Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-VI
Lesson: Object Oriented Programming Features
Lesson Developer: Archana Singhal
College/Department: I.P College, University of Delhi

Table of Contents

1. Object Oriented Programming Concepts

C++ is an object oriented Programming language. This means C++ supports features that allow Object Oriented style of Programming. In this chapter we would be discussing about what Object Oriented Programming is? What are the features of an Object Oriented Programming Language? We would also be focusing on basic concepts of object oriented Programming and how these concepts relate to the real world. Object-orientation is also introduced as a concept which makes developing of projects easier.

Learning Objectives

After this lesson you would be able to :

- Explain what is Object Oriented Programming.
- To understand the concept of Abstraction and Encapsulation.
- To define polymorphism and inheritance.
- Enumertae the Advantages of Object Oriented Programming.

1.1 What is Object Oriented Programming?

Object Oriented Programming is a different approach to programming as compared to Procedural Programming. Object Oriented Programming focuses on data rather than functions. It requires a different way of thinking about developing systems. In procedural programming method is used as fundamental building blocks. Instead Object Oriented Programming systems use classes and objects as basic building blocks. Its focus is to model the system with the use of objects, which aims to represent the real world entities.

In Object Oriented Programming, programs are organized as cooperative collection of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes having inheritance relationships.

Object Oriented Programming model has the following elements

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

We will be discussing these features in detail in the subsequent sections.

1.1.1 Abstraction

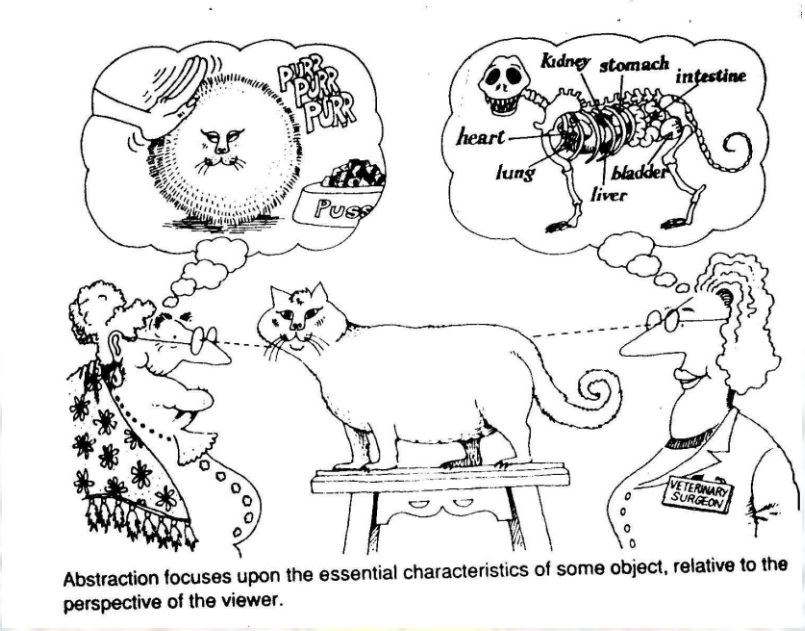
Abstraction is a fundamental way through which human beings deal with complexity. Hoare suggests that "abstraction arises from recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and ignore for the time being the differences". It means using abstraction we can concentrate on a particular aspect while ignoring others which are not relevant in that context.

An Abstraction stands for the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer [3].

Abstraction focuses on only those necessary details that are required in a particular scenario ignoring other details that are not required. For example you are having television at home but you don't require knowing internal circuitry to switch on a television. But for an engineer internal details are as important as others details. So abstraction allows different users to look at same thing differently according to their requirements.

So abstraction looks at a problem by ignoring details that are not directly related to it from a user's perspective and thus reduces the complexity.

Value addition: IMAGE
Heading text : Abstraction



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Source: Object Oriented Design- by Grady Booch, The Benjamin/Cumings Publishing Company

Suppose an organization has to keep track of its employee's details. They will be keeping those details only which are required in the organization. For example Employee's name, address, salary detail, department in which he or she is working. They may also record how many dependants are there if any for insurance purpose. There might be some other details about the employee like what sport does he or she like? Which restaurant he is going on weekends, what kind of food does he or she like? Whether he or she is exercising regularly or not? Organization can ignore these details, because they are not relevant in their domain. But a doctor treating that person would like to keep these details. So a person would be seen differently in different contexts by different people.

1.1.2 Encapsulation

The Encapsulation is one of the most important aspects of an Object Oriented Programming Language.
Encapsulation is combining data and functions together into a single unit called an object.
It leads to the following:

- Access of data contained in an object can be strictly controlled because now data can be accessed only through those functions which are part of the object.
- Functional part or behavioral part can be accessed without knowing the implementation detail of those functions.

Encapsulation and abstraction are complementary to each other. Abstraction ~~compliments~~ complements encapsulation by exposing to the user of an object a clear and well defined interface to that object. So without abstraction there would be no meaning of encapsulation. Similarly abstraction would not be useful without encapsulation. If the object and function would not be together, functions would be operating from outside. So data would not be safe from outside world. It would defeat the actual purpose of abstraction.

