# Class and Objects

## Introduction

In OOP data and functions are wrapped up into a single unit, which is called class. In other words, class is a unit that contains a group of logically related data items and functions that work on them. Once a class is defined, we can declare variables of that type i.e., we can create objects. A class type variable is called an object or instance. A class would be the data type, and an object would be the variable. Classes have the property of information hiding. It allows data and functions to be hidden, if necessary, from external use. Classes are also referred to as programmer-defined data types.

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

The variables and functions defined inside the class are called ***class members***. Variables declared inside the class are called ***data members*** and functions inside the class are called ***member functions***.

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword ***class*** followed by the ***class name***; and the ***class body***, enclosed by a pair of curly braces. A class definition must be followed by a semicolon. Classes are generally declared with the following format:

class class\_name

{

access-specifier:

variable declarations;

function declarations;

……………………………….

access-specifier:

variable declarations;

function declarations;

…………………………………

access-specifier:

variable declarations;

function declarations;

};

**Example:**

class Room

{

int length;

int breadth;

};

## Visibility labels or Access Specifiers or Encapsulation Modifiers

The members of a class are classified into three categories: *private, public, and protected*. *Private*, *protected*, and *public* are reserved words and are called member *access specifiers* or *visibility labels*. There are 3 types of access specifiers or visibility modifiers. They are;

1. *public*

By default class members are private. If all the three visibility labels are absent, then by default, all the members are private.

1. *protected*
2. *private.*

**private** members of a class are accessible only from within the same class using member functions. We cannot access them outside of the class. That is only the member functions can have access to the private members (both data & functions).

**protected** members are accessible with member functions of the same class and also from any class immediately derived from it (i.e. immediate subclasses).

**public** members are accessible from anywhere where the object is visible. That is, public members (both data & functions) can be accessed from outside the class.

##### Example 1:

class student

{

private:

int roll;

float marks;

public: //Remember member functions are public, so can be accessed from main()

void getdata(int a, float b) // *function definition*.

{

roll = a;

marks = b;

}

void displaydata()

{

cout<<"Roll number : "<<roll<<"\nMarks : "<<marks;

}

};

int main()

{

student ram; // here, ram is an object of type student.

ram.getdata(12,56); // *function call using object ram*, values 12 and 56 are assigned to data members roll and marks.

ram.displaydata() // another *function call* to display data.

return 0;

}

In this example functions are defined inside the class.

**C++ Structure and Class:**

By default, the members of a class are private, while, by default, the members of a structure are public. Following simple example represents this fact.

struct student

{

int roll;

float marks;

};

int main()

{

student Ram; //C++ declaration, but error in C.

Ram.roll = 10; **//allowed, public**

Ram.marks =150; **//allowed, public**

}

class student

{

int roll;

float marks;

};

int main()

{

student Ram;

Ram.roll = 10; **//not allowed, private**

Ram.marks =150; **//not allowed, private**

}

## Object Declaration/Define C++ Objects

Once a class is defined, you can declare objects of that type. The syntax for declaring an object is the same as that for declaring any other variable.

**Syntax:**

***class\_name*** object\_name;

**Example:**

The following statements declare two objects of type ***student***:

***student*** Ram, Hari; //Creating two objects Ram and Hari.

## Accessing Class Members

Once an object of a class is declared, it can access the public members of the class. The private members cannot be accessed directly from outside of the class. The private data of class can be accessed only by the member functions of that class. The public member can be accessed outside the class from the main function. When an object of the class is created then the members are accessed using the ‘.’ dot operator.

**Synatx:**

for calling a public member data,

*object\_name.data\_member\_name;*

for calling a public member function,

*object\_name.function\_name (actual\_arguments);*

## Scope of Class and its Members

class Sample

{

int x, y ;

void func()

{

cout<<"Hello from private function"<<endl;

}

protected:

int k;

void PFunc()

{

cout<<"Hello from Protected function"<<endl;

}

public:

int z ;

void funcall()

{

cout<<"Hello from Public function"<<endl;

func(); **//Allowed**

PFunc(); **//Allowed**

}

void setdata(int a, int b, int c, int d)

{

x=a;

y=b;

k=c;

z=d;

}

void display()

{

cout<<"x:"<<x<<endl;

cout<<"y: "<<y<<endl;

cout<<"z: "<<z<<endl;

cout<<"k: "<<k<<endl;

}

}; **//class definition ends here**

int main( )

{

Sample P;

P.x=0; **// error, x is private.**

P.k=100; **//error, k is protected.**

P.z=10 ; **// ok, z is public.**

cout<<"z: "<<P.z<<endl; //ok

P.func(); **// error, func() is private.**

P.PFunc(); **//error, PFunc() is proteceted.**

P.funcall(); **//ok, funcall()**is public.

