OBJECT ORIENTED PROGRAMMING

UNIT-II

FUNCTIONS:

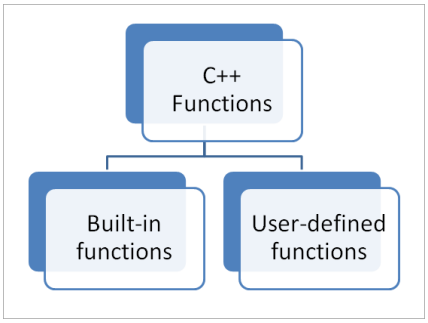
- A function is a group of statements that together perform a task.
- Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

Advantage of functions

There are the following advantages of C++ functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call functions any number of times in a program and from any place in a program.
- We can track a large program easily when it is divided into multiple functions.
- Reusability is the main achievement of functions.

Types of functions:



1.Built-in Functions

- Built-in functions are also called library functions. We can directly use these functions in our code.
- These functions are placed in the header files of C++.The functions which are declared in the C++ header files such as strlen, strcmp,ceil(x), cos(x), floor(x), etc.

2.User-Defined Functions

- C++ also allows its users to define their own functions. These are the user-defined functions.
- We can define the functions anywhere in the program and then call these functions from any part of the code

The general syntax for user-defined functions (or simply functions) is as given below:

return_type functionName(param1,param2,….param3)

{

Function body;

}

So as shown above, each function has:

- **Return type:** It is the value that the functions return to the calling function after performing a specific task.
- **functionName** : Identifier used to name a function.
- **Parameter List:** Denoted by param1, param2,…paramn in the above syntax. These are the arguments that are passed to the function when a function call is made. The parameter list is optional i.e. we can have functions that have no parameters.
- **Function body:** A group of statements that carry out a specific task.

As already mentioned, we need to 'declare' a function before using it.

Function Aspects

There are three aspects of a C++ function.

- **Function declaration**: A function must be declared globally in a C++ program to tell the compiler about the function name, function parameters, and return type.

- **Function call** : Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of parameters as it is declared in the function declaration.

- **Function definition** : It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

SN	C++ function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

Function Declaration
A function declaration tells the compiler about the return type of function, the number of parameters used by the function and its data types.

some examples of the function declaration below for your reference.
int sum(int, int); or int sum(int a, int b);

Above declaration is of a function ‘sum’ that takes two integers as parameters and returns an integer value.

void swap(int, int);

This means that the swap function takes two parameters of type int and does not return any value and hence the return type is void.

void display();

Function Definition
A function definition contains everything that a function declaration contains and additionally it also contains the body of the function enclosed in braces ({}).

```
void swap(int a, int b)

{

//statements;

}
```

Calling A Function

The function that calls another function is called the “Calling function”.

Eg: swap(4,3); or swap(a,b);

Example : program to find maximum value using functoin

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

int max(int num1, int num2);  // function declaration

void main ()
{
   int a = 100;
   int b = 200;
   int ret;

   ret = max(a, b);  // function call
   cout << "Max value is : " << ret << endl;

   getch();
}

int max(int num1, int num2)  // function definition
{

   if (num1 > num2)
      return num1;
   else
      return num2;

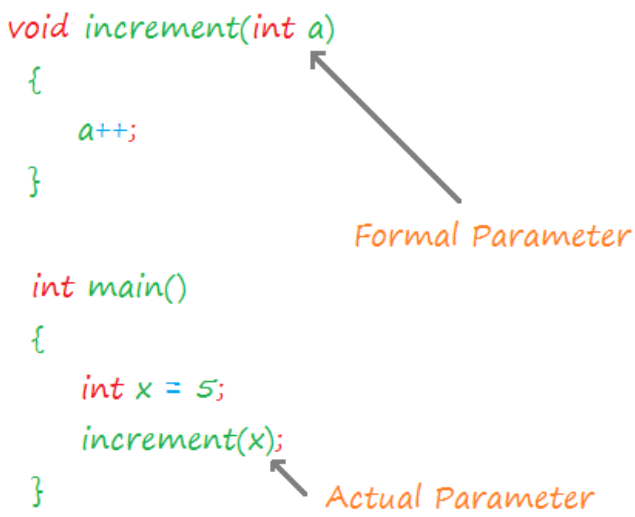

}
```

it would produce the following result −

```
Max value is : 200
```

Formal And Actual Parameters

- The parameters that are defined in the function definition are called **Formal Parameters**.
- The parameters in the function call which are the actual values are called **Actual Parameters.**



Call by Value

When a function is called in the call by value, the **value of the actual parameters is copied into formal parameters.**

Example: Program to swap two numbers using call by value

```
#include <iostream.h>
#include<conio.h>
 void swap(int a, int b) ;
void main()
{
int a,b;
cout<<"Enter the two numbers to be swapped: ";
cin>>a>>b;

cout<<"a = "<<a;
cout<<"\tb = "<<b;
swap(a,b);
getch();
  }
void swap(int a, int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
cout<<"\nAfter swapping inside Swap:\n ";
cout<<"a = "<<a;
```

