object is created) memory is allocated.

## Defining class and Declaring object

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
Class classname
{
  Access specifier: // Can be public, private, protected
  Data member; // Variable to used
  Member function(); // Method to access data member
};
```

Declaring object:- When a class is defined only the specification for the object is defined, no memory or storage is allocated. To use the data and access function defined in the class, you need to create object.

```
Class_name Object_name;
```

Accessing data member

Function: The data members and member function of class can be accessed using the dot(.) operator with the object. for example if the name of object is obj and you want to access the member function with the name print name() then you will have to write obj. print name().

## Accessing Data member

The public data member are also accessed in the same way given however the private data member are not allowed to be accessed directly

by the object. Accessing a data member depend soldy on the access Contral of that data member.

This access Contral is given by access modifiers. There are three access modifiers : public, private and protected.

* Defining Member Function:
- Outside of the class definition
- Inside the class definition

# Outside of the class definition:- Member functions that are declare inside a class have to be defined separately outside the class. Their definition are very much like the normal functions. They should have a function header and a function body.

The general form of a member function definition is

```
return_type class_name::function_name (argument)
{
  function body
}
```

" to define outside of the class we use scope resolution (::)"

# Inside the class definition :- Another method of defining a member function is to replace the function declaration by the actual function definition inside the class

Example:

```
Class item
{
    int number;
    float cost;
    public:
        void getdata(int a, int b);
        void putdata()
    {
        Cout << number << "\n";
        Cout << cost << "\n";
    }
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restriction and limitation that apply to an inline function are also applicable here. Normally only small functions are defined inside the class definition.

* Return Statement:- A Return Statement ends the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediatly following the call A return Statement can return a value to the calling function.

* Returing by Reference:

* Friend function: A friend function of a class is defined outside that class scope but it has the right to access all private and protected members of the class. Even though the prototype for friend function apper in the class definition, friends are not member functions.

Using the Keyword friend Compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with Keyword friend.

Declaration*

```
Class class_name
{
  friend data_type function_name(arguments);
};
```

The function definition does not use either the Keyword friend or Scope resolution operator.

Example:

```
#include<iostream.h>
#include<conio.h>
class xyz
{
  int num=100;
  char ch='z';
  public:
        friend void disp(xyz obj);
};
   void disp(xyz obj)
  {
    cout<<obj.num<<endl;
    cout<<obj.ch<<endl;
  }
   int main()
  {
    xyz obj;
    disp(obj);
    return 0;
  }
```

Output:

```
100
z
```

* Friend class: A friend class is a class that can access the private and protected members of a class in which it is declared as friend. This is needed when we what to allow a particular class to access the private and protected member of a class.

Declaration *

```
Class className
{
    friend returnType functionName(arguments);
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
 class xyz
{
  char ch='A';
  int num=11;
  public:
       friend class ABC;
};
  class ABC
{
  public:
       void disp(xyz obj)
       {
         Cout<<obj.ch<<endl;
         Cout<<obj.num<<endl;
       }
};
   int main()
   {
     ABC obj;
     xyz obj2;
     obj.disp(obj2);
```

return 0;
}

Output:

| |
|---|
| A |
| 11 |

* Static function: By declaring a function member as Static, you make it independent of any particular object of the class A Static member function can be called even if no objects of the class exist and the Static function are accessed using only the class name the Scope resolution operator.

A Static member function can only access Static data member. Static member function have a class Scope and they to not have access to the this pointer of the class you could use a Static member function to determine whether Same objects of the class have been created or not.

Static member variable program.

```
#include<iostream.h>
#include<conio.h>
  void test()
{
   Static int x=1;
   x=++x;
   int y=1;
   y=++y;
   Cout<<"x= "<< "\n";
   Cout<<"y= "<< "\n";
```

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

Output:

```
x = 2
y = 2
x = 3
y = 2
```

Using Class Static member variable.

```
#include<iostream.h>
#include<conio.h>
 class Example
{
   Static int x;
   public:
       void function( )
       {
         x++;
       }
       void function( )
       {
       Cout<<"x -"<<x<<" n";
       }
};
   int Example::x;
   int main( )
{
```

```
Example obj1, obj2, obj3;
Cout<<" intial value of x"<<"\n";
obj1.function2();
obj2.function2();
obj3.function2();
obj1.function1();
obj2.function1();
obj3.function1();
Cout<<" Value of after Calling function1"<<"\n";
obj1.function2();
obj2.function2();
obj3.function2();
return0;
}
```

