Classes and Objects

Classes and Objects

Classes:

A class is a way to bind data and functions together in a single data type. The variables and functions enclosed in a class are called ***data members*** and ***member functions***. Since classes by default are private, class allows the data (and functions) to be hidden if necessary from external use.

This mechanism of binding data and functions that operate on that data is call ***data encapsulation***.
This mechanism of hiding data of a class from the outside world (other classes) so that any access to it either intentionally or unintentionally can't modify the data is called ***data hiding.***

Class Declaration:

A class specification has two parts:
(I) Class Declaration
(II) Class Function definitions
The class declaration describes the type and scope of its member. The class definitions describe how the class functions are implemented. The general form of class declaration is

```
class <class name>
{

        private:            \\ by private we mean that members can be accessed only from within the class i.e
                                                    member data can be accessed by the member functions.

                data members;
                member functions;


        protected:          \\ by protected we mean that members can be accessed only by member functions
                                                    and friends of that class.
                data members;
                member functions;


        public:             \\ by public we mean that members can be accessed outside class also.

                data members;
                member functions;
};
```

e.g. The following declaration illustrates the specification of a class student.

```
        class student
        {
                int rollno;
                float marks;
                public:
                        void getdata ( );
                        void display ( );
        };
```

Creating Object:

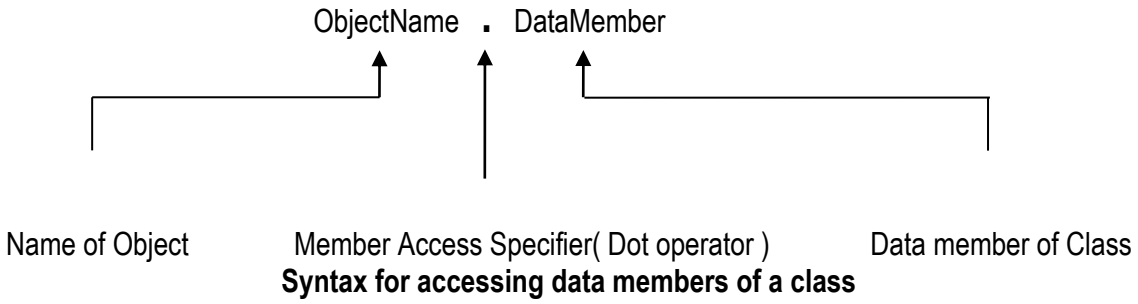
An object is an *instance* of a class. In general a class is a user defined data type, while an object is an instance of an class.

e.g. Let *student* be the name of class
student S1,S2;
creates variables S1, S2 of type student. Once the objects are declared memory is allocated. Objects can also be created by placing their names immediately after the closing brace.

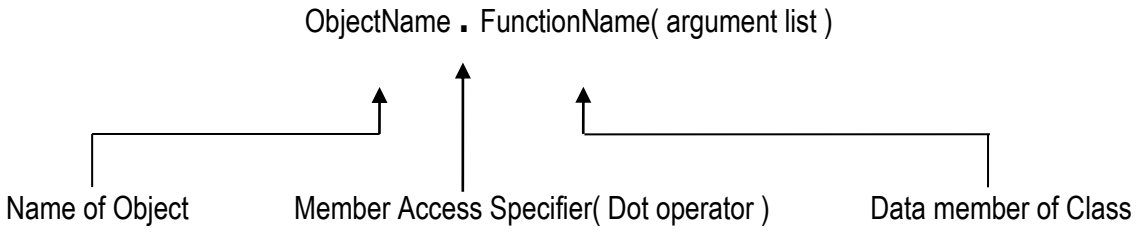
e.g. class *student*
{ …..
}S,S2,S3;

Accessing Class members

After creating the object there is a need to access the class member. This can be done by using a dot (.) operator. The syntax for accessing members is as follows:



Syntax for accessing data members of a class



Syntax for accessing member Function of a class

e.g. Function call statement

S1.getdata(129,704.5); assign value 129 to rollno and 704.5 to marks of object S1.
S1.display( ); will display value of data members.

Similarly the statement

S1.roll = 129; is invalid because data member is private.