Value addition: Did you Know?
Heading text : Encapsulation
Source:

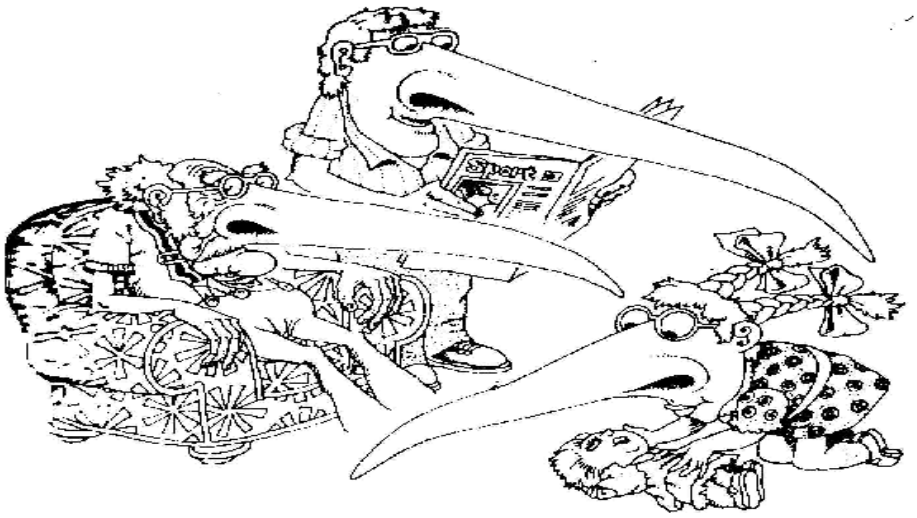
1.1.3 Inheritance

What is Inheritance?

Inheritance is a very important feature of Object Oriented Programming.
Inheritance is a process of deriving new classes from already existing classes.

As the name suggests a class is able to inherit characteristics from other classes. One class can share the structure and behavior defined in one or more classes. Or we can say that a class is able to pass on its state and behaviors to its children. The child class can then extend the state and behaviors to reflect the differences it represents. So child class is a more specialized version of the parent. Thus inheritance is also called generalization/specialization process.

Value addition: Image
Heading text : Inheritance



A subclass may inherit the structure and behaviour of its superclass.

.

Source: Object Oriented Design- by Grady Booch, The Benjamin/Cumings Publishing Company

In this process of Inheritance a class being inherited is also called a base class or super class, while subclass is the name given to the class that is inheriting from the super class. In Figure 44.2, X is a base class and Y, Z and W are the derived classes.

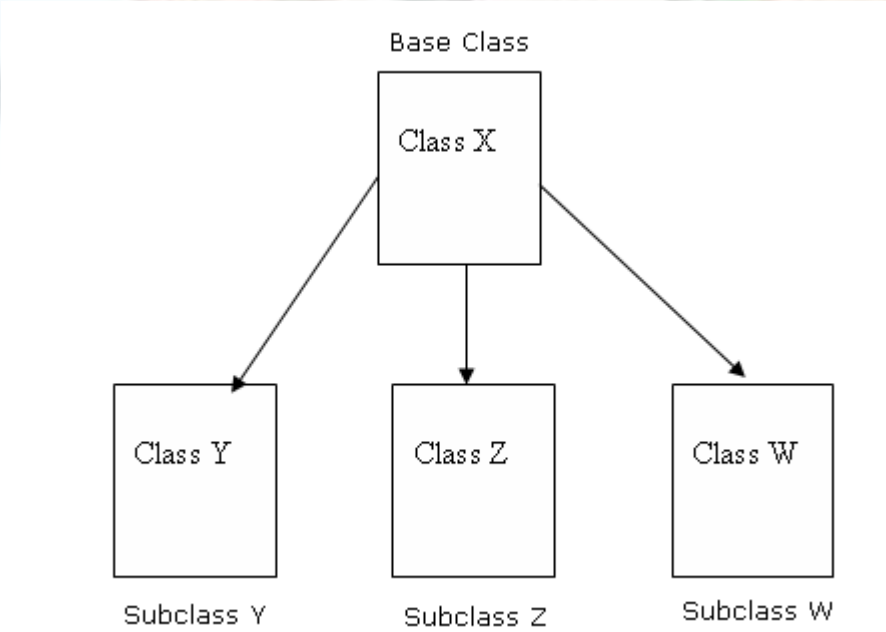


Figure 44.2

To take a more real world example this time we could have a super class called "Person". Its state holds the person's name, address, DateofBirth and has behaviors like EnterDetail() to input the details of a person and Display() to display details of a person. . We could make two new classes that inherit from Person called "Student" and "Worker". They are more specialized versions because although they have names, addresses, and DateofBirth but they also have characteristics that are different from base class as well as from each other. Worker could have a state that holds a job title and place of employment whereas Student might hold data such as Father's Name, area of study and an institution of learning.

So Person is a base class or a super class and Student and Worker are the subclasses of base class Person. They are inheriting characteristics from base class Person and having their own characteristics in addition.

Value addition: FAQs?
Heading text : Inheritance
Body text: Ques: How many subclasses a super class can have? Ans: There is no limitation to how many subclasses a super class can have. You can have as many subclasses as you want. .