Output:

Initial Value of x

```
X=0
X=0
X=0
```

Value of x after Calling function1

```
X=3
X=3
X=3
```

* Static member function program.

```
#include<iostream.h>
#include<conio.h>
  class example
{
    Static int Number;
```

```
int n;
Public:
    void set_n()
{
    n= ++Number;
}
    void show_n()
{
    Cout<<"Value of n = "<<n<<endl;
}
    Static void show_Number()
{
    Cout<<"Value of Number = "<<Number<<end l;
}
};
    int Example:: Number;
    int main()
{
    Example exa1, exa2;
    exa1.set_n();
    exa2.set_n();
    exa1.show_n();
    exa2.show_n();
    example:: show_Number();
    return 0;
}
```

Output:

Value of n=1
Value of n = 2
Value of Number = 2

# UNIT-3rd

# Object, Class and Inheritance

*Constructor: A Constructor in C++ is a Special method that is automatically called when an object of a class is created. To create a Constructor, use that the same name as the class.

Example:

```
#include<iostream.h>
#include<conio.h>
 Class class_name
{
   public:
       class_name() // Constructor
     {
       Cout<<" Hello world";
      }
};
     int main()
    {
      class_name obj ;
      return 0;
    }
```

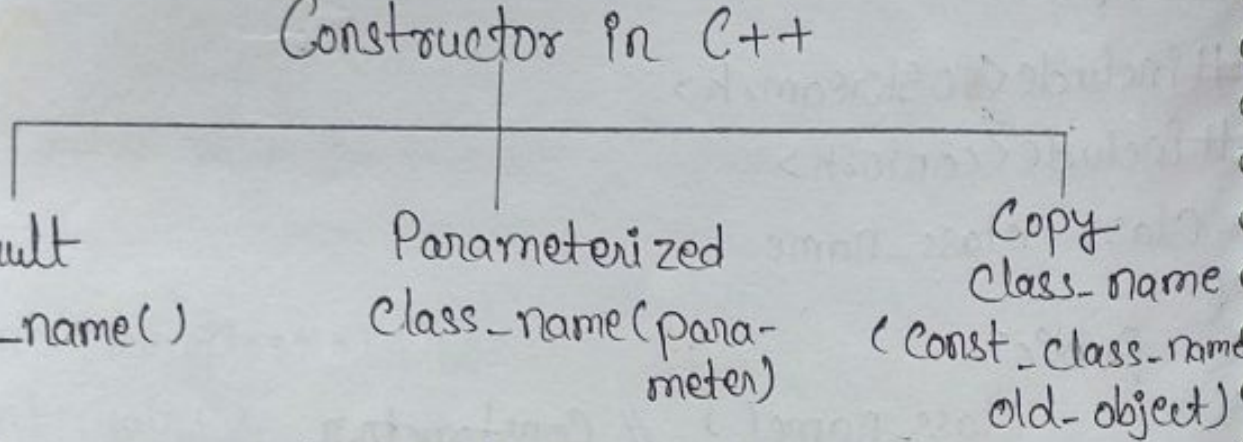
Output:

Hello world

Note - It is default Constructor

How Constructor are different from a normal member function.

* Constructor has same name as the class itself.
* Constructor don't have a return type.
* A Constructor is automatic call when object is Created.
* If we do not Specify a Constructor, C++ Compiler generates a default Constructor for us (expects no parameter and has an empty body).




1) Default Constructor: Default Constructor is the Constructor which does not take any argument. It has no parameter.

Example:

```
#include<iostream.h>
#include<conio.h>
 Class Construct
{
   public:
        int a,b;
        Construct // Default Constructor
    {
```

```
        a = 10;
        b = 20;
    }
};
    void main()
    {
        Construct c
            Cout<<" a:" <<c.a<<endl;
            Cout<<" b:" <<c.b<<endl;
            getch();
    }
```

Output:

a: 10
b: 20

2) Parameterized Constructor: It is possible to pass argument to Constructors. Typically, these argument help initialize an object when it Created. To Create a parameterized Constructor, Simply add parameters to it. The way you would to any other function when you define the Constructor's body use the parameter to initialize the object.