Constructors

Declaration and Definitions:

A constructor is a special member function which is used to initialize the objects of a class.
The general syntax is as follows:

```
class classname
{
        private:   ……
        public:
                classname( Parameter list );  //constructor declaration
};

classname : : classname(Parameter list )
{                                       //constructor definition
        ……
}
```

Default Constructor: The constructors that can take no arguments are called default constructors.

e.g. time t1,t2;

```
class time
{
        private:
                int hr,min;

        public:
                time( ); //default constructor
};

time : : time( ) : hr(0), min(0)
{ }
```

Parameterized Constructor: The constructors that can take arguments are called parameterized constructors.

e.g. time t1(2,2), t2 (5,7);

```
class time
{
        private:
                int hr,min;
        public:
                time(int x, int y);   //parameterized constructor
};

time : : time(int x, int y ) : hr(x), min(y)
{ }
```

This is done by passing argument to constructor function when the objects are created. This can be done in two ways.

1. By calling the constructor implicitly (Implicit calling) time t1(11,11);
2. By calling the constructor explicitly (Explicit calling) time t1 = time(11,11);

Copy Constructor:

A copy constructor takes a reference to its own class as parameter i.e. a constructor having a reference to an instance of its own class as an argument is known as copy constructor.
Copy constructors are used in following situations:

- The initialization of an object by another object of the same class.
- Return of objects as function value.
- Stating the object as by value parameters of the function.

e.g. time t1; //default constructor

time t2 = t1; //copy constructor is invoked

time t3(t1);

The general syntax of copy constructor is classname (classname & ptr)

e.g. time(time & ptr)

where time is a user defined class name and ptr is a pointer to a class object time.

Special characteristics of constructor:

a. A constructor name must be same as the name of class.
b. Constructor should be declared in the public part.
c. They are invoked automatically when the object are created.
d. A constructor may not be static.
e. They do not have return type, not even void.
f. They are used to initialize data members of a class.
g. They cannot be inherited, though a derived class can call the base class constructor.
h. They are used to allocate resources such as memory to the dynamic data members of a class.
i. Constructor can have default arguments

Destructors

A destructor is a special member function which is used to destroy the object that has been created by constructor.

The main characteristics are

a. The destructor is having same name as name of class except that it is preceded y tild (~).
b. They cannot be declared static, const.
c. They should have public access in class declaration.
d. A destructor has no return type and neither it takes any argument.

The general syntax is as follows:

```
class classname
{
        private:
                ……
        public:

        classname ( );  //constructor

        ~classname ( );  //destructor
};
```

```
e.g
class time
{
        private:
                int hr,min;
        public:

        time ( ); //default constructor

        ~time ( ); // destructor
};
```

Program to demonstrate constructor and destructor function

```
class sample
{
        public:
                sample ( )
                {
                        cout<<"\n Object is born";
                }
                ~sample ( )
                {
                        cout<<"\n Object dies";
                }
};

void main ( )
{
        sample S;

        cout<<endl<<"Main terminated";
}
```

The output is

```
Object is born

Main terminated

Object dies
```

The output shows when the constructor and destructor are called.

The destructor is called for an object when it reaches the end of its scope.

Note: Objects are destroyed in the reverse order of creation.

Member Functions

Class method definition

The data member of a class must be declared within the body of the class, whereas member functions of the class can be defined in the following ways:

(i) Outside the class definition
(ii) Inside the class definition

(I) Member function outside the class Definition

In C++, the member functions can be declared outside the class declaration. i.e. declare function prototype within the body of a class and then define it outside the body of a class. This is done by using ***scope resolution operator* (: :).** It acts as an identity label to inform the compiler, the class to which the function belongs. The general syntax is

```
return-type classname : : functionName ( argument declaration )
{

fuction body

}
```

(II) Member function inside the class Definition

In other method of function definition is to replace the function declaration by the actual function definition inside the class. All the member functions defined within the body of class are treated as inline by default.

```
#include<iostream.h>
#include<conio.h>
class sum
{
        private:
                int a;
                 int b;
        public:

        sum(int x, int y): a(x),b(y)

        void add1()
        {
                cout<<"\n SUM1 ="<<a+b;
        }

        void add2( );

};
```

```
void sum : : add2()
{
                cout<<"\n SUM2 ="<<a+b;
}

void main()
{
        Sum s;
        clrscr( );

        s.add1( );

        s.add2( );

        s.add3( );

        getch( );
}
```

Inline Functions:

An inline member function is like a macro, any call to this function in a program is replaced by the function itself. This is called inline expansion. By this the overhead occurred in the transfer of control by the function call and function return statements are cut down.
C++ treats all the member functions defined within a class as inline and those defined outside as non-inline. Member functions defined outside the class declaration can be made inline by prefixing the keyword inline.