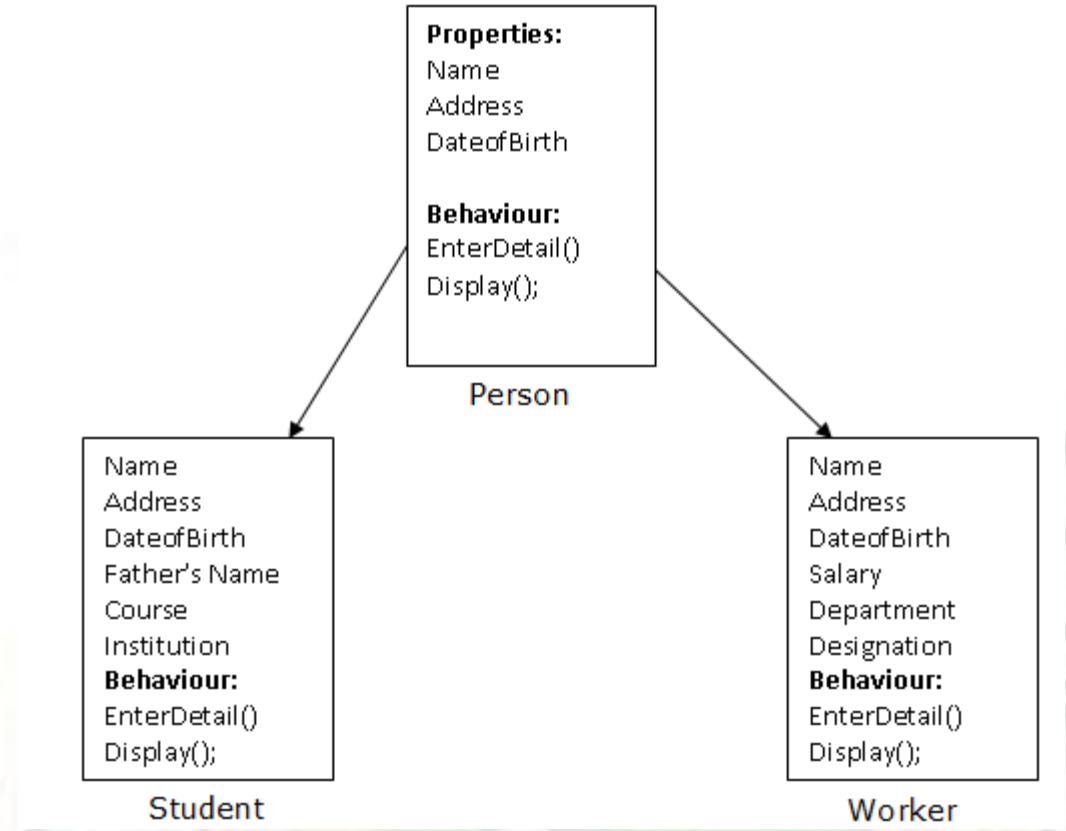


Figure 44.3

Advantages of Inheritance

One of the most important advantages of Inheritance is code reusability. It allows programmers to re-use the previously written code. It will save time while developing new software.

In the Person class example we don't need to create new fields in the Employee and Student class to hold the Name, Address and DateofBirth because we can inherit them from the base class Person. Similarly functions EnterDetail() and Display() are also inherited from class Person and need not to be redefined. In addition to the functions inherited from base class, subclasses can also have their own specialized functions like Student class can have a function Display_Course_Detail() to display details of the course opted by him or her. Similarly Worker class can have a function Display_Salary_Detail()to display details of the salary of an employee.

Value addition: Did you Know?
Heading text :
Body text: A key advantage of using inheritance is code reusability.

Source:

1.1.4 Polymorphism

Polymorphism is a powerful feature of the object oriented programming language C++.
In Polymorphism "Poly" means "many" and "morph" means "form". Thus, polymorphism is the ability to take many different forms at the same time. Polymorphism enables one common interface for many implementations, thus object can act differently under different circumstances.

Types of Polymorphism)

C++ provides two different types of polymorphism.

- Compile-time
- Run-Time

Compile-Time:

The compile-time polymorphism is implemented with the help of function and operator overloading.

Consider the following example:

A+B, where A=6,B=5

If A and B are integers then A+B refers to integer addition.

The same + operator can be used with different meaning if A and B were strings:

A+B , where A= "C++", B="Programming"

If A and B are String type then A+B refers to concatenation of strings.

Similarly If A and B are real numbers then A+B refers to addition of real numbers.

A+B A=10.4,B=5.2

So operator + behaves differently in different contexts such as integer, float or strings. +is said to be overloaed and this phenomena is called operator overloading.

The concept of function overloading is also a branch of polymorphism. Function overloading allows a programmer to define multiple functions with same name but with different arguments.

In polymorphism, a single function functioning in many ways depending upon the usage of the function in different context.

Suppose a programmer wants to draw different shapes such as circles, squares, rectangles, triangles etc. One way is to define a function for each shape with different name and parameters that draws that shape. Another convenient approach is that the programmer can define functions with same name but different signature. This approach leads to different functions executed by the same function call.

This is called compile-time polymorphism because which function to call is decided at compile time only depending on the number or type of argument being passed.