Example:

```
#include<iostream.h>
#include<conio.h>
 Class point
{
```

```
Private:
        int x, y;
Public:
        point (int x1, int y1)
    {
        x = x1;
        y = y1;
    }
        int getx()
    {
        return x;
    }
        int gety()
    {
        return y;
    }; }
        int main()
    {
        point p(10,50)
        Cout<<" P1.x :"<<P1.getx()<<" P1.y :"<<P1.gety();
        return 0;
    }
```

Output:

P1.x = 10, P1.y = 50

* Copy Constructor: A Constructor is a special type of member function that is called automatically when an object is created.

A Constructor has the same name as that of the class if does not have a return type.

The Copy Constructor in C++ is used to copy data of one object to another.

Example:

```
#include<iostream.h>
#include<conio.h>
  Class polar
{
        int radius, angle;
        public:
        polar (polar & c)
        {
         radius = c. radius;
         angle = c. angle;
        }
        Show( )
        {
        cout<<" radius:"<< radius<<endl;
        cout<<" angle:" <<angle;
        }
        }
        void main( )
        {
         polar
         pdar P1(P2);
         P3. Show( );
         getch( );
}
```

Output:

radius = 100

angle = 2100

We need to parametered constructor to access the copy constructor's value.

* Class Destructure: Destructor is a member function which destructor or delets an object.

Syntax: ~Constructure_name();

- Destructure function is automatically invoked when the object are destroyed.
- It cannot be declared static or constant.
- The destructure does not have argument, no return type not even void.
- An object of a class with a destructure cannot become a member of the union.
- A destructure should be declared in the public section of the class.
- The programmer cant's access the address of destructure.

A destructure function is called automatically when the object goes out of scope:

1. The function ends.
2. The program ends.

3. A block containing local variables ends.
4. A delet operator is called.

Program:

```
#include<iostream.h>
Using namespace std;
Class emp
{
    public:
    emp()
    {
    Cout <<" Constructor invoked" << endl;
    }
    ~emp()
    {
    Cout<<" Constructor invoked'<< endl;
    }
};
    void main()
    {
    emp e1;
    emp e2;
    getch();
}
```

Output:

Constructor invoked
Constructor invoked
Destructor invoked
Destructor invoked

*Object as a function argument:

The object of a class can be passed as argument to member functions as well as non member functions either by value or by reference.

Program:

```
#include<iostream.h>
 class weight{
 int kg;
 int g;
 public:
        void getdata();
        void putdata();
        void sum_weight(weight,weight);
};
        void weight::getdata()
        {
        cout<<"\n kilograms:";
        cin>>kg;
        cout<<"\n grams:";
        cin>>g;
        }
        void weight::putdata()
        {
        cout<<kg<<" kgs. and"<<g<<"grams.\n";
        }
        void weight::sum_weight(weight.w1,weight
                                .w2).
```

```
    g = w1.g + w2.g;
    Kg = g/1000;
    g = g%1000;
    kg = w1.kg + w2.kg;
}
int main()
weight w1, w2, w3;
cout<<" Enter weight in kilograms and grams \n";
Cout <<" \n Enter weight #1";
w1.getdata();
Cout<<" \n Enter weight #2";
w2.getdata();
w3.sum_weight(w1, w2);
Cout<<" \n weight #1 = ";
w1.putdata();
Cout<<" \n weight #2 = ");
w2.put data();
Cout <<" Total weight ";
w3.putdata();
return 0;
}
```

## * Struct Vs Class *

| S NO | Class | Structure |
|---|---|---|
| 1. | Classes are of reference types. | Structs are of value types. |
| 2. | All the reference types are allocated on heap memory. | All the value types are allocated on stack memory. |
| 3. | Allocation of large reference type is cheaper than allocation of large value type. | Allocation and deallocation is cheaper in value type as compare to reference type. |
| 4. | Class has limitless features. | Struct are used in small programs. |
| 5. | Class is generally used in large programs. | Struct are used in small programs. |
| 6. | Classes can contain constructor or destructor. | Structure does not contain parameter less constructor or destructor, but can contain parameterized constructor or static constructor. |
| 7. | Classes used new keyword for creating instances. | Struct can create an instance, with or without new keyword. |
| 8. | A class can inherit from another class | A Struct is not allowed to inherit from another Struct or class. |

| | | |
|---|---|---|
| 9. | The data member of a class can be protected. | The data member of struct can't be protected. |
| 10. | Function member of the class can be virtual or abstract. | Function member of the struct cannot be virtual or abstract. |
| 11. | Two variable of class can contain the reference of the same object and any operation on one variable can affect another variable. | Each variable in struct contain its own copy of data (except in ref and out parameter variable) and any operation on one variable can not effect another variable. |

* Array as class member:

Array can be declared as the members of a class. The array can be declared as private, public or protected members of the class.