```
cout<<"\tb = "<<b;
}
```

Output:

Enter the two numbers to be swapped: 3 2
a = 3 b = 2
After swapping inside Swap:
a = 2 b = 3
After swapping inside Main:
a = 3 b = 2

Call by Reference:

- In this technique, instead of passing copies of actual parameters, we pass references to actual parameters.
- In call by reference technique, we use these references of actual parameters and as a result, the changes made to formal parameters in the function are reflected back to the calling function.

Example: Program to swap two numbers using call by reference

```
#include <iostream.h>
#include<conio.h>
void swap(int &a, int &b) ;
void main()
{
int a,b;
cout<<"Enter the two numbers to be swapped: ";
cin>>a>>b;

cout<<"a = "<<a;
cout<<"\tb = "<<b;
swap(a,b);
getch();
  }

void swap(int &a, int &b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
cout<<"\nAfter swapping inside Swap:\n ";
```

```
cout<<"a = "<<a;
cout<<"\tb = "<<b;
}
```

Call by Address (Pointer)

- In C++, we can also pass parameters to function using pointer variables.
- This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call.
- The call by pointer technique produces the same results as that of pass by reference.
- **Example: Program to swap two numbers using call by address**
-

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

void swap(int *a, int *b);
void main()
{
int a,b;
cout<<"Enter the two numbers to be swapped: ";
cin>>a>>b;

cout<<"a = "<<a;
cout<<"\tb = "<<b;
swap(&a, &b);
getch();
  }
void swap(int *a, int *b)
{
     int  *temp;
    *temp = *a;
     *a = *b;
    *b = *temp;
cout<<"\nAfter swapping inside Swap:\n ";
cout<<"a = "<<*a;
cout<<"\tb = "<<*b;
}
```

Default Arguments (Parameters):

- 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.

- In this case, when we invoke the function, we don't specify parameters. Instead, the function takes the default parameters that are provided in the prototype.

Eg: 1. void display(int p,float a, float r=1.2); //function declration;

1. void display(int p, float a, float r) //function definition
   {
   Statements to be executed;
   }

2. display(20,3.1) // Function call

It is important to note that we must add default value from right to left; cannot provide value to a particular argument in the middle of an argument list

Ex:
```
int sum(int x, int y, int z=5, int w=0) // legal
  int sum(int x, int y=19, int z, int w=6)  //illegal
  int sum(int x=2, int y, int z, int w=2)  //illegal
  int sum(int x, int y=4, int z=0, int w=110)  //legal
```

The following Example demonstrates the use of Default Parameters.

```
#include <iostream>
#include <conio.h>
 int operate(int a, int b = 3, int c = 2);
void printline(char ch='*' , int len=45);
 void main()
 {
  int a,b,c;
  printline();
  cout<<"Enter values for a,b and c: "; cin>>a>>b>>c;
  cout<<endl;
  cout<<"Call to mathoperation with 1 arg : "<<operate(a);
  cout<<endl;
```

```
  cout<<"Call to mathoperation with 2 arg : "<<operate(a,b);
  cout<<endl;
  cout<<"Call to mathoperation with 3 arg : "<<operate(a,b,c);
  cout<<endl;
printline('=');
 }
int operate(int a, int b, int c)
{
   return ((a*b)/c);
 }
void printline(char ch, int len)
{
for(int i=1;i<=len;i++)
{cout<<ch; }
cout<<"\n"
}
```

Output:

```
************************************

Enter values for a,b and c: 10 4 6

Call to mathoperation with 1 arg: 15
Call to mathoperation with 2 arg: 20
Call to mathoperation with 3 arg: 6

============================
```

Advantages of Default Arguments:

- Default arguments are useful when we want to increase the capabilities of an existing function by adding another default arguments.
- It helps in reducing the size of program.
- Default arguments improves consistency of program.

Inline function in C++

Why do we need an inline function in C++?

The main use of the inline function in C++ is to save memory space. Whenever the function is called, then it takes a lot of time to execute the tasks If the length of the function is small, then the substantial amount of execution time spent is small.

Syntax for an inline function:

```
inline return_type function_name(parameters)
{
  // function code?
}
```

We cannot provide the inlining to the functions in the following circumstances:

- If a function is recursive.
- If a function contains a loop like for, while, do-while loop.
- If a function contains static variables.
- If a function contains a switch or go to statement

```
#include <iostream.h>
#include<conio.h>
inline int div(int x, int y)
{
   return(x/y);
}
inline int mul(int a, int b)
{
   return(a*b);
}

void main()
{
   cout<<"division of 'x' and 'y' is:"<<div(25,5);A
   cout<<"multiplication of of 'a' and 'b' is:"<<mul(2,3);
```

```
    getch();

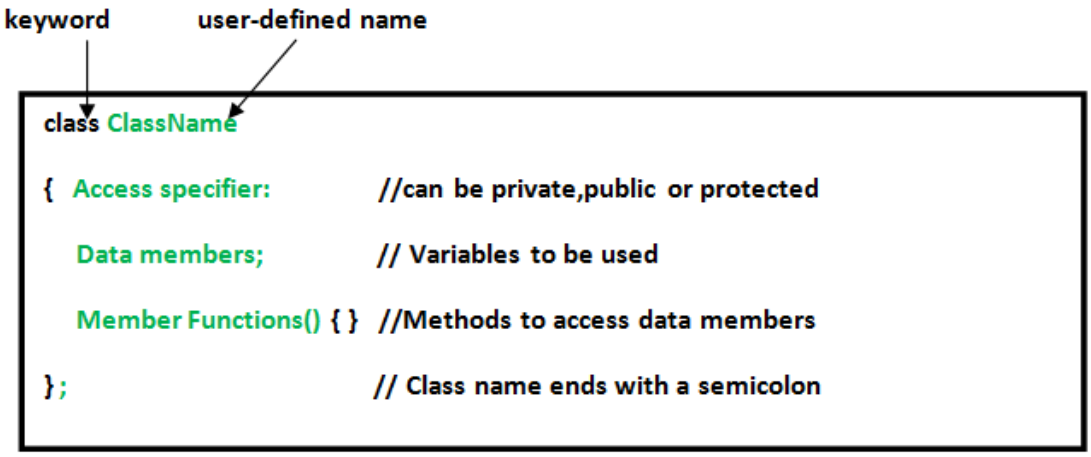
}
```