P.setdata(4, 5, 6, 8); //ok, **setadata i**s public.

P.display(); **//ok, display is public.**

return 0;

}

## Defining Member function of class

Member functions can be defined in two places:

* Outside the class definition.
* Inside the class definition.

### Inside the class definition

* When a function is defined inside the class, it is treated as an inline function.
* Normally only small functions are defined inside the class.

**Example:**

class Student

{

int roll;

char name[30];

public:

void getdata(void)

{

cout<<" Enter name: ";

cin>>name;

cout<<" Roll: ";

cin>>roll;

}

void display()

{

cout<<"Name: " <<name<<endl<<"Roll: " <<roll<<endl;

}

};

int main()

{

Student A, B;

A.getdata();

A.display();

B.getdata();

B.display();

return 0;

}

### Outside the class definition:

* When function is defined outside the class, function header contains the ‘*identity label*’ or ‘*membership label’*.
* This *label* tells the compiler that which class the function belongs to. General form of a member function definition is:

*return-type* class-name : : function-name (argument declarations)

{

**//function body.**

}

The membership label ***class-name ::*** tells the compiler that the function function-name belongs to the class class-name.

**Characteristics of member functions**:

* Several classes can use the same function name. The ‘membership label’ will resolve their scope.
* Member functions can access the private data of the class.
* A member function can call another member function directly, without using the dot operator.

##### Example 2:

class student //specify a class

{

private :

int rollno; //class data members

float marks;

public:

void initdata(int r, int m)

{

rollno=r;

marks=m;

}

void getdata(); **//member function to get data from user**

void showdata();**// member function to show data**

};

void student :: getdata()

{

cout<<"Enter Roll Number : ";

cin>>rollno;

cout<<"Enter Marks : ";

cin>>marks;

}

void student :: showdata()

{

cout<<"Roll number : "<<rollno<<" \nMarks : "<<marks;

}

int main()

{

student st1, st2; //define two objects of class student

st1.initdata(5,78); //call to member function to initialize

st1.showdata() ;//call, member function to display data

st2.getdata(); //call, member function to input data

st2.showdata(); //call, member function to display data

return 0;

}

**Another Example:**

#include<iostream>

#include<cstring>

using namespace std;

class Student

{

int roll;

char name[30];

public:

void getdata(int a, char c[])

{

roll= a;

strcpy(name, c);

}

void display()

{

cout<<"Roll: " <<roll<<endl<<"Name: " <<name;

}

};

int main()

{

char x[30]= "Ram";

int a= 20;

Student A;

A.getdata(a, x);

A.display();

return 0;

}

##### Exercise:

1. Define a class **Interest** with following specifications:
   1. Data Members
      1. principal
      2. time
      3. rate
   2. Member Functions
      1. getdata() to assign initial values
      2. display() to display values
      3. Finterest() to find and display the interest.
2. Define a class **Room** with following specifications:
   1. Data Members
      1. length
      2. width
   2. Member Functions
      1. getdata() to assign initial values
      2. display() to display length and width
      3. area() to find and display the area.
3. Define a class **Employee** with following specifications:
   1. Data Members
      1. name
      2. age
   2. Member Functions
      1. getdata() to assign initial values
      2. display() to display name and age
4. Define a class **Account** with following specifications:
   1. Data Members
      1. name of the depositor
      2. account number
      3. balance amount in the account
   2. Member Function
      1. To assign initial values
      2. To deposit an amount
      3. To withdraw an amount after checking the balance.
      4. To display name and balance.