Program:

```
#include<iostream.h>
#include<conio.h>
 const int size = 5;
 class Student
 {
        int roll_no;
        int marks[size];
        public:
```

```
    void getdata();
    void tot_marks();
};
void Student :: getdata()
{
cout<<"\n Enter roll no:";
cin>>roll_no;
for(int i=0; i<size; i++)
{
cout<<" Enter marks in Subject"<<(i+1)<<" ";
cin>>marks[i];
void Student :: tot_marks()
{
 int total=0;
 for(int i=0; i<size; i++)
{
 total+=marks[i];
 cout<<"\n\n total marks"<< total;
 }
  int main()
{
  Student Stu;
  Stu.getdata();
  Stu.tot_marks();
  return 0;
}
```

Output:

Enter roll no: LOL
Enter marks in Subject 1: 67
Enter marks in Subject 2: 67
Enter marks in Subject 3: 67
Enter marks in Subject 4: 67
Enter marks in Subject 5: 67
Total marks = 335

*Operator Overloading: In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex number, Fractional Number, Big Integer, etc.

Program:

```
#include<iostream.h>
 Using namespace std;
 class complex {
 private:
          int real, imag;
```

```
public:
    Complex (int r=0, int i=0)
    complex operator+ (Complex const &obj)
    {
    complex res;
    res.real = real + obj.real;
    res.img = imag + obj.imag;
    return res;
}
    void print()
    {
    cout << real << " + i" << imag << endl;
    }
};
    int main()
    {
    complex C1(10,5), C2(2,4);
    complex C3 = C1 + C2;
    C3.print();
    }
```

Output:

12 + i9

# Inheritance:

The capability of a class to derive properties and characteristics from another class is called inheritance. Inheritance is one of the most important feature of object oriented programming.

Sub class (Derived class):- The class that inherits properties from another class is called Sub class or Derived class.

Super class (Base class):- The class whose properties are inherited by sub class is called Base class or Super class.

To inherit from a class, use the : symbol.

```
// Base class
class vehicle
{
public:
        String brand = "Ford";
        void honk()
        {
        Cout << "Tuut, tuut: \n";
        }
};
// Derived class
class Car: public vehicle
{
 String model = "mustang";
```

```
};
    int main()
    {
    Car mycar;
    mycar.honk();
    Cout<<mycar.brand<<"    "<<mycar.model;
    return 0;
}
```

Output:

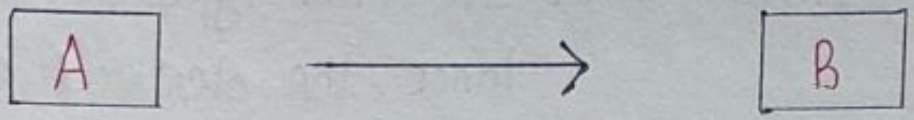
Tuut, tuut!
Ford Mustang

## Types of Inheritance

We have 5 different types of inheritance.

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance.

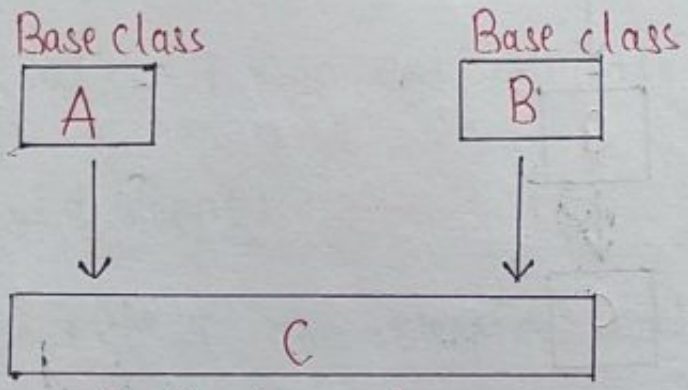
1. Single Inheritance: In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of inheritance.