Inline returnType classsname : : Functionname (arguments)
{
body of inline function
}

```
#include<iostream.h>
#include<conio.h>
class sum
{
        private:
                int a;
                 int b;
        public:

        inline void add( );
};
inline void sum : : add()
{
                cout<<"\n SUM2 ="<<a+b;
}
```

```
void main()
{
        Sum s;
        clrscr( );
        s.add( );

        getch( );
}
```

Friend Functions:

Friend function is a special mechanism for letting non member functions access private data. A friend function possesses the following characteristics:

1. The scope of the friend function is not limited to class in which it has been declared friend.

2. A friend function can be invoked like a normal function without the help of any object.

3. Unlike the class member functions, it can't access the member name directly & has to use an object name and dot operator with each member name to access the private and public members.

4. It can be declared either in private or public part of class without affecting its meaning.

5. It has objects as arguments.

```
class time
{
   private:
          int hr,min;
   public:

          time( ) : hr(0), min(0) //default constructor

          void get( );

          friend put( time t );
};
void time : : get (  )
{
          cout<<"\n enter time";
          cin>>hr>>min;
}
```

```
void put ( time t  ) //object passed to friend function
{
          cout<<"\n time is "<< t.hr <<" : "t.min<<"hrs";
}
void main( )
{
      time t1;
      clrscr(  );

      t1.get( );   //calling mem fun using object
      put(t1);    //friend function called without any
                     object reference
      getch( );
}
```

Friend Classes

A **friend class** in C++ can access the "private" and "protected" members of the class in which it is declared as a friend. We can also declare all the member functions of one class as the friend of another class. In such cases, the class is called a friend class. This can be specified as follows:

```
class A
{
          ……

          friend class B;                  //all member functions of B are friend to A
};
```

Two classes having same friend

A non member function may have friendship with one or more classes. When a function has declared to have friendship with more than one class, the friend class should have forward declaration. This means that it needs to access the private members of both classes.

```
class B; //forward Declaration
class A
{
        private:
                int data;
        public:
                A(int x):data(x)  //parameterized
constructor
                { }
                friend add(A,B);
};
class B
{
        private:
                int data;
        public:
                B(int x):data(x)  //parameterized
constructor
                { }
                friend add(A,B);
};
```

```
void add( A a1, B b1)
{
        int sum;
        sum = a1.data + b1.data ;
        cout<<"\n Sum = "<<sum;
}

void main( )
{
        A a1(11);
        B b1(22);
        clrscr( );
        add(a1,b1); //friend fun called with two
                                objects of different
classes A & B
        getch( );
}
```

Static Data members

If a data item is defined as static, then only one such item is created for the entire class, no matter how many objects there are. A static member variable has certain special characteristics:

(1) It is initialized to zero when the first object of its class is created. No other initialization is permitted.

(2) Only one copy of that member is created for the entire class and is shared by all other objects of that class, no matter how many objects are created.

Static Member functions

The static member functions have following properties:

(1) A static function has access to only other static members (functions or variables) declared in the same class.

(2) A static member function can be called directly by using the classname as follows:

classname : : functionname( );

(3) A static functions acts as global for data members of its class without affecting the rest of the program.

Program illustrating use of static data member and static member function

```
class data
{
        private:
                static int count;
        public:
                data( )
                {  count++;  }
        static void counter( )
        {
                cout<<"\n Total objects = "<<count;
        }
};
data : : count = 0;
```

```
void main( )
{
        clrscr( );
        data : : counter( );
        data d1,d2;
        data : : counter( );
}
```

OUTPUT:

Total objects = 0

Total objects = 2

Constant Member functions

Declaring a member function with the ***const*** keyword specifies that the function is a "read-only" function that does not modify the objet for which it is called.

To declare a constant member function, place the ***const*** keyword after the closing parenthesis of the argument list. The ***const*** keyword is required in both the declaration and the definition. The constant member function cannot modify any data members or call any member functions that aren't constant.

```
class data
{
        private:
                int x;
        public:
                data( ) : x(0)
                {  }
                void NonConstFun( )
                {
                cout<<"\n Enter value of x = ";
                        cin>>x;
                }
```

```
const void ConstFun( )
                {
                        cout<<"\n Value of x = "<<x;
                }
};

void main( )
{

        data d1;

        clrscr( );

        d1.NonConstFun ( );
        d1. ConstFun( );
}
```

OUTPUT:

```
Enter value of x = 11
 Value of x = 11
```

Overloading Member Functions

Need of operator overloading,

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function overloading

Function overloading is a concept that allows multiple functions to share the same name with different argument type i.e. function definition can have multiple forms.