## Memory Allocation for Objects

For each object, the memory space for data members is allocated separately because the data members will hold different data values for different objects. However, all the objects in a given class use the same member functions. Hence, the member functions are created and placed in memory only once when they are defined as a part of a class specification and no separate space is allocated for member functions when objects are created.

data1

data2

data1

data2

data1

data2

function1

function2

**object1**

**object2**

**object3**

**Common for all objects**

## Nesting of Member Functions

A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions. For example, the class below shows the nesting of member function ***findinterest()*** inside the member function ***findtotal()***.

class total

{

private:

float principle, time, rate;

float findinterest()

{

return principle \* time \* rate / 100;

}

public:

void setdata(float p, float t, float r)

{

principle = p;

time = t;

rate = r;

}

float findtotal()

{

return principle + findinterest();

}

};

**Remember:** Like private data member, some situations may require certain member functions to be hidden from the outside call. In the above example, the member function *findinterest()* is in private block and is hidden from the outside call. A private member function can only be called by another member function that is a member function of its class.

**Another Example:**

void set :: display(void)  
{  
 cout << "largest value=" << largest() <<"\n";  
}  
  
int main()  
{  
 set A;  
 A.input();  
 A.display();   
return 0;  
}  
**The output of program would be:**  
Input value of m and n  
25 18  
Largest value=25

class set  
{  
int m,n;  
public:  
void input(void);  
void display(void);  
void largest(void);  
};  
int set :: largest(void)  
{  
if(m >= n)  
return(m);  
else  
return(n);  
}

void set :: input(void)  
{

cout << "Input value of m and n"<<"\n";  
cin >> m>>n;

}

## Static Members of a Class

Class can have ***static data members*** and ***static member functions***. Static variables declared inside the class are called ***static data members*** and member functions that are declared ***static*** are called ***static member functions*** *of the class*.

* For e.g.: class test

{

***static*** int a; **// static data member a.**

int b;

public:

***static*** void getdata(); **// static member function getdata().**

};

### Static Data Members

* The static member variables have similar properties to that of C static variables (life time and scope).
* But while declared as a class member static member variable has certain special characteristics as:
  + It is initialized to zero when the objects of its class are created.
  + Static data member is shared between all objects of the class. That is only one copy of that member is created for the entire class no matter how many objects are created.
  + It is visible only within the class, but it has the lifetime of entire program.
  + *Static variable are normally used to maintain values common to the entire program.* 
    - *For example, a static data member can be used to count the occurrences of all the objects.*
  + The type and scope of each static member variable must be defined outside the class definition. This is necessary because the static member variables are stored separately rather than as a part of an object.
    - **Syntax defining a static data member:**

data\_type class\_name **::** static\_variable\_name; //assigned 0.

data\_type class\_name **::** static\_variable\_name=value; //assigning some value

* + Static member variables are associated with the class rather than with any class object, so they are also known as ***class variables***.

##### Example 3:

class test

{

private:

static int count;

int m\_nID;

public:

void Getdata(int a) **// function defined inside the class**

{

m\_nID = a;

count++;

}

void GetCount(void)

{

cout<<"Count is: " <<count <<endl;

}

};

int test::count; **// definition of static data member, automatically assigned 0.**

int main()

{

test cFirst, cSecond, cThird;  **// objects are created, count is initialized to 0.**

cout<<" Before reading data: " << endl;

cFirst.GetCount();

cSecond.GetCount(); **// display count**

cThird.GetCount();

cout<<" Reading data..... " << endl;

cFirst.Getdata(120);

cSecond.Getdata(23); **// getting data into objects**

cThird.Getdata(11);

cout<<" After reading data: "<< endl;

cFirst.GetCount();

cSecond.GetCount();  **// display count**

cThird.GetCount();

return 0;

}

The static variable ***count*** is automatically initialized to 0 when objects are created. It is incremented whenever data is read into objects.

**Initial value can be assigned to static variable as;**

int test::count = 10; //10 is assigned to static data member count.