Classes and Objects:

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 members and member functions, which can be accessed and used by creating an instance of that class.

Generally , a class specification has two parts:

1. Class declaration
2. Class function definition



For example:1

```
class Cube
{
    public:

    int side; // data member(variable) declaration
    void getVolume(); //Member function declaration
};
```

For example:2

```
class Cube
{
    private:
    int side; // data member(variable) declaration

    public:
    void getVolume();    //Member function declaration
};
```

For example:3

```
class Cube
{

    int side; // by default it is private

    public:
    void getVolume();
};
```

Access Specifiers:
There are three access modifiers : **public, private and protected**.

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

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 functions or the friend functions are allowed to access the private data members of a class.

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

Object

An **Object** is an instance of a Class.

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

Syntax:

```
ClassName ObjectName;
```

Eg: Cube ca, emp,s;
Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object.

Syntax: to access data member (Variable)

Object_name. datamamber_name=value;

Eg: ca.side=4; //allowed only to public data members

Syntax: to access member functions (member function calling)

Object_name. memberfunction_name(actual parameters);

Eg: ca.getVolume();

Member Functions in Classes
There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

- **Inside class definition**

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

Syntax:
returntype function_name(argument_list)
{
// statement to be executed;
}
Eg:

```
int getVolume()
    {
     //Statement to be executed;
}
```

Inside class:

```
class Cube
{
    public:
    int side;
    int getVolume()
    {
     return side*side*side; //returns volume of cube
    }
};
```

- **Outside class definition**

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

To define a member function outside the class definition we have to use the scope resolution **::** operator along with class name and function name.

Syntax:

return_type classname :: function_name(argument_list)
{
//statement to be executed;
}
Eg:

```
int Cube :: getVolume()
{
    //statement to be executed;
}
```

Outside the class:

```
class Cube
{
    public:
    int side;
    int getVolume();
}

// member function defined outside class
definition
int Cube :: getVolume()
{
    return side*side*side;
}
```

The main function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

Calling Class Member Function in C++

Similar to accessing a data member in the class, we can also access the public member functions through the class object using the dot operator (.).

Below we have a simple code example, where we are creating an object of the class Cube and calling the member function getVolume():

```
void main()

{

    Cube C1;

    C1.side = 4;     // setting side value

    cout<< "Volume of cube C1 = "<< C1.getVolume();
```

```
}
```

Nesting of member function: Refer class work notes.

Example Program: to calculate cube value using class with function definition inside the class

```
class Cube
{
   public:
   int side;
   int getVolume()
   {
    return side*side*side; //returns volume of cube
   }
};

void main()

{

   Cube C1;

   C1.side = 4;    // setting side value

   cout<< "Volume of cube C1 = "<< C1.getVolume();

}
```

Example Program: to calculate cube value using class with function definition outside the class

```
class Cube
{
   public:
```

```
    int side;
    int getVolume();
}

// member function defined outside class definition
int Cube :: getVolume()
{
    return side*side*side;
}

void main()

{

    Cube C1;

    C1.side = 4;    // setting side value

    cout<< "Volume of cube C1 = "<< C1.getVolume();

}
```

Static data member:

A data member can be qualified as static. Static member variable has certain special characteristics.

- it is initialized to zero when the first object of its class is created.
- Only one copy of that member is created for the entire class and it is shared by all the objects of that class
- It is visible only within the class, but its lifetime is the entire program

1.Syntax: to declare inside the class
static datatype variable_name;

eg: static int a;

2.syntax: to define the static variable outside the class

datatype class_name :: variable_name;

eg: int simple :: a;

3.syntax : to initialize the static variable
datatype class_name :: variable_name = value;

eg: int simple :: a =12;

Example Program: the use of a static data member to count the occurrences of all the objects.