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function print() is being used to print different data types:

```
#include <iostream>
class printData
{
  public:
    void print(int i)
   {
      cout << "\n Printing int: " << i ;
    }

    void print(double  f)
   {
      cout << "\n Printing float: " << f << endl;
    }
};
```

```
void  main( )
{
  printData pd;

  // Call print to print integer
  pd.print(5);
  // Call print to print float
  pd.print(500.263);

}
Output:
Printing int: 5
Printing float: 500.263
```

Operator overloading,

The mechanism of giving new meaning to an operator is know as operator overloading.

We can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Things to remember while using Operator overloading in C++ language

1. Operator overloading cannot be used to change the way operator works on built-in types. Operator overloading only allows to redefine the meaning of operator for user-defined types.

2. There are two operators assignment operator(=) and address operator(&) which does not need to be overloaded. Because these two operators are already overloaded in C++ library. For example: If *obj1* and *obj2* are two objects of same class then, you can use code **obj1=obj2;** without overloading = operator. This code will copy the contents object of *obj2* to *obj1*. Similarly, you can use address operator directly without overloading which will return the address of object in memory.

3. Not all operators in C++ language can be overloaded. The operators that cannot be overloaded in C++ are ::(scope resolution), .(member selection), .*(member selection through pointer to function) and ?:(ternary operator).

Types of Operator Overloading :

Unary operator overloading	Unary operators are the operators that work on a single operand.

class point
{
private:
int x, y, z;
public:
 	
point( ) : x(0), y(0), z(0)
{ }
 	
point (int a, int b, int c)
{ x = a ; y = b ; z = c ; }
 	
void display ( )
{
 	
cout<<" \n point = "<< x <<" : " << y <<" : " << z;
}
operator – ( )
{
x = -x ;
y = -y ;
z = -z ;
}
};
	void main ( )
	{
	point p1(11,12,13);
	clrscr( );
	p1.display ( );
	-p1;
	p1.display ( );
	getch( );
	}
	Output:
	Point = 11,12,13
	Point = -11,-12,-13

Binary operator overloading	Binary operators are the operators that work on two operand.

class point
{
private:
int x, y, z;
public:
point( ):x(0),y(0),z(0)
{ }
point (int a, int b, int c)
{ x = a ; y = b ; z = c ; }
void display ( )
{ cout<<" \n point = "<< x <<" : " << y <<" : " << z; }
 	
point operator + (point p)
{
point temp;
temp.x = x + p.x;
temp.y = y + p.y;
temp.z = z + p.z;
return temp;
}
};
	void main ( )
	{
	point p1(11,12,13);
	clrscr( );
	p1.display ( );
	-p1;
	p1.display ( );
	getch( );
	}
	Output:
	Point = 11,12,13
	Point = -11,-12,-13

Constructor overloading

In C++, Constructor is automatically called when object (instance of class) create.It is special member function of the class. Which constructor has arguments that's called Parameterized Constructor.

- One Constructor overload another constructor is called Constructer Overloading
- It has same name of class.
- It must be a public member.
- No Return Values.
- Default constructors are called when constructors are not defined for the classes.

```
#include<iostream>
#include<conio.h>
class Example
  {
   int a,b;

          public:
   Example()   //Constructor without Argument
        {
             a=50;     b=100;
            cout<<"\nIm Constructor";
   }

Example(int x,int y) //Constructor
{                              with Argument
           a=x;      b=y;
          cout<<"\nIm Overloaded Constructor";
   }
   void Display()
  {
          cout<<"\nValues :"<<a<<"\t"<<b;
   }
};
```

```
void main()
{
      Example Object(10,20);
      Example Object2;
      // Constructor invoked.
      Object.Display();
      Object2.Display();

      getch();

}
```