Run-Time Polymorphism:

In run-time polymorphism call to overridden function is decided at run time. This is also termed as late binding since a decision as to which version of the method to invoke cannot be made at compile time. Because there is no way of knowing the actual type of the object whose reference is stored in the reference variable will only be known at run time.

The run-time polymorphism is implemented with inheritance and virtual functions. This will be explained in detail in later sections.
Polymorphism is used to give different meanings to the same concept. This is the basis for Virtual function implementation.

To implement run-time polymorphism the following conditions must apply:

- All different classes must be derived from a single base class.
- The member function must be declared virtual in the base class.

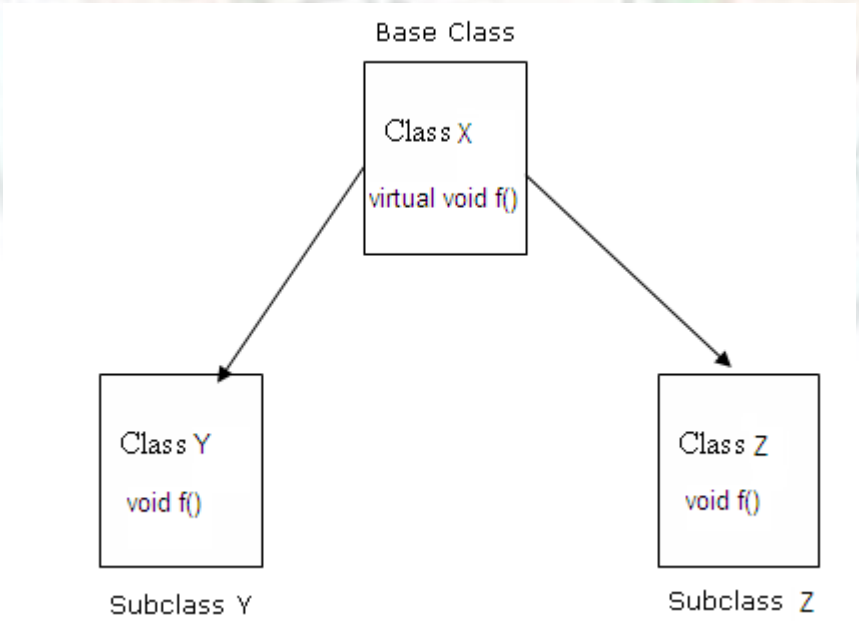


Figure 44.1

In the figure 44.1, the member function f() is made as virtual in the base class. This is redefined in derived classes Y and Z. Which version of f() should be invoked will be decided at run time depending on the address of the object through which we invoke the function. This concept would be discussed in detail later.

Advantages of Polymorphism:

- Provides easier maintenance of applications.
- Helps in achieving robustness in applications.

1.1.5 Advantages of Object Oriented Programming

One of the principal advantages of object-oriented programming over procedural programming is that they enable programmers to create modules that do not need to be changed when a new type of object is added. This leads to the up gradation of smaller systems to larger systems easily. This also makes object-oriented programs easier to modify.

A programmer can also create new objects that inherit several features from existing objects. This will save the time while developing new software.

1.2 Classes and Objects I

Till now we were just discussing the preliminaries. We were writing simple programs to display some data, to perform some calculation and making decisions and so on. Now we will introduce the notion of classes and objects, which is an essential part of an object oriented Programming language like C++. Class is a user defined data type which provides flexibility to a programmer to define a data type of its own, depending on the application requirement.

Learning Objectives

After this lesson you would be able to:

- Explain what is a class and what is an object
- Differentiate between classes and objects
- Define data members and function members of a class
- Create objects of classes

1.2.1 What is a Class?

In C++ data and functions which manipulate that data are encapsulated together into a single unit called class.
So a class is a collection of data as well as associated functions which operate on that data.
A class consists of two parts

- Attributes and
- Behavior

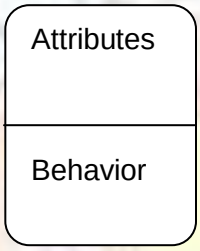


Figure 45.1 Class

Attributes are the properties through which we describe a class. Data members are used to define the attributes of the class.
Behavior of the class is defined through a set of functions called member functions. These member functions define various actions performed on the data of the class.

Let us illustrate this by using a real world example.
Suppose you have to create a bank account. You will go to bank and the bank will require the following information:

- Name of the customer
- Address of the customer
- Type of the account (savings or recurring)

To create an account you have to deposit some initial amount. After creating your account the bank will assign you some unique account number through which it can distinguish your account from another person's. So finally your account will have an account number and a balance also. Thus an account will be described by the following attributes:

- Name of the account holder
- Account Number
- Address
- Type of account
- Balance

The attributes or properties of the person's account distinguish it from other people's accounts.
Now you can do the following tasks with your account once it has been created:

- Deposit money in the account
- Withdraw money from the account
- Check balance in the account

This set of actions defines the behaviour of the account.

We can design a class Bank_account with certain attributes and behaviour to perform this job as shown in figure 45.2.