Base class

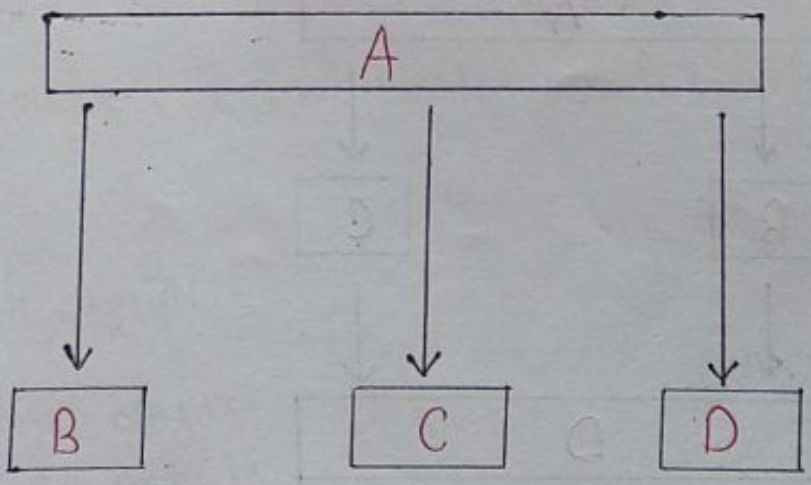
Derived Class

2. Multiple Inheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.

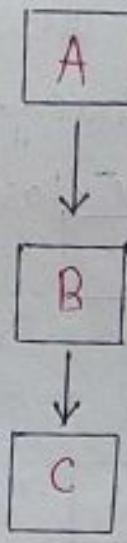



Derived Class

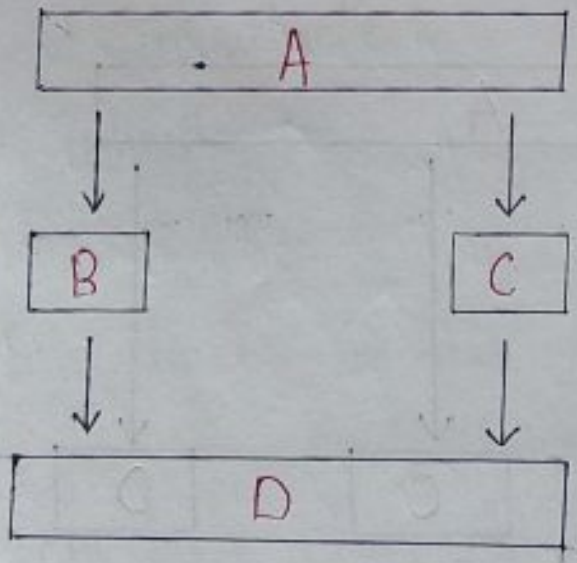
3. Hierarchical Inheritance: In this type of inheritance, multiple derived classes inherits from a single base class.



4. Multilevel Inheritance: In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The super class for one, is sub class for the other.




5. Hybrid Inheritance: Hybrid inheritance is combination of Hierarchical and multilevel inheritance.



# Simple example of Inheritance

```
#include<iostream.h>
#include<conio.h>
  class Animal //Base class
  {
    public:
          void eat()
          {
          cout<<" I can eat"<<endl;
          }
          void sleep()
          {
          cout<<" I can sleep<<" endl;
          }
  };
        class dog: public Animal //Derived class
        {
         public:
         void bark()
        {
        cout<<" I can bark: woof woof"<<endl;
        }
  };
      int main()
      {
       dog dog1;
       dog1.eat();
       dog1.sleep();
```

```
    dog1.bark();
    return 0;
}
```

Output:

```
I can eat
I can sleep
I can bark.
```

## * Access Modifiers OR Access Specifiers *

Access modifiers or access specifiers in a class are used to a sign the accessibility to the class member. That is it sets some restrictions on the class member not to get directly accessed by the outside function.

There are 3 types of access modifiers available in C++:

1. Public 2. Private 3. Protected

1. Public: All the class member declared under the public specifier will be available to everyone. The data member and member function declared as public can be accessed by other classes and functions too. The public member at a class can be accessed from anywhere in the program using the direct member access operator (.) with the

Object of that class.

Program:

```
#include<iostream.h>
#include <conio.h>
  class circle
  {
    public:
        double radius;
        double compute_area()
        {
         return 3.14 * radius * radius;
        }
  };
    void main() {
    circle obj;
    obj . radius = 5.5;
    Cout << " Radius is :" << obj . radius << " \n";
    Cout << " Area is " << obj . Compute_area() << " \n";
    return 0;
  }
```

Output:

Radius is : 5.5

Area is : 94.985

2. Private: The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member function or the function are allowed to access the private member of a class.

