

UNIT 3

CLASSES AND OBJECTS

LH – 8HRS

PRESENTED BY
ER. SHARAT MAHARJAN
OBJECT ORIENTED PROGRAMMING (OOP)

CONTENTS (LH – 8HRS)

- 3.1 A Simple Class and Object, (class definition: data members, member functions),
- 3.2 Accessing members of class,
- 3.3 Initialization of class object (Using Constructor: Default Constructor, Parameterized Constructor, Copy Constructor, The Default Copy Constructor),
- 3.4 Destructor
- 3.5 Objects as Function Arguments,
- 3.6 Returning Objects from Functions,
- 3.7 Structures and Classes,
- 3.8 Memory allocation for Objects,
- 3.9 Static data members,
- 3.10 Member functions defined outside the class (using scope resolution operator)

Introduction

- The main purpose of C++ programming is to add **object orientation to the C programming language** and **classes are the central feature of C++** that supports object-oriented programming and are **often called user-defined types**.
- **A class** is used to specify the form of an object and it **combines data representation and methods for manipulating that data** into one neat package. **The data and functions within a class are called members of the class.**
- **Class is a blueprint of real world objects.**
- 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.**

3.1 A Simple Class and Object

1. Class:

- A class definition **starts** with the keyword **class** followed by the **class name**; and the **class body**, enclosed by a pair of curly braces terminated by semicolon.
- The **body of the class** contains the **keywords private, public and protected**.
- **Private data and functions** can only be **accessed from within the member functions of that class**. **Public data or functions** are accessible from everywhere. Usually the data within a class is private and functions are public. The data is hidden so it will be safe from accidental manipulation, while the **functions** that operated on the data **are public so they can be accessed from outside the class**.

The general form of class declaration is:

```
class class_name
{
    private:
        data-type variable1;
        data-type variable2;
    data-type function1(argument declaration){
        //function body
    }
    data-type function2(argument declaration){
        //function body
    }
    .....
    .....
    public:
        data-type variable3;
        data-type function3(argument declaration){
            //function body
        }
    .....
};
```

For example:

```
class square{  
    private:  
        int length;    //attribute or property of integer data-type  
    public:  
        void setLength(int l){    //function1  
            length=l;  
        }  
        int getLength(){    //function2  
            return length;  
        }  
        int findArea(){    //function3  
            return length*length;  
        }  
        int findPerimeter(){    //function4  
            return 4*length;  
        }  
};
```

2. Objects:

- The **class declaration does not define any objects but only specifies what they will contain. Once class has been declared, we can create objects (variables) of that type by using the class name. For example:**

square s;

- It **creates an object 's' of type square. We can create any number of objects from the same class. For example:**

square s1,s2,s3;

- Objects can also be created when a class is defined by placing their names immediately after the closing brace. For example:

class square{

.....

}s1,s2,s3;

3.2 Accessing members of class

- When an **object** of the class is created then the members are accessed using the **'.'** dot operator. For example:

```
square s;
```

```
s.setLength(5);
```

```
cout<<"Length="<<s.getLength()<<endl;
```

```
cout<<"Area= "<<s.findArea()<<endl;
```

```
cout<<"Perimeter= "<<s.findPerimeter()<<endl;
```

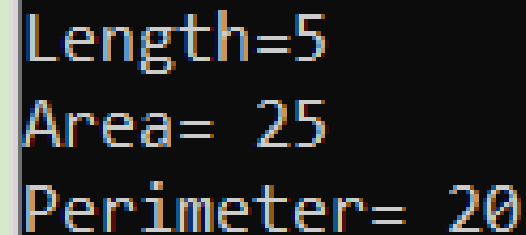
- Private class members can't be accessed in this way outside of the class. For example, if the following statements are written inside the main function, the program generates compiler error.

```
s.length=10;
```


LAB 1: A complete program using class:

```
#include<iostream>
#include<conio.h>
using namespace std;
class square{
    private:
        int length;
    public:
        void setLength(int l){
            length=l;
        }
        int getLength(){
            return length;
        }
        int findArea(){
            return length*length;
        }
        int findPerimeter(){
            return 4*length;
        }
};
int main(){
    square s;
    s.setLength(5);
    cout<<"Length="<<s.getLength()<<endl;
    cout<<"Area= "<<s.findArea()<<endl;
    cout<<"Perimeter= "<<s.findPerimeter()<<endl;
    getch();
    return 0;
}
```

OUTPUT:

A screenshot of a terminal window with a black background and white text. The output shows three lines: 'Length=5', 'Area= 25', and 'Perimeter= 20'. The text is displayed in a monospaced font, and there is a small cursor at the end of each line.