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

class product
{
static int count;
int num;
public:
void changenum(int a)
{
num=a;
count++;
cout<< " num is :"<<num;

}
void getvalue(void)
{
cout<<" count: ";
cout<<count<<"\n";
}

};

int product::count;

Void main()

{

product x,y,z;

x.getvalue();
```

```
y.getvalue();
z.getvalue();

x.changenum(20);
y.changenum(50);
z.changenum(10);

cout<<"after reading data \n";

x.getvalue();
y.getvalue();
z.getvalue();
getch();

}
```

Output:

```
count : 0
count : 0
count : 0

num is 20
num is 50

num is 10

after reading data
count: 3
count: 3
count: 3
```

Static member function:

Member functions can also be declared as static. Properties of static member functions:

- A static function can have access to only other static members declared in the same class.
- A static member function can be called using the class name as follows:

  Class-name:: function_name;

  Example Program for static member function:

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

class test
{
static int count;
int code;
public:
void setcode()
{
code=count++;
}
void showcode()
{
cout<<" object number:" <<code<<"\n";
}

static void showcont()
{
count<<"count:"<<count<<"\n";
}
};

int test::count;

void main()

{

test ta,tb;

ta.setcode();
tb.setcode();

test:: showcount();

test tc;
tc.setcode();

test:: showcount();
```

```
ta.showcode();
tb.showcode();
tc.showcode();


getch();

}
```

Arrays of 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 functions defined in the class, you need to create objects.

Syntax:

```
ClassName ObjectName[number of objects];
```

Below is the C++ program for storing data of one Employee using array:

```
// C++ program to implement the above approach
#include<iostream.h>
#include<conio.h>
class Employee
{
  int id;
  char name[30];
  public:
  void getdata();//Declaration of function
  void putdata();//Declaration of function
};
void Employee::getdata(){//Defining of function
```

```
  cout<<"Enter Id : ";
  cin>>id;
  cout<<"Enter Name : ";
  cin>>name;
}
void Employee::putdata(){//Defining of function
  cout<<id<<"\n";
  cout<<name;
  cout<<endl;
}
void main(){
  Employee emp; //One member
  emp.getdata();//Accessing the function
  emp.putdata();//Accessing the function
getch();
}
```

Below is the C++ program for storing data of one Employee using **array of objects**:

```
// C++ program to implement the above approach

#include<iostream.h>
#include<conio.h>
class Employee
{
  int id;
  char name[30];
  public:
    void getdata();
    void putdata();
```

```
};

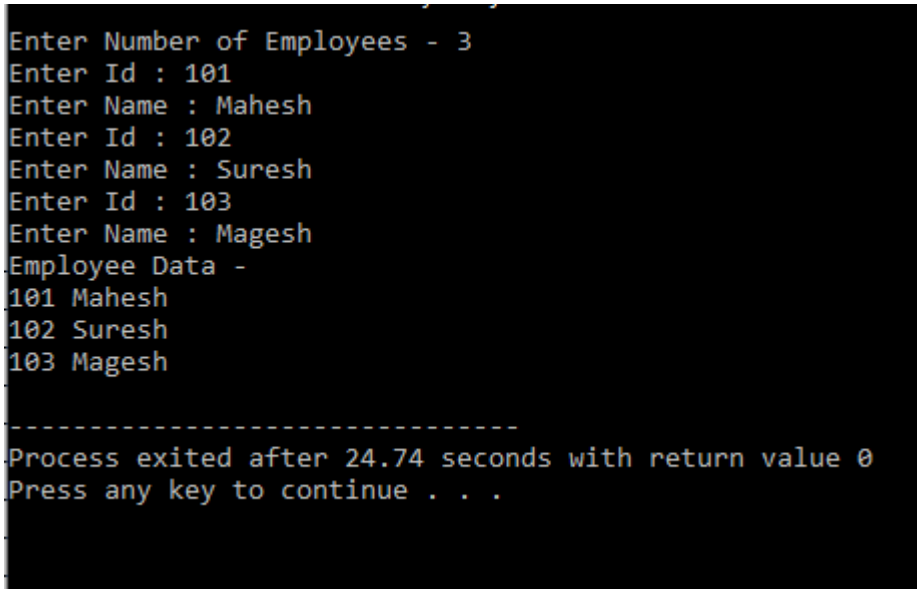
void Employee::getdata()      // Defining the function outside
{
  cout << "Enter Id : ";
  cin >> id;
  cout << "Enter Name : ";
  cin >> name;
}

void Employee::putdata()     // Defining the function outside
{
  cout << id << " ";
  cout << name << " ";
  cout << endl;
}
 void main()
{
  Employee emp[30];   // This is an array of objects
  int n, i;
  cout << "Enter Number of Employees - ";
  cin >> n;

    for(i = 0; i < n; i++)
    emp[i].getdata();    // Accessing the function
     cout << "Employee Data - " << endl;
       for(i = 0; i < n; i++)
    emp[i].putdata();     // Accessing the function

}
```

Output:



Friend function:

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By 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 the keyword friend.

Declaration of friend function in C++

```
1. class class_name
2. {
3.     friend data_type function_name(arguments);          // syntax of friend function.
4. };
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator** outside the class.

Characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of the class
- It can be invoked like a normal function without the help of any object
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name(eg: object. X;)
- It can be declared either in public or private
- Usually it has the objects as arguments.

C++ friend function Example

The simple example of C++ friend function used to calculate mean value.