### Static Member Functions

* Member functions that are declared using ***static*** keyword are called **static member functions** of the class.
* A **static member function** can have access to only **static class members** (both function and variables) declared in the same class.
* A **static member function** can be called using the class name (instead of its objects) as follows:

class name :: function\_name(Args.);

##### Example 4:

#include<iostream>

using namespace std;

class test

{

private:

static int count;

int m\_nID;

public:

void Getdata(int a) **//function defined inside the class.**

{

m\_nID = a;

count++;

}

static void GetCount(void) **// static function so can't access non-static class member.**

{

cout<<"Count is: " <<count <<endl;

cout<<" m\_nID :" << m\_nID; **// illegal**

}

};

int test::count; **// definition of static data member.**

int main()

{

test t1; **//object created, so count is initialized to 0**.

t1.Getdata(120);

test :: GetCount(); **// display count, u can also use t1.GetCount().**

return 0;

}

## Objects as Function Arguments

Like any other built-in type objects can be used as function arguments. This can be done in three ways:

* **Pass by value or call by value.**
* **Pass by reference or call by reference.**
* **Pass by address(pointer) or call by address**

If we pass object value to function that is called by value, in this case any changes made to the object inside the function do not affect the actual objects.

In call by reference, objects are passed through reference. If we pass objects as a call by address, we pass the address of the object. Therefore, any change made to the object inside the function will reflect in the actual object. The object which is used as an argument when we call the function is known as actual object and the object which is used as an argument within the function definition is called formal object.

**Pass by value:**

* A copy of entire object is passed to the function. Any changes made to the object inside the function do not affect the object used in the function call.
* Consider the following program: in this program function ***sum ()****,* takes two objects as arguments.

##### Example 5

#include<iostream>

using namespace std;

**class time**

{

int hrs;

int minutes;

public:

void getdata(int a ,int b)

{

hrs = a;

minutes =b;

}

void display(void);

void sum(time, time); **//function declaration with objects as function arguments**

};

**void time :: sum(time t1, time t2)** **// function definition outside the class**

{

minutes = t1.minutes+t2.minutes; **// inside the function body, *minutes* and *hrs* belongs to calling object, in this case T3(look function call in main()).**

hrs = minutes/60;

minutes =minutes%60;

hrs=hrs+t1.hrs+t2.hrs;

}

**void time:: display(void)**

{

cout<< hrs<<" hours and " << minutes<<" minutes";

}

**int main()**

{

time T1,T2,T3; **// three objects are created.**

int a,b,c,d;

cout<<"enter values for a,b,c,d: "<<endl;

cin>>a>>b>>c>>d;

T1.getdata(a,b);

T2.getdata(c,d);

cout<<"T1 : " <<"\t";

T1.display();

cout<<endl;

cout<<"T2 : " <<"\t";

T2.display();

cout<<endl;

T3.sum(T1,T2); **// function call with object as arguments.**

cout<<"T3 : " <<"\t";

T3.display();

return 0;

}

### Pass by Reference:

* A reference of an object is passed to the function. Formal objects now become the aliases to actual objects. So any changes made to the object inside the function will automatically reflect in the actual object.
* Consider the following program: in this program function ***swap ()***, takes reference of two objects as arguments.

##### Example :Swaping valyes of two objects

#include<iostream>

using namespace std;

**class number**

{

int a;

public:

**void getdata(int x)**

{

a=x;

}

void display(void);

void swap(number &, number &); **//function declaration, reference to objects(same as reference variable).**

};

**void number:: swap(number & n1, number &n2)**

{ number t;

t.a=n1.a;

n1.a=n2.a;

n2.a=t.a;

}

**void number:: display(void)**

{

cout<< a;

}

int main()

{number n1,n2;

n1.getdata(2);

n1.display);

n2.getdata(5);

n2.display();

n1.swap(n1,n2);

n1.display();// it displays 5

n2.display();// it displays 2

return 0;}

### Pass by Address or Call by address:

* If we pass objects as a call by address, we pass the address of the object. Therefore, any change made to the formal object inside the function will reflect in the actual object.

##### Example :Swaping valyes of two objects

* Like any other built-in type we can define object pointers. Once we define object pointer, we need to use (->) arrow operator to access the class members i.e., data members and member functions.