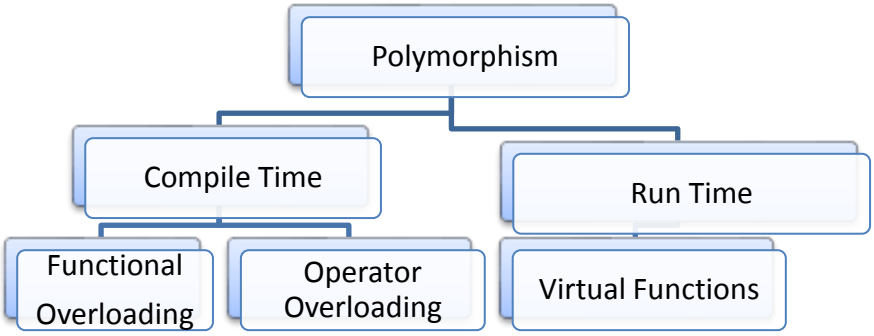
Output:

```
Im Constructor
Im Overloaded Constructor
Values :10      20
Values :50      100
```

Polymorphism & Virtual Functions

Polymorphism

Polymorphism can be defined as the ability to use the same name for two or more related but technically different tasks. The main advantage of Polymorphism is that it reduces complexity by allowing same name to perform different tasks.



Compile Time: The information is known to compiler at the compile time and therefore compiler is able to select the appropriate function for a particular call at compile time itself. This is called early binding or static binding or static linking. **It is also known as compile time polymorphism because early binding simply means that an object is bound to its function call at compile time.**

Run Time: The information is known at the run time and therefore compiler is able to select the appropriate function for a particular call at run time itself. This is called late binding or dynamic binding or dynamic linking. **Since the function is linked with particular object after compilation i.e., during program execution, the process is called late binding.**

Virtual Functions

A virtual function is that one that does not really exist but it appears real in some parts of program. These functions are defined in base class in the public section and they provide mechanism by which the derived class can override it. These functions are bound dynamically.

```
class base
{
        public:
                virtual void show( )
                { cout<<"\n Base class show called ";
        }
};
class derv1: public base
{
        public:
          void show( )
            { cout<<"\n Derv1 class show called "; }
};
```

```
class derv2: public base
{
          public:
            void show( )
              { cout<<"\n Derv2 class show called "; }
};
Void main( )
{
      base *bptr ;    derv1 d1 ;        derv2 d2;
      bptr = &d1 ;  bptr->show( ) ;
      bptr = &d2 ;  bptr->show( ) ;
}
Output:   Derv1 class show called
          Derv2 class show called
```

Pure virtual function

A pure virtual function is a function declared in base class that has no definition relative to the base class. To declare a pure virtual function, use the general form:

virtual returntype functionname(parameter-list) = 0 ;

e.g: virtual void show( ) = 0 ;

```
class base
{
        public:
                virtual virtual void show( )= 0 ;

};
class derv1: public base
{
        public:
          void show( )
            {  cout<<"\n Derv1 class show called ";  }
};
```

```
Void main( )
{
       base *bptr ;
      derv1 d1 ;
       bptr = &d1 ;  bptr->show( ) ;
}
Output:   Derv1 class show called
```

Abstract Base Class

A class that contains at least one pure virtual function is said to be abstract. No object of an abstract class can be created.

Although we cannot create objects of an abstract class, we can create pointers and references to an abstract base class. This allows abstract class to support run-time polymorphism, which relies upon base-class pointer.

```
class base
{
        public:
                virtual virtual void show( )= 0 ;

};
class derv1: public base
{
        public:
          void show( )
            {  cout<<"\n Derv1 class show called ";  }
};
```

```
Void main( )
{
    base obj;     // error cannot create objects of an
                  abstract class
      base *bptr ;
       derv1 d1 ;
       bptr = &d1 ;  bptr->show( ) ;
}
Output:   Derv1 class show called
```

Virtual Destructors

Virtual Destructors member function is invoked to free memory automatically. But the destructor of the derived class is not invoked to free the memory which was allocated by the constructor of the derived class. It is because the destructor is non virtual and the message will not reach the destructor under late binding.
So it is better to have destructor as virtual and virtual destructors are essential in program to free the memory space effectively using late binding.

```
class base
{
        public:
                virtual ~ base( )
                { cout<<"\n Base class Destructor "; }
};
class derv1: public base
{
        public:
        ~ derv1( )
                { cout<<"\n Base class Destructor ";}
};
```

```
Void main( )
{
   base obj;     // error cannot create objects of an
                 abstract class
    base *bptr ;
     derv1 d1 ;
     delete bptr;
}
Output:  Base  class Destructor
         Derv1 class Destructor
```