```
1. #include <iostream.h>
2. #include<conio.h>
3. class Box
4. {
5.     private:
6.         int a
7.         int b;
8.     public:
9.         void getvalue()
10. {
11. cout<<"enter the value of a & b\n";
12. cin>>a>>b;
13.     friend int printmean(Box  x); //friend function
14. };
15. int printmean(Box x)
16. {
17.
18.     return (x.a+x.b)/2;
19. }
20. void main()
21. {
22.     Box x;
23.    x.getvalue();
24.    cout<<"Mean value is : "<< printmean(x)<<endl;
25.     getch();
26. }
```

Output:

```
Length of box: 10
```

A function friendly to two classes

Let's see a simple example when the function is friendly to two classes.

```
1. #include <iostream.h>
2. #include<conio.h>
3. class CSE;          // forward declarartion.
4. Class BSC
5. {
6.     int x;
7.     public:
8.     void setdata(int i)
9.     {
10.        x=i;
11.    }
12.    friend void min(BSC, CSE);        // friend function.
13. };
14. class CSE
15. {
16.    int y;
17.    public:
18.    void setdata(int i)
19.    {
20.        y=i;
21.    }
22.    friend void min(BSC, CSE);                // friend function
23. };
24. void min(BSC b, CSE c)
25. {
26.    if(b.x<=c.y)
27.    cout <<" minimum is "<< b.x << endl;
28.    else
29.    cout << "minimum is "<<c.y <<endl;
30. }
```

```
31.  void main()
32. {
33.   BSC b;
34.   CSE c;
35.   b.setdata(10);
36.   c.setdata(20);
37.   min(c,b);
38.    getch();
39. }
```

Output:

```
10
```

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes CSE and BSC.

C++ Constructor

- In C++, constructor is a special member function.
- The constructor in C++ **has the same name as class name.**
- It is named as "constructor" because it constructs the value of data member of a class.
- It is used to initialize the data members of new object generally.

Characteristics:

1. They should be declared in the public section
2. It is invoked automatically at the time of object creation.
3. They do not have return types, not even void
4. They cannot be inherited.
5. They can have default arguments

Syntax to create constructor:

```
Class classname
{
public:
Classname();

};
```

Eg:

```
Class sample
{
Public:
sample();

};
```

Syntax to define constructor outside class:

```
class class_name
{
 int g, h;
 public:
 class_name();  // Constructor Declared
 . . .
 };
class_name :: class_name()
{
 g=1; h=2;  // Constructor defined
}
```

Calling constructor member function:

This can be done in two ways:

- By calling constructor explicitly

  Class_name object= classname;

- By calling constructor implicitly

  Classname object;

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

- Copy constructor

C++ Default Constructor

- A constructor which has no argument is known as default constructor.
- It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream.h>
#include<conio.h>
class student
 {
   public:
        student()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
void main(void)
{
    student s1; //creating an object of Employee
    student s2;
    getch();
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C++ Parameterized Constructor

- A constructor which has parameters is called parameterized constructor.
- It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream.h>
#include<conio.h>
class Employee {
  public:
```

```
    int id; //data member (also instance variable)
    string name; //data member(also instance variable)
    float salary;
    Employee(int i, string n, float s)
     {
        id = i;
        name = n;
        salary = s;
     }
    void display()
     {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
     }
};
void main()
 {
   Employee e1 =Employee(101, "Alice", 890000);    //creating an object of Employee
   Employee e2=Employee(102, "Wonder", 59000);
   e1.display();
   e2.display();
   getch();
}
```

Output:

```
101  Alice  890000
102  Wonder  59000
```

C++ Destructor

- A destructor works opposite to constructor; it destructs the objects of classes.
- It can be defined only once in a class.
- Like constructors, it is invoked automatically.
- A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Syntax:

```
~classname()
{
// statements;
}
```

Eg:

```
~simple()
{
// statements;
```

}

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream.h>
#include<conio.h>
class alpha
 {
   public:
       alpha()
       {
          cout<<"Constructor Invoked"<<endl;
       }
       ~alpha()
       {
          cout<<"Destructor Invoked"<<endl;
       }
};
void main()
{
cout<<" \n Enter main";
    alpha a1, a2;
{
  cout<<"Enter block1\n";
alpha a3;
}
{
cout<<"\n enter block2\n";
alpha a4;
}
    getch();
}
```

Output:

```
Enter main
Constructor Invoked
Constructor Invoked

Enter block1

Constructor Invoked

Destructor Invoked

Enter block2
Constructor Invoked

Destructor Invoked

Destructor Invoked
Destructor Invoked
```

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
1. #include <iostream>
2. using namespace std;
3. class Employee {
4.   public:
5.      int id; //data member (also instance variable)
6.      string name; //data member(also instance variable)
7.      float salary;
8.      Employee(int id, string name, float salary)
9.       {
10.          this->id = id;
11.          this->name = name;
```

```
12.          this->salary = salary;
13.       }
14.       void display()
15.       {
16.          cout<<id<<"  "<<name<<"  "<<salary<<endl;
17.       }
18. };
19. int main(void) {
20.    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21.    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
22.    e1.display();
23.    e2.display();
24.    return 0;
25. }
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.

Operator overloading is used to overload or redefines most of the operators available in C++.

It is used to perform the operation on the user-defined data type.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

the process of overloading involves the following steps:

1. create a class that defines the data type that is to be used in the overloading operations
2. declare the operator function **operator op() in the** public mode
3. define the operator function to implement the required operations

Syntax of Operator Overloading