Length=5
Area= 25
Perimeter= 20

3.3 Initialization of class object

- When an **object** is created all the members of the object are allocated **memory spaces**. Each object has its individual copy of member variables.
- However the **data members are not initialized automatically**. If left uninitialized these members **contain garbage values**. Therefore it is important that the **data members are initialized to meaningful values at the time of object creation**.
- When a C++ program runs, it **creates certain objects in the memory and when the program exit the objects must be destroyed so that the memory could be reclaimed for further use**. C++ provides mechanisms to cater to the above two necessary activities through **constructors and destructors methods**.

1. Constructor:

It is a **special member function that is executed automatically whenever an object is created**. It is used for automatic initialization. Automatic initialization is the **process of initializing object's data members** when it is first created **without making a separate call to a member function**. The **name of the constructor is same as the class name**. For example:

```
class square{  
    private:  
        int length;  
    public:  
        square(){    //constructor  
            length=0;  
        }  
        .....  
};
```

Types of Constructors:

- a. Default Constructor
- b. Parameterized Constructor
- c. Copy Constructor
- d. Default Copy Constructor

a. Default Constructor:

- Default constructor is the constructor which **doesn't take any argument**. It has no parameters. For example:

LAB 2:

```
#include<iostream>
#include<conio.h>
using namespace std;
class hello{
    public:
        hello(){
            cout<<"Hello World";
        }
};
int main(){
    hello h;
    getch();
    return 0;
}
```

OUTPUT:

A screenshot of a terminal window with a black background. The text "Hello World" is displayed in a light blue monospaced font. A white cursor (underscore) is positioned at the end of the text.

b. Parameterized Constructor:

The constructors **that can take arguments** are called parameterized constructors. For example:

LAB 3:

```
#include<iostream>
#include<conio.h>
using namespace std;
class square{
    private:
        int length;
    public:
        square(int l){
            length=l;
        }
        int getLength(){
            return length;
        }
};
int main(){
    square s(5);    //square s=square(5); is valid too.
    cout<<"Length="<<s.getLength()<<endl;
    getch();
    return 0; }
```

OUTPUT:



c. Copy Constructor:

A copy constructor is a **member function** that **initializes an object using another object of the same class**. It has a **single parameter of reference type** that refers to the class itself. For example:

LAB 4:

```
#include<iostream>
#include<conio.h>
using namespace std;
class item{
    private:
    public:
        int code, price;
        item(int c, int p){ //parameterized constructor
            code=c;
            price=p;
        }
        item(item &x){ //copy constructor-need to pass object of item as argument
            code=x.code; //b.code=a.code
            price=x.price;
        }
        void display(){
            cout<<"Code: "<<code<<endl<<"Price: "<<price;
        }
};

int main(){
    item a(100,200);
    item b(a); //copying a into b
    b.display();
    getch();
    return 0; }
```

OUTPUT:

```
Code: 100
Price: 200
```

d. Default Copy Constructor:

- If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a **member-wise copy between objects**. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection etc.
- **Default copy constructor can be invoked** using assignment operator as below:
 square s2=s1;

Constructor Overloading:

- We can define more than one constructor in a class either with different number of arguments or with different type of argument which is called constructor overloading. It is a variation of function overloading and is used to achieve polymorphism. For example:

LAB 5:

```
#include<iostream>
#include<conio.h>
using namespace std;
class item{
    private:
        int code,price;
    public:
        item(){                //default constructor
            code=price=0; }

        item(int c, int p){    //parameterized constructor
            code=c;
            price=p; }

        item(item &x){         //copy constructor
            code=x.code;
            price=x.price; }

        void display(){
            cout<<"Code: "<<code<<endl<<"Price: "<<price<<endl<<endl; }
};
int main(){
    item a;
    item b(100,200);
    item c(b);    //copying b into c
    a.display();
    b.display();
    c.display();
    getch();
    return 0; }
```

OUTPUT:

```
Code: 0
Price: 0

Code: 100
Price: 200

Code: 100
Price: 200
```

3.4 Destructor

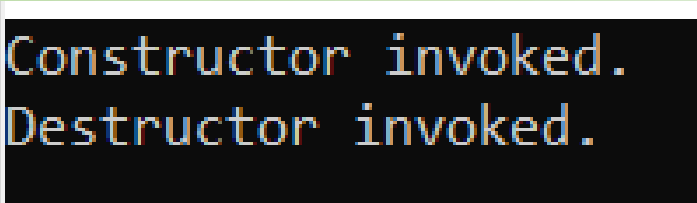
- **Destructor is a member function which is invoked automatically whenever an object is going to be destroyed for destroying the memory that was allocated for the object by the constructor. Meaning, a destructor is the last function that is going to be called before an object is destroyed.**
- **Destructors have same name as the class preceded by a tilde (~)**
Destructors don't take any argument and don't return anything.

LAB 6: For example:

```
#include<iostream>
#include<conio.h>
using namespace std;
class employee{
    public:
        employee(){
            cout<<"Constructor invoked."<<endl;
        }
        ~employee(){
            cout<<"Destructor invoked."<<endl;
        }
};

int main(){
    employee e;
    getch();
    return 0; }
```

OUTPUT:



```
Constructor invoked.
Destructor invoked.
```

3.5 Objects as Function Arguments

Like any other data type, an object may be used as a function argument in three ways:

- a. Pass by value (x=a)
- b. Pass by reference (&x=a)
- c. Pass by pointer (*x=&a)

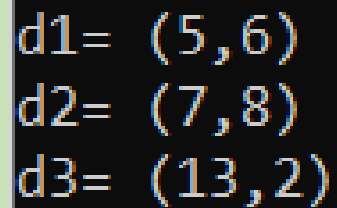
a. Pass by value:

- If arguments are passed by value to the method, a copy of the object is passed to the function. Any changes made to the object inside function do not affect the object used in the function call. It is slow when working with large objects.
- An object can be passed by value to the function as below:

LAB 7:

```
#include<iostream>
#include<conio.h>
using namespace std;
class distances{
    private:
        int feet,inches;
    public:
        void setData(int f, int i){
            feet=f;
            inches=i; }
        void addDistance(distances d1, distances d2){           //1foot = 12inches
            feet = d1.feet + d2.feet;
            inches = d1.inches + d2.inches;
            feet = feet + inches/12;
            inches = inches%12; }
        void showData(){
            cout<<"("<<feet<<","<<inches<<")"<<endl; }
};
int main(){
    distances d1,d2,d3;
    d1.setData(5,6);
    d2.setData(7,8);
    d3.addDistance(d1,d2);
    cout<<"d1= ";
    d1.showData();
    cout<<"d2= ";
    d2.showData();
    cout<<"d3= ";
    d3.showData() ;
    getch();
    return 0;}
```

OUTPUT:



```
d1= (5,6)
d2= (7,8)
d3= (13,2)
```

b. Pass by reference:

- If arguments are passed by reference, an address of the object is passed to the function. The function works directly on the actual object used in the function call. It is faster even while working with large objects.
- An object can be passed by reference to the function as below:

LAB 8:

```
1st.cpp x Untitled1.cpp x
1  #include<iostream>
2  #include<conio.h>
3
4  using namespace std;
5
6  class distances{
7  private:
8      int feet,inches;
9  public:
10     void setData(int f, int i){
11         feet=f;
12         inches=i;
13     }
14     void addDistance(distances &d1, distances &d2){
15         feet= d1.feet+d2.feet;
16         inches=d1.inches+d2.inches;
17         feet=feet+inches/12;
18         inches=inches%12;
19     }
20     void showData(){
21         cout<<"("<<feet<<","<<inches<<")"<<endl;
22     }
23 };
24
25 int main(){
26     distances d1,d2,d3;
27     d1.setData(5,6);
28     d2.setData(7,8);
29     d3.addDistance(d1,d2);
30     cout<<"d1= ";
31     d1.showData();
32     cout<<"d2= ";
33     d2.showData();
34     cout<<"d3= ";
35     d3.showData();
36     getch();
37     return 0;
38 }
```

OUTPUT:

```
d1= (5,6)
d2= (7,8)
d3= (13,2)
```

c. Pass by pointer:

- It can also be used to work directly on the actual object used in the function call. It is faster while working with large objects. For example:

LAB 9:

```
1st.cpp x  Untitled1.cpp x
1  #include<iostream>
2  #include<conio.h>
3
4  using namespace std;
5
6  class distances{
7  private:
8      int feet,inches;
9  public:
10     void setData(int f, int i){
11         feet=f;
12         inches=i;
13     }
14     void addDistance(distances *d1, distances *d2){
15         feet= d1->feet+d2->feet;
16         inches=d1->inches+d2->inches;
17         feet=feet+inches/12;
18         inches=inches%12;
19     }
20     void showData(){
21         cout<<"("<<feet<<","<<inches<<")"<<endl;
22     }
23 };
24
25 int main(){
26     distances d1,d2,d3;
27     d1.setData(5,6);
28     d2.setData(7,8);
29     d3.addDistance(&d1,&d2);
30     cout<<"d1= ";
31     d1.showData();
32     cout<<"d2= ";
33     d2.showData();
34     cout<<"d3= ";
35     d3.showData() ;
36     getch();
37     return 0;
38 }
```

OUTPUT:

```
d1= (5,6)
d2= (7,8)
d3= (13,2)
```

3.6 Returning Objects from Functions

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

- a. Return by value
- b. Return by reference
- c. Return by pointer

a. Return by value:

In this method, a **copy of the object is returned** to the function call. One can return any type of object by value. It is slow when working with large objects.

For example:

LAB 10:

```
#include<iostream>
#include<conio.h>
using namespace std;
class point{
    private:
        float x,y;
    public:
        void setPoint(float a, float b){
            x=a;
            y=b;
        }
        point midPoint(point p1, point p2){
            point p;    //declare object of point to return later
            x=(p1.x+p2.x)/2;
            y=(p1.y+p2.y)/2;
            p.x=x;
            p.y=y;
            return p;
        }
        void showMidpoint(){
            cout<<"Co-ordinate of midpoint:"<<endl;
            cout<<"x= "<<x<<endl<<"y= "<<y<<endl;
        }
};

int main(){
    point c,d,e;
    c.setPoint(1,3);
    d.setPoint(2,4);
    e.midPoint(c,d);
    e.showMidpoint();
    getch();
    return 0;
}
```

E(obj) ko value —

OUTPUT:

```
Co-ordinate of midpoint:
x= 1.5
y= 3.5
```

b. Return by reference:

- In this method, an address of the object is returned to the function call. We can't return local objects by reference because they go out of scope when exiting from the method and hence no longer exists in memory. It is faster even while working with large objects. For example:

LAB 11:

```
#include<iostream>
#include<conio.h>
using namespace std;
class point{
    private:
        float x,y;
    public:
        void setPoint(float a, float b){
            x=a;
            y=b; }
        point &midPoint(point &p1, point &p2){
            x=(p1.x+p2.x)/2;
            y=(p1.y+p2.y)/2;
            p1.x=x;
            p1.y=y;
            return p1; }
        void showMidpoint(){
            //cout<<"Co-ordinate of midpoint:"<<endl;
            cout<<"x= "<<x<<endl<<"y= "<<y<<endl; }
};
int main(){
    point c,d,e;
    c.setPoint(1,3);
    d.setPoint(2,4);
    e.midPoint(c,d);
    e.showMidpoint();
    getch();
    return 0; }
```