#include<iostream>

using namespace std;

**class number**

{

int a;

public:

**void getdata(int x)**

{

a=x;

}

void display(void);

void swap(number &, number &); **//function declaration, reference to objects(same as reference variable).**

};

**void number:: swap(number \* n1, number \*n2)**

{ number t;

t.a=n1->a;

n1->a=n2->a;

n2->a=t.a;

}

**void number:: display(void)**

{

cout<< a;

}

int main()

{number n1,n2;

n1.getdata(2);

n1.display);

n2.getdata(5);

n2.display();

n1.swap(&n1,&n2);

n1.display();// it displays 5

n2.display();// it displays 2

return 0;}

## Returning Objects From Functions

A function not only receives objects as arguments but also can return them. A function can return objects also in 3 ways:

1. **Return by value**

In this method ,a copy of the object is returned to the function call. Like pass by value, it makes program to work slow when we are working with large objects.

##### Example :

#include<iostream>

using namespace std;

class Complex

{

float real;

float img;

public:

void getdata(float a , float b)

{

real = a;

img =b;

}

void display(void);

Complex Difference(Complex); **//Remember, here function Difference has return type Complex.**

};

Complex Complex::Difference(Complex C)

{

Complex s;

s.real = real - C.real;

s.img = img - C.img;

return s;

}

void Complex:: display(void)

{

cout<<real<<" + " <<img<<"i"<<endl;

}

int main( )

{

Complex C1, C2, C3;

float a,b,c,d;

cout<<"enter values for a, b, c, d: "<<endl;

cin>>a>>b>>c>>d;

C1.getdata(a,b);

C2.getdata(c,d);

cout<<"C1 : ";

C1.display();

cout<<"C2 : ";

C2.display();

C3 = C1.Difference(C2); **//This call to Difference() function returns an object, and**

cout<<"C3 : "; //**content of that object is assigned to C3.**

C3.display();

return 0;

}

Exercise:

Create a class called Dollar with two data members’ dol and cent. Construct different member functions with the following operations.

* + To input data for Dollar objects.
  + To show the data of Dollar objects.
  + Member function to add two Dollar objects and then return the result.

1. **Return by Reference:**

A C++ function can **return** a **reference** in a similar way as it **returns** value. When **returning** a **reference**, be careful that the object being referred to does not go out of scope. So it is not legal to **return** a **reference** to local objects. But you can always **return** a **reference** on a static variable. Like passing by reference, return by reference also makes program faster while working with larger objects.

Example

#include<iostream>

using namespace std;

class Complex

{

float real;

float img;

public:

void getdata(float a , float b)

{

real = a;

img =b;

}

void display(void);

Complex & Difference(Complex &); **//Remember Difference has reference as return type**

};

Complex & Complex::Difference(Complex &C)

{

C.real = real - C.real;

C.img = img - C.img;

return C;

}

void Complex:: display(void)

{

cout<<real<<" + " <<img<<"i"<<endl;

}

int main( )

{

Complex C1, C2, C3;

float a,b,c,d;

cout<<"enter values for a, b, c, d: "<<endl;

cin>>a>>b>>c>>d;

C1.getdata(a,b);

C2.getdata(c,d);

cout<<"C1 : ";

C1.display();

cout<<"C2 : ";

C2.display();

C3 = C1.Difference(C2);

cout<<"C3 : ";

C3.display();

return 0;

}

1. **Return by pointer:**

In this method , return by pointer returns address of the object to the function call. It also cannot return local objects and makes program to work faster in case of larger objects.

Example

#include<iostream>

using namespace std;

class Complex

{

float real;

float img;

public:

void getdata(float a , float b)

{

real = a;

img =b;

}

void display(void);

Complex\* Difference(Complex \*); **//Remember, here function Difference has return type as Complex type pointer .**

};

Complex\* Complex::Difference(Complex \*C)

{

C->real = real – C->real;

C->img = img – C->img;

return C;

}

void Complex:: display(void)

{

cout<<real<<" + " <<img<<"i"<<endl;

}

int main( )

{

Complex C1, C2, \*C3;

float a,b,c,d;

cout<<"enter values for a, b, c, d: "<<endl;

cin>>a>>b>>c>>d;

C1.getdata(a,b);

C2.getdata(c,d);

cout<<"C1 : ";

C1.display();

cout<<"C2 : ";

C2.display();

C3 = C1.Difference(&C2);

cout<<"C3 : ";

C3->display();

return 0;

}