```
1. return_type class_name  : : operator op(argument_list)
2. {
3.     // body of the function.
4. }
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream.h>
#include<conio.h>
class Test
{
```

```
  private:
     int a;
int b;
int c;
  public:
     void getdata(int x, int y, int z);
     void display();
      void operator -();
     };
void Test::getdata(int x, int y, int z)
{
        a=x;
    b=y;
    c=z;
      }
Void Test::display()
{
cout<"a="<<a;
cout<"b="<<b;
cout<"c="<<c;
}
  void Test::operator -()
      {
       a=-a;
    b=-b;
    c=-c;
      }
void main()
{
    Test t;
int p,q,r;
  cout<"Enter any three values\n"
cin>>p>>q>>r;
    t.getdata( p,q,r) // calling of a function "void operator ++()"
    t.display();
  –t;
  t.display();
```

```
    getch();
}
```

Output:

```
The Count is: 10
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```
#include <iostream>
using namespace std;
class A
{

    int x;
      public:
      A(){}
    A(int i)
    {
       x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
```

```
29.     return 0;
30. }
```

Output:

```
The result of the addition of two objects is : 9
```

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

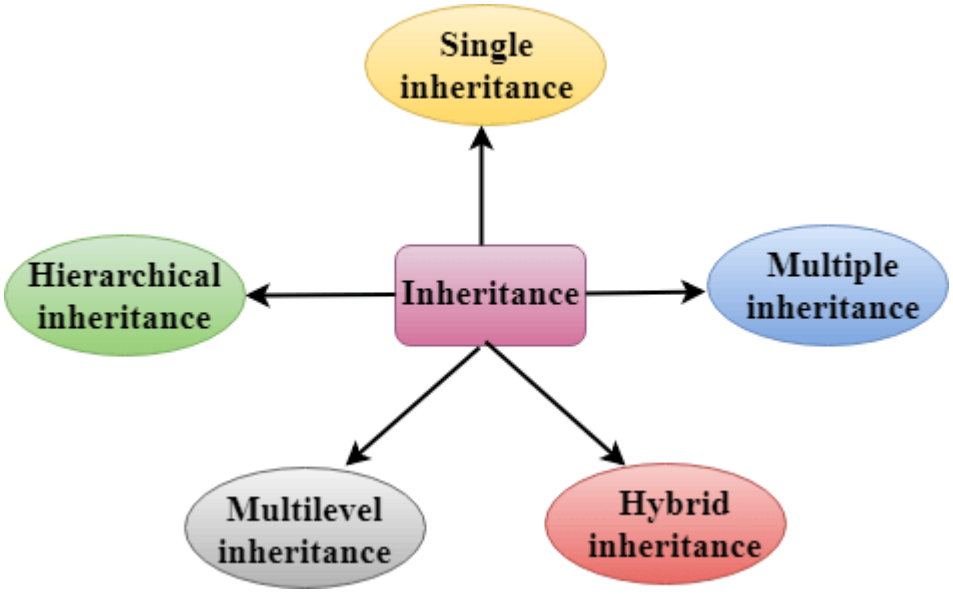
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

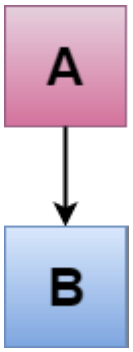
C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

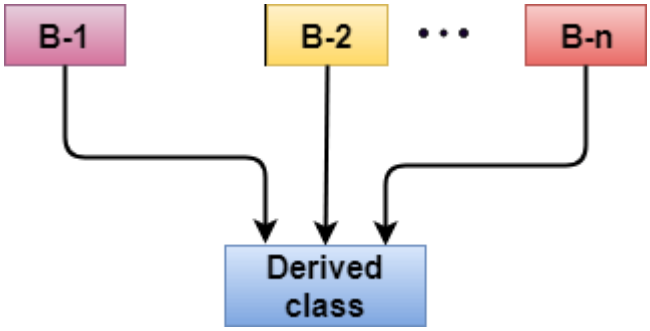
C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



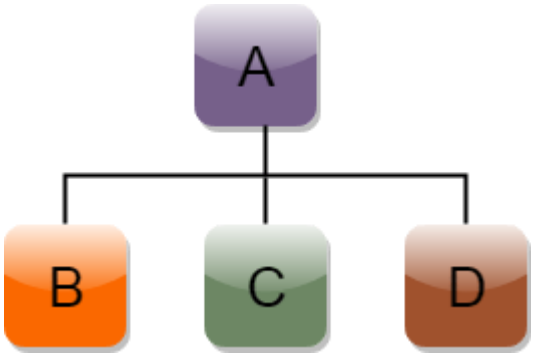
C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



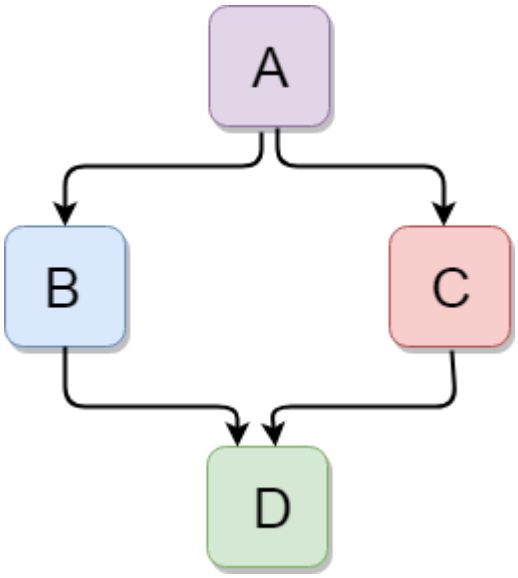
C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

1. **class** derived_class_name :: visibility-mode base_class_name