OUTPUT:

```
Co-ordinate of midpoint:
x= 1.5
y= 3.5
```

c. Return by pointer:

- It returns address of the object to the function call. It also can't return local objects. It is faster even while working with large objects. For example:

LAB 12:

```
#include<iostream>
#include<conio.h>
using namespace std;
class point{
    private:
        float x,y;
    public:
        void setPoint(float a, float b){
            x=a;
            y=b;
        }
        point *midPoint(point *p1, point *p2){
            x=(p1->x+p2->x)/2;
            y=(p1->y+p2->y)/2;
            p1->x=x;
            p1->y=y;
            return p1;
        }
        void showMidpoint(){
            cout<<"Co-ordinate of midpoint:"<<endl;
            cout<<"x= "<<x<<endl<<"y= "<<y<<endl;
        }
};

int main(){
    point c,d,e;
    c.setPoint(1,3);
    d.setPoint(2,4);
    e.midPoint(&c,&d);
    e.showMidpoint();
    getch();
    return 0; }
```

OUTPUT:

```
Co-ordinate of midpoint:
x= 1.5
y= 3.5
```

3.7 Structures and Classes pardaina

Structure is an **user defined type** which contains the collection of different types of data under the same name.

Syntax:

```
struct struct_name
{
    data-type var-name;
    data-type var-name;
    .....
};    //end of structure
```

e.g.

```
struct employee{
    int eid,sal;
};
```

LAB 13:

```
#include<iostream>
#include<conio.h>
using namespace std;
struct employee{
    int eid,salary; //public by default
};

int main(){
    employee e;
    e.eid=101;
    e.salary=25000;
    cout<<"Employee ID="<<e.eid<<endl<<"Employee Salary="<<e.salary<<endl;
    getch();
    return 0;
}
```

OUTPUT:

```
Employee ID=101
Employee Salary=25000
```

- C++ has made following **three extensions to C structure**, which makes the C++ structure more powerful than C structure.

1. C++ structures allows us to define functions as a member of structure

```
struct employee{  
.....  
.....  
void setData(){}  
.....  
};
```

2. C++ structures allows us to use the access specifiers private and public.

```
struct employee{  
    private:  
        int eid,sal;  
    public:  
        void setData(){  
            //function body  
        }  
        .....  
};
```

3. C++ structure allows us to use structures similar to that of primitive data types while defining variables.

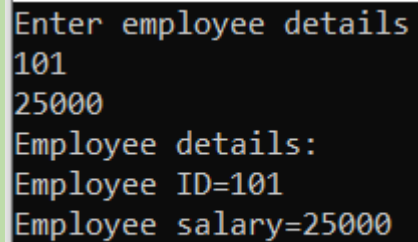
```
struct employee e; //C style  
employee e; //C++ style
```

LAB 14:

```
#include<iostream>
#include<conio.h>
using namespace std;
struct employee{
    private:
        int eid,salary;
    public:
        void setData(){
            cout<<"Enter employee details"<<endl;
            cin>>eid>>salary;
        }
        void getData(){
            cout<<"Employee details:"<<endl;
            cout<<"Employee ID="<<eid<<endl<<"Employee salary="<<salary<<endl;
        }
};

int main(){
    employee e;
    e.setData();
    e.getData();
    getch();
    return 0; }
```

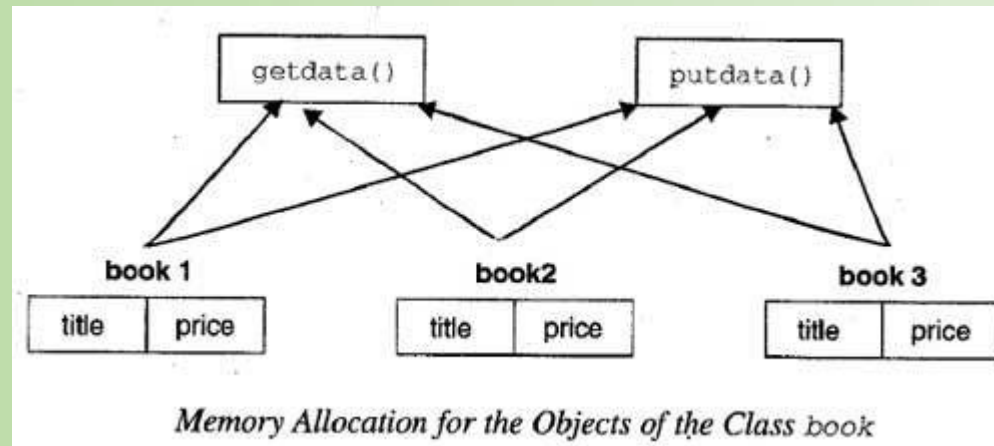
OUTPUT:

A screenshot of a terminal window showing the output of the C++ program. The text is as follows:

```
Enter employee details
101
25000
Employee details:
Employee ID=101
Employee salary=25000
```


3.8 Memory allocation for Objects Theory

- 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.



3.9 Static data members (shared by objects) suyash imp

- If a **data member in a class is defined as static**, then **only one copy of that member is created for the entire class** and is **shared by all the objects of that class**, no matter how many objects are created.
- Hence, **these data members are normally used to maintain values common to the entire class** and are also called class variables.
- **Memory for static data members is allocated at the time of declaration** and is **initialized to zero** when the **first object is created**.
- **We can't put it in the class definition but it can be initialized outside the class using the scope resolution operator (::)** to identify which class it belongs to.

Syntax:

```
class A{  
static int i;  
public:  
//.....};  
int A::i=1;
```

LAB 15:

```
#include<iostream>
#include<conio.h>
using namespace std;
class employee{
    private:
        char name[20];
    public:
        static int objectCount;
        employee(){
            objectCount++;
        }
        void setData(){
            cout<<"Enter employee name: "<<endl;
            cin>>name;
        }
        void getData(){
            cout<<"Employee name: "<<name<<endl;
        }
};
int employee::objectCount=0;
int main(){
    employee e1,e2;
    e1.setData();
    e1.getData();
    e2.setData();
    e2.getData();
    cout<<"Total object created= "<<employee::objectCount<<endl;
    getch();
    return 0;
}
```

OUTPUT:

```
Enter employee name:
Ram
Employee name: Ram
Enter employee name:
Sita
Employee name: Sita
Total object created= 2
```

3.10 Member functions defined outside the class

Suyash imp

- We can **define member functions of the class in two ways: inside the class and outside of the class**. In first case both method declaration and definition is done at the same place inside the class.
- In second case, method is declared inside the class but it's definition is provided outside of the class by using scope resolution operator.
- Defining member functions outside of the class is considered as good programming convention because it keeps class declaration separate from class definition. We can provide our class declaration to others by hiding actual logic of the program(**abstraction**) from them.

General form:

```
return-type class-name :: function-name(argument declaration){  
    //function body  
}
```

- The symbol double colon(::) is called binary scope resolution operator.

LAB 16:

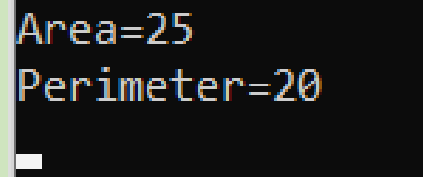
```
#include<iostream>
#include<conio.h>
using namespace std;
class square{
    private:
        int length;
    public:
        void setData(int l){           //method definition inside the class
            length=l;
        }
        int getArea(); //abstraction
        int getPerimeter();
};

int square :: getArea(){               //method definition outside of the class
    return length*length;
}

int square :: getPerimeter(){
    return 4*length;
}

int main(){
    square s;
    s.setData(5);
    cout<<"Area="<<s.getArea()<<endl;
    cout<<"Perimeter="<<s.getPerimeter()<<endl;
    getch();
    return 0;
}
```

OUTPUT:



```
Area=25
Perimeter=20
```

THANK YOU ALL FOR ATTENTION