Program:

```
#include<iostream.h>
#include<conio.h>
  class circle
{
  private:
        double radius;
  public:
        void compute_area(double r)
        {
        radius=r;
        double area = 3.14 * radius * radius;
       cout<<"Radius is :"<< radius <<endl;
       Cout<<"Area is :" << area;
       }
};
       void main()
      {
       circle obj;
```

```
    obj.compute_area(1.5);
    getch();
}
```

Output:

Radius is : 1.5

Area is : 7.065

3. Protected: Protected access specifiers is similar to private access modifiers in the sense that it can't accessed outside of its class unless with the help of friend class, the difference is that the class member declared as protected can be accessed by any subclass (derived class) of that class as well.

Program:

```
#include<iostream.h>
#include<conio.h>
  class Circle
{
  protected:
        int id_protected;
};
    Class child: public parent
  {
   void setId(int id)
   {
```

```
    id_protected = id;
  }
  void disp_id()
  {
  cout<<"id_protected is"<<id_protected;
  }
};  int main(){
  Circle c;
  c.setId(8L);
  c.disp.Id();
  return 0;
}
```

Output:

id_protected is : 8L

| Base Class member Access specifiers | Type of Inheritance | | |
|---|---|---|---|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | protected | Protected | Private |
| Private | Not accessible hidden | Not accessible hidden | Not accessible hidden |

# *Overriding*

As we know, inheritance is a feature of OOP that allows us to Create derived classes from a base class. The derived classes inherit features of the base class.

Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed.

This is known as function overriding in C++. The function in derived class overrides the function in base class.

Program:

```
#include<iostream.h>
#include<conio.h>
  class base
{
  public:
        void print()
        {
        cout<<" Base function"<<endl;
        }
};
   class Derived : public base
   {
```

```
public:
        void print()
        Cout<<" Derived function" <<endl;
        }
};

int main()
Derived derived1;
derived.print();
return 0;
}
```

Output:

Derived function

As we can see, the function was overridden because we called the function from an object of the derived class. Had we called the print() function from an object of the base class the function would not have been over-ridden.

# *Member function*

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them-

```
Class Box {
    public:
        double length;  // Length of a box
        double breadth; // Breadth of a box
        double height;  // Height of a box
        double getVolume(void); // Returns box volume
};
```

Member functions can be defined within the class definition or separately using scope resolution operator, :: Defining a member function within the class definition declares the fn inline, even if you do not use the inline specifier. So either you can define Volume() function as below-

```
Class Box
{
 public:
      double length;
      double breadth;
      double height;
      double getVolume(void)
      {
      return length * breadth * height;
      }
};
```

If you like, you can define the same function outside the class using the scope resolution operator (::) as follows:

```
double Box:: getVolume(void)
{
 return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator(.) on a object where it will manipulate data related to that object only as follows -

```
Box myBox;   // Create an object
myBox.getVolume();  // Call member fn for
                       the object
```

Let us put above concepts to set and get the value of different class members in a class –

Program:

```
#include<iostream.h>
Using namespace std;
Class Box
{
 public:
      double length;
      double breadth;
      double height;
    //Member fn declaration
      double getVolume(void);
      void setLength(double len);
      void setBreadth(double bre);
      void setHeight(double hei);
};
    // Member fn definitions
     double Box:: getVolume(void)
    {
```

```
    return length * breadth * height;
}
void Box :: SetLength(double len)
{
    length = len;
}
void Box :: setBreadth(double bre)
{
    breadth = bre;
}
void Box :: setHeight(double hei)
{
    height = hei;
}
// Main fn for the program
int main()
{
    Box Box1; // Declare Box1 of type Box
    Box Box2; // Declare Box2 of type Box
    double volume = 0.0; // Store the volume of
                         a box here

    // Box 1 Specification
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);
```

```
// Box 2 Specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

// Volume of box 1
volume = Box1.getVolume();
Cout<<"Volume of Box1 : "<<volume<<endl;

// Volume of box 2
volume = Box2.getVolume();
Cout<<"Volume of Box2 : "<<volume<<endl;
return 0;
}
```

Output:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```