```
2. {
3.     // body of the derived class.
4. }
```

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

C++ Single Level Inheritance Example: Inheriting Fields

Syntax

```
class base_class_1
{
    // class definition
};
class derived_class: visibility_mode base_class_1
{
    // class definition
};
```

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

#include <iostream.h>

```
#include<conio.h>
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
      public:
    void bark(){
   cout<<"Barking...";
    }
  };
int main(void) {
   Dog d1;
   d1.eat();
   d1.bark();
   return 0;
}
```

Output:

```
Eating...
Barking...
```

Let's see a simple example.

```
#include <iostream.h>

#include<conio.h>

// Base class
class Shape
 {

protected:
      int width;
      int height;

public:
      void setWidth()
       {
         cout<”enter the value of width\n”;
         cin>>width;
       }
    void setHeight(int h);
};
  void Shape::setHeight(int h)
     {
         height = h;
      }



// Derived class
class Rectangle: public Shape
  {
   int s;
   public:
      void getArea();
   };
void Rectangle::getArea()
 {
         s= (width * height);
        cout<<” Total Area is “<<s;
      }


void main()
 {
   Rectangle Rect;
```

```
    Rect.setWidth();
    Rect.setHeight(7);

    // Print the area of the object.
    Rect.getArea();

    getch();
}
```

```
#include <iostream>
using namespace std;
class A
{
    int a = 4;
    int b = 5;
    public:
    int mul()
    {
        int c = a*b;
        return c;
    }
};

class B : private A
{
    public:
    void display()
    {
        int result = mul();
        std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
    }
};
int main()
{
   B b;
   b.display();
```

```
28.
29.     return 0;
30. }
```

Output:

```
Multiplication of a and b is : 20
```

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

- Multiple Inheritance

The inheritance in which a class can inherit or derive the characteristics of multiple classes, or a derived class can have over one base class, is known as Multiple Inheritance

Syntax

```
class base_class_1
{
   // class definition
};
class base_class_2
{
   // class definition
};
class derived_class:  visibility_mode  baseclass_1 ,   visibility_mode baseclass_2
{
   // class definition
};
```

Example : for Multiple Inheritance

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

class electronicDevice
{
  char name[20];
    public:

      void mobilename()
      {
     cout<<" what is your mobile's brand name\n";
```

```
    cin>>name;
   cout << "I am an electronic device.\n\n";
        }
};

class Computer
{
    int Gb;

  public:
        // constructor of the base class 2
        void gbdata(int a)
        {
       Gb = a;

    cout << "I am a computer.\n"<<"my GB is  "<<Gb;
        }
};


class Linux : public electronicDevice, public Computer
{
char name3[15];
public:
void os();

};


void Linux::OS()
{

cout<<"which operating system you have installed?";
cin>>name3;
   cout << "I am the operating system\n";

}

void main()
{

       Linux b;
     b.mobilename();
   b.gbdata(500);
b.os();

cout<<"\n DONE";
  getch();
}
```

- Multilevel Inheritance

The inheritance in which a class can be derived from another derived class is known as Multilevel Inheritance.

Syntax

```
class  class_A
{
   // class definition
};
class  class_B:  visibility_mode  class_A
{
   // class definition
};
class  class_C: visibility_mode class_B
{
   // class definition
};
```

Example: Multilevel inheritance- to print the student details

```
#include <iostream.h>
#include<conio.h>
class student
{
            protected:
            int rollnumber;
 public:
      void getdata()
            {
            cout << "Enter your roll number ";
            cin >>rollnumber;
            }
void putdata();

};


void student::putdata()
{
cout<<”ROLL NUMBER IS”<<rollnumber;
}
class submarks : public student  // derived class from base class
```

```
{
            public:
            int m1,m2,m3,m4,m5;
            void readdata()
            {
               cout << "\nEnter all your marks for m1,m2,m3,m4,m5";
              cin >>m1>>m2>>m3>>m4>>m5;
            }
void showmarks();

};

void submarks::showmarks()
{
cout<<”M1=”<<m1<<”\nM2=”<<m2<<”\nM3=”<<m3<<”\nM4=”<<m4<<”\nM5=”<<m5;
}

class displaydetails : public submarks   // derived from class derive1
{
            private:
            int total;
            public:
            void result()
            {
            total=m1+m2+m3+m4+m5;

           cout << "\n THE RESULT IS"<<  total;
            }

};
void main()
{
    displaydetails  a;      //object of derived class
    a.getdata();
    a.readdata();
    a.putdata();
    a.showmarks();
   a.result();
    getch();
}                          //end of program
```

- Hierarchical Inheritance

The inheritance in which a single base class inherits multiple derived classes is known as the Hierarchical Inheritance.