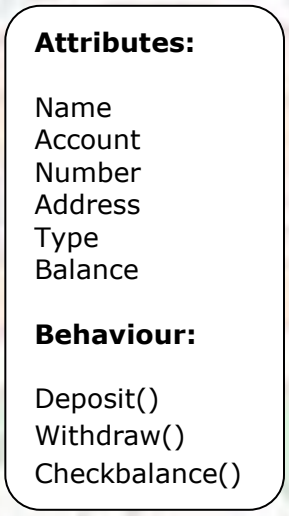


Figure 45.2 Bank _account Class

Similarly if an organization wants to store the data of all its employees an employee class can be defined as follows:

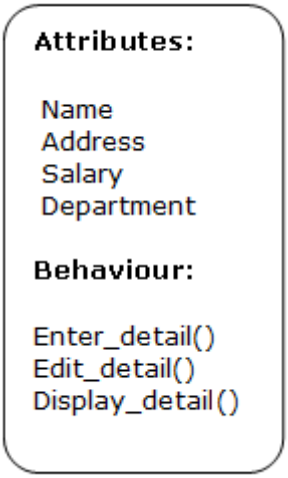


Figure 45.3 Employee Class

Every employee has a name, address, salary and department in which he or she is working. These are called the attributes or properties of the employee through which he or she can be distinguished from other employees. Behaviour of the class will be defined by the set of actions associated with the class. For example Bank will be adding employee's detail, editing employees detail, and displaying employee's details etc., all this functionality will be added as member functions of the class.

Class specification

As defined in earlier section a class is a collection of data and functions that operates on data. It is a user defined data type. It allows the user to define any type of class depending on its requirements. This is also called abstract data typing. For example, complex number data type is not a defined type in C++ but it is possible to define a new type called complex through classes that contains data and required operators to accomplish the tasks associated with complex numbers.

A class in C++ can be defined as follows:

```
class class_name
        {
            private:
                        ┐
                        ├  body
            public:     ┘
```

```
};
```

The word class is a keyword used to define a class and class_name is a variable name. The body of the class is enclosed in braces and terminated by semicolon. The body of the class consists of two sections - private and public. In general all the data of the class is kept in private section and all the functions are defined in public section. The public and private are keywords and define the rights to access various members of the class. These will be explained in detail in later sections. Body of the class is enclosed in curly brackets ({}) and terminated by semicolon (;).

Using this syntax we can define a class account and class Employee which we have discussed in earlier section as follows:

```
class Bank_Account
        {
         private:
                char name[30];
                int accountNo;
                char type[10];
                double Balance;
                char address[60];
         public:
                void create_account();
                void deposit_money();
                void withdraw_money();
                check_balance();

        };

class Employee
        {
         private:
                char name[30];
                int age;
                double salary;
                char address[60];
        public:
                void Enter_detail();
                void Edit_detail();
                void Display_detail();

        };
```

Value addition: FAQs

Heading text : Difference between a class and a structure
Body text: In C++ both a class and a structure are used to hold data as well as functions. The only difference between class and structure is that in a class default access right is private while in structure everything is public by default.

1.2.2 Data Members and Member Functions

The class Bank_account has **data members** - name which is name of account holder, accountno, type, balance and address field. The add_emp(), delete_emp() and print_detail() are the **function members** of the class Bank_account.
Similarly in class Employee data items name, age, salary and address are data members and Enter_detail(), Edit_detail() and Display_detail() are the member functions of class Employee.

Member Function Definition

Definition of the member functions can be given either inside the body of the class or outside the class.

Definition inside the Class:

Suppose, we have to define the member function - print_detail() we can define it in the class body using the following code:

Value addition: Did you know?
Heading text : Member functions
Body text: If member functions are defined inside class they become Inline functions by default. Inline function is the optimization technique used by the compilers. We simply add inline keyword to function prototype to make a function inline. Inline function instruct compiler to insert complete body of the function wherever that function got used in code. **Advantages :-** 1) It does not require function calling overhead. 2) It also save overhead of variables push/pop on the stack, while function calling. 3) It also save overhead of return call from a function. 5) After in-lining compiler can also apply intraprocedural optmization if specified.

This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..
Source:

```
class Employee
        {
         private:
                char name[30];
                int age;
                double salary;
                char address[60];

         public:
                void add_emp();
                void delete_emp();
                void print_detail(){

                                        cout<<endl<<"Name ="<<name;
                                        cout<<endl<<"Age ="<<age;
                                        cout<<endl<<"Salary ="<<salary;
                                        cout<<endl<<"Address ="<<address;

                                    }
        };
```

Function defined inside the class becomes inline by default. That's why only small functions are suggested to be inline.

Definition Outside the class:

Normally the definition of the function is provided outside the class because we don't want every function to be inline. The definition of the function outside the class can be given as follows:

```
Return_type name_of _the _class :: function_name(parameter declaration)
                                {

                                Body

                                }
```

Where :: is called scope resolution operator.

Now the print_detail() function of class Employee will be defined as follows:

```
void  Employee::print_detail()
       {
          cout<<endl<<"Name ="<<name;
          cout<<endl<<"Age ="<<age;
          cout<<endl<<"Salary ="<<salary;
          cout<<endl<<"Address ="<<address;
        }
```

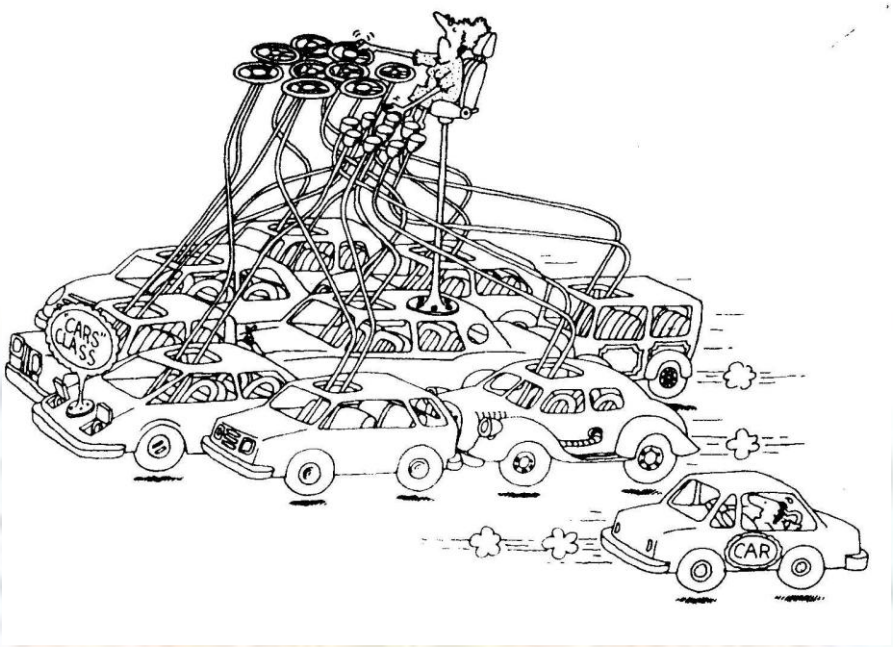
1.2.3 What is an Object?

Just like we use variables to create objects of the standard types integer, char, float and so on, we can create variables of user defined classes also. We can define an object as variable of a class type. An object is an instance of that class.

So far we have not created any object. We have just defined a class which will serve as a template which defines the structure of the objects. Using this structure actual objects are built.

Since all the objects of the class share the same properties we can also define a class as a collection of similar type of objects. Defining a class does not have any significance unless we create objects of that class. A class can be used only by creating objects. Earlier we have defined a class Bank_account. Now let us discuss how to create objects of that class.

Value addition: Image
Heading text : Class
Body text:



A class represents a set of objects that share a common structure and a common behaviour.

Source: Object Oriented Design- by Grady Booch, The Benjamin /Cumings Publishing Company

Creating Objects

Once a class has been defined objects of the class can be created as follows:

Class_name object_name;

Class_name is the name of the class used for creating objects and object_name is the name of the object to be created. For example:

```
#include <iostream>
using namespace std;
int main()
 {
   Bank_account customer1;
   return 0;
 }
```

In the statement
Bank_account customer1;
we create an object of class Bank_account named customer1. Similarly we can create an object of class Employee as

Employee emp;

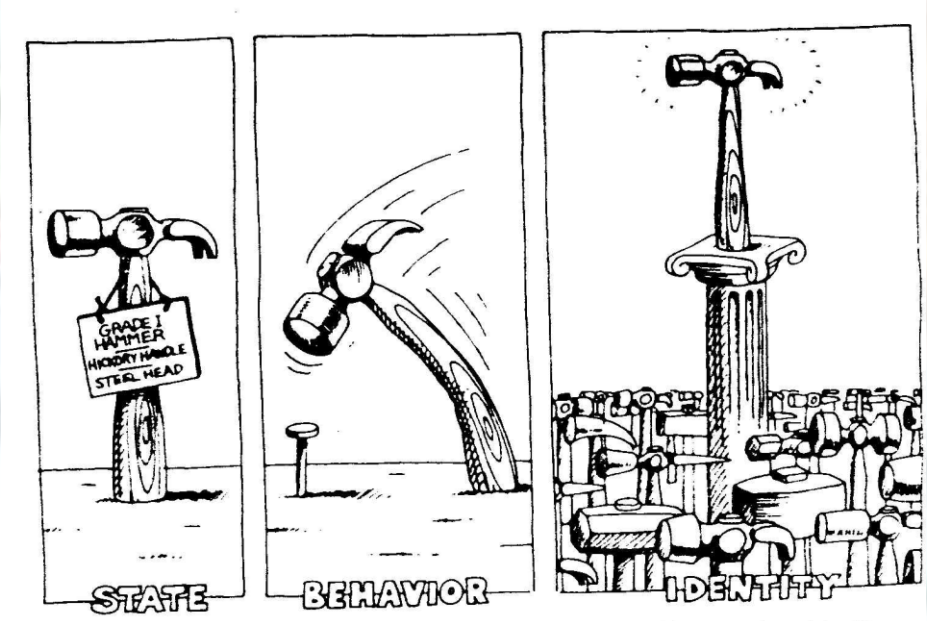
Value addition: Did you know?

Heading text : Object

Body text:

Every object has state behaviour and Identity. The structure and behaviour of similar objects are defined in their common class.

- The state of an object encompasses all of the properties of the objects plus current values of each of these properties**.**
- Behaviour of an object is completely defined by its actions.
- Identity is that property of the object which distinguishes it from all other objects. In Programming languages identity is created using different variables.



An object has state, exhibits some well-defined behavior, and has a unique identity.

Source: Object Oriented Design- by Grady Booch, The Benjamin /Cumings Publishing Company

Memory space for an object is allocated while creating it.
Total space allocated for object e will be the sum of the space required to store all the data members of e. Memory allocation will be discussed in detail in the later section.

Value addition: FAQs
Heading text : Difference between a class and an Object.
Body text: Ques. What is the difference between a class and an object? Ans: A class defines the structure of the object. Based on this structure actual objects are built. This structure will be used by every object of that class. A class has no use without objects. Objects are the instance of the classes through which we can operate in a program.

Value addition: Did you Know?
Heading text :
Body text: Memory space for an object is allocated when they are declared.
Source:

1.3 Classes and Objects II

In the previous section we have introduced the idea of classes and objects. We have also discussed how to define member functions of a class and how to create objects of that class. Now we will discuss in detail how memory is allocated for various objects and how the members of the class i.e. data and functions are accessed outside the class. Public and private access specifies have also been introduced in this chapter. We will be now discussing in detail about accessing public and private members of class.

Learning Objectives

After this lesson you would be able to:

- Explain memory allocation for objects
- Access data members and member functions
- Illustrate public and private access specifier

1.3.1 Memory allocation for Objects

Earlier we have discussed how to create objects. But we have not discussed anything regarding allocation of memory for various members whether data or functions for an object.
Now we will be discussing in detail about memory allocation for objects.

Consider the following example:

```
#include<iostream>
Using namespace std;

Class Example
                {
                   int x,y
                public:
                   void display();
                };

int main()
{
  Example o1;
  Example o2;
  .
  .
  .
 return 0;
}
```

In the function main we are creating two objects o1 and o2 of class Example by writing two statements. We can also create multiple objects in the same statement as follows.

Example o1,o2;

The objects o1 and o2 both have an individual copy of x and y.

Similarly we can create any number of objects of the same class. When objects of the class are created memory is allocated at the same time. Every object of the class will be having its own copy of data members defined in the class.

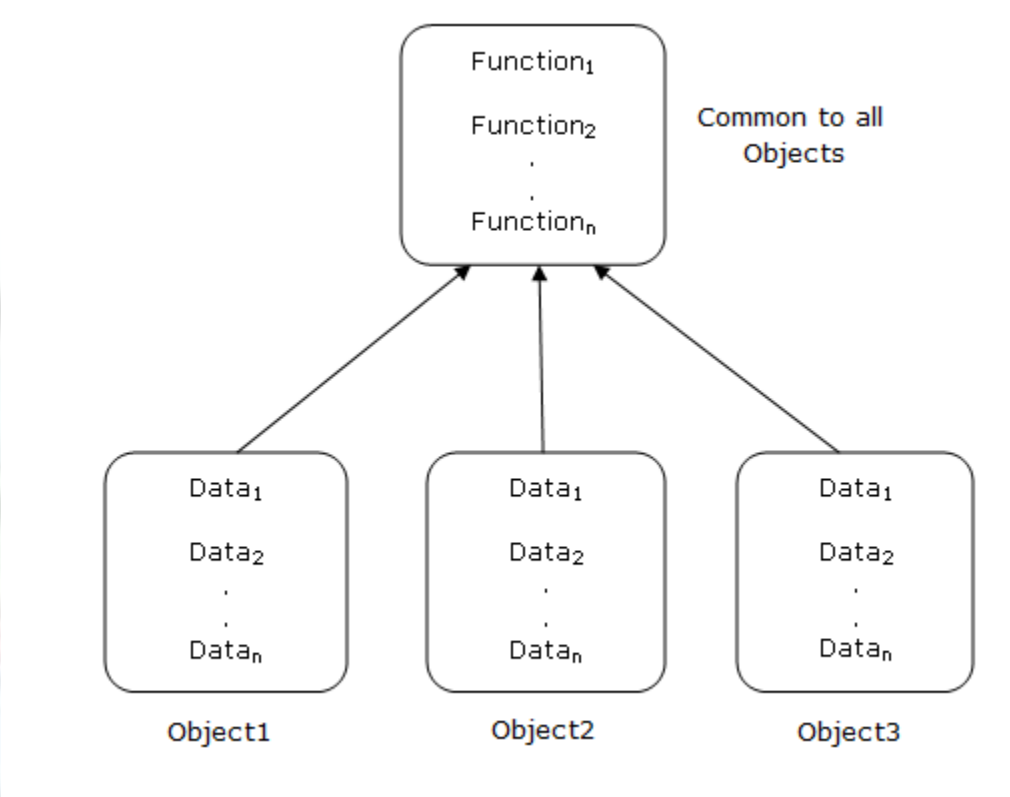
For every object separate memory space is allocated for all the data members but memory for the member functions is allocated once, while specifying the class definition and that is used by every object of that class.

Value addition: Did you Know?

Heading text : Memory allocation of objects

Body text:

For every object separate memory space is allocated for all the data members but memory for the member functions is allocated once, while specifying the class definition and that is used by every object of that class.



Objects sharing class single copy of class functions

1.3.2 Accessing Class Members

Once a class has been defined and objects are created we can use it to access its data and functions. Data of the class can be accessed only through the objects of that class. Members defined in public section behave differently than those defined in private section. Let us discuss how to access public and private members of the class.

Public Access Specifier

Members defined in public section are called public members.

Public members of the class can be directly accessed outside the class using the class object.
Consider the following example:

```
#include <iostream>
using namespace std;

Class Accessrights
{
public:
int i,j;
void display();//displays values of i and j.
}

void Accessrights ::display()
{
 cout<<endl<<"i="<<i;
 cout<<endl<<"j="<<j;

}

int main()
{
Accessrights o;
o.i=10;//ok, since i is public member.
o.j=20;//ok, since j is public member.
o.display();//ok
return 0;
}
```

Output is:
i=10
j=20

In the function main we are creating an object o of class Accessrights and assigning 10 and 20 to i and j respectively. These statements are valid since i and j have been defined in public section.
Similarly member functions defined in public section can be accessed outside the class with class objects.

```
o.display();
```

This statement will make a call to member function display() to display the values of i and j as shown in the output.

Private Access specifier

Private is the default access specifier of the class i.e. if we don't write any keyword in the class than everything is private by default. The private members of the class whether data or function cannot be directly accessed outside the class, they can be accessed only through the public member functions of the class.

Consider the following example:

```
#include <iostream>
using namespace std;

class Example{
                  private:
                    int x,y;
                 public:
                    int z;
                    void display();
                 }
int main()
{
Example O;
O.x=10;//Invalid, since x is private member.
O.y=20 ;//Invalid, since y is private member.
O.z=30;//O.K., since z is public member.
O.display();//O.K.
}
```

In function main() O.x=10 and O.y=20 are invalid statements. Because x and y are defined in private section they cannot be directly accessed outside the class. But they can be accessed through the public member function of the class. We can define a public member function set_xy() in the class to set the values of x and y. Let us demonstrate this with the help of following example:

Value addition: Did you know?
Heading text : Accessing PrivateMembers
Body text: Private member of a class (whether data or function) cannot be accessed directly outside the class with class object. They can be accessed only inside public member functions of the class.

```
Class Example
{
private:
        int x,y;
public:
        void set_xy(int i, int j
 };

void Example:: set_xy(int i, int j)
        {
```

```
x=i;
y=j;
}
```

Similarly if a function is defined in private section it cannot be invoked through class object directly. It can only be used inside public member functions of the class.

Consider the following example:

```
Class Example2
        {
        private:
                int x,y;
                void update(int i, int j);

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

In the above class update() function is defined to update the values of x and y.
But if you try the following code it will give an error message:

```
#include <iostream>
using namespace std;

int main()
{
Example2 e;
e.update()// private member cannot be accessed directly outside the class.
}
```

However function update can be called inside read() function to change the values of x and y by the values read inside the function read().

```
void Example2::read()
{
int i,j;
cin>>i>>j;
Update(i,j);//call without object.

}
```

Value addition: Did you know?
Heading text : Calling a member function inside another

member function.

Body text:

You don't need an object to call a member function inside another member function.

```
#include<iostream>
using namespace std;

class X
        {
           int x,y
        public:

          void set_xy(int i,int j)
                {
                  x=i;y=j;

                 display();//call to display() without object.
                }

        void display()
            {cout<<endl<<"i=""<<i;}
              cout<<endl<<"j=""<<j;}
            }


 int main()
 {
   X O;
   O.set_xy(5,10); //call with object   .
 }
```

1.3.3 The Complex Number Class

Let us design a complex number class to demonstrate the concepts discussed in previous sections. In the following example data has been defined in private section and all member functions have been defined in public section of the class. The set_value() function has been defined to initialize the value of the real and imaginary part of a complex number, since they cannot be directly initialized in main().

Add_complex() and multiply_complex() functions have been defined to add and multiply two complex numbers.

Value addition: Did you know?

Heading text : Default access specifier

Body text:

In a class if you don't write any keyword public or private everything is private by default.

```
class X
        {

          int  i,j;//
          public:
            void display();
        }
```

Is equivalent to

```
class X
        {
          private:
             int  i,j;//
          public:
             void display();
        }
```

```
#include<iostream>
using namespace std;

class Complex
        {
                double Real;
                double Imaginary;

        public:
                void set_value(double, double );
                Complex add_complex(Complex);
                Complex multiply_complex(Complex );
                void print();

        }

void Complex :: set_value(double r, double i)
{

Real=r;
Imaginary =i;
```

```
}

Complex Complex::add_complex( Complex C )
{
   Complex Temp;
   Temp.Real = Real+C.Real;
   Temp.Imaginary = Imag+C.Imaginary;
   return tTmp;
}

Complex Complex::multiply_complex( Complex C)
{
   Complex temp;
   temp.Real  = Real*C.Real - Imaginary*C.Imaginary;
   temp.Imaginary = Real*C.Imaginary + Imag*C.Real;
}


void Complex::print()
{

   cout<<Real<<"+"<<Imaginary<<"i