Syntax

```
class class_A
{
   // class definition
};

class class_B: visibility_mode class_A
{
   // class definition
};

class class_C : visibility_mode class_A
{
   // class definition
};

class class_D: visibility_mode class_A
{
   // class definition
};
```

```
// C++ program to demonstrate hierarchical inheritance

#include <iostream.h>
#include<conio.h>

// base class
class Principal {

  protected:
  char cname[30];
   int no_dept;

   public:

void  collegedetails();
void  displaycollege();
```

```
};

void  Principal::displaycollege() {

    cout<<”Enter your College name”; cin>>cname;
    cout<<”\n Enter no. of departments available in your college”;
    cin>>no_dept;
    }
void  Principal::displaydetails(){
 cout<<”My college name :”<<name;
 cout<<”\n no. of Dept”<<no_dept;
}


// derived class 1
class HOD : public Principal {

int no_programme;
char pname[30];
char pname2[30];
public:
    void programmedetails();
    void displayprogramme();
 };
void HOD::programmedetails()
{
cout<<”\n enter the no. of programme”;
cin>>no_programme;
cout<<”\n Enter Programme name”;
cin>>pname1>>pname2;
}
void HOD ::displayprogramme()
{
cout<<”No. of Programes in the Department:”<<no_programme;
cout<<”\n Programme names are:”<<pname1<<pname2;
}


// derived class 2
class Lecturer : public Principal {

public:
  char sname[20];
 void subjectname();
void displaysubject();
};
```

```
void Lecturer::subjectname()
{
cout<<”\Enter the Subject name:” cin>>sname;
}
void Lecturer::displaysubject()
{
cout<<”\n Subject handling:”<<sname;
}


void main() {
        HOD H;
    cout << "HOD Class:" << endl;
    H.collegedetails();  // Parent Class function
    H.programmedetails();

    cout<<”\n Lecturer Class”;
    Lecturer L;

    L.subjectname();

    H.displaycollege();  // Parent Class function
    H.displayprogramme();

    L.displaysubject();

    getch();
}
```

- Hybrid Inheritance

Hybrid Inheritance, as the name suggests, is the combination of two or over two types of inheritances.

```
class class_A
{
   // class definition
};
class class_B
{
   // class definition
};
class class_C: visibility_mode class_A, visibility_mode class_B
{
   // class definition
```

```
};
class class_D: visibility_mode class_C
{
   // class definition
};
class class_E: visibility_mode class_C
{
   // class definition
};
```

Exercise: Develop a C++ program to print your form 4 / 6 results using Hybrid inheritance

RUN TIME POLYMORPHISM:

- **Definition:** subclass object and subclass method overrides the Parent class method during runtime. Hence it is called runtime polymorphism
- It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.

It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the **virtual keyword.**

```
1.  #include <iostream.h>
2.  #include<conio.h>;
3.  class food
4.  {
5.     public:
6.  void  eat( )
7.  {
8.  cout<<"Eating... food";
9.     }
10. };
11. class bread: public food
12. {
13.  public:
14.  void eat( )
15.    {          cout<<"Eating bread...";
16.    }
17. };
18. void main( )
19.  {
20.    bread d;
21.    d. eat();
22.    getch();
```

23. }

C++ Virtual Functions

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the **virtual keyword.**
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A **'virtual' is a** keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.

For example, consider the code below:

```
class Base {
   public:
    void print()
{
        // code
```

```
    }
};

class Derived : public Base {
   public:
    void print()
{
        // code
    }
};
```

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call the `print()` function, it calls the `print()` function of the `Base` class.

In other words, the member function of `Base` is not overridden.

```
void main() {
    Derived derived1;
    Base *base1 = &derived1;

     base1->print();   // calls function of Base class

    getch();
}
```

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the `virtual` **keyword**.

```
class Base {
   public:
    virtual void print()
 {
        // code
    }
};
```

Virtual functions are an integral part of polymorphism in C++.

Example 1: C++ virtual Function

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

class Base {
   public:
```

```
void message()
{
cout<<”\n Message from Base function”;
}
virtual void print()
   {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:

void message()
{
cout<<”\n Message from Derived function”;
}
    void print()
  {
        cout << "Derived Function" << endl;
    }
};

void main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();
   base1->message();
Base b;
base1= &b;
base1->print();
base1->message();

    getch();
}
```

Run Code

Output

```
Message from Base function
Base function
```

```
Derived Function
Message from Base function
```

Here, we have declared the `print()` function of `Base` as `virtual`.

So, this function is overridden even when we use a pointer of `Base` type that points to the `Derived` object `derived1`.

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream.h>
#include<conio.h>;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
```

```
11.     void show()
12.     {
13.         cout << "Derived class is derived from the base class." << endl;
14.     }
15. };
16. void main()
17. {
18.     Base *bptr;
19.     //Base b;
20.     Derived d;
21.     bptr = &d;
22.     bptr->show();
23.     getch();
24. }
```

Output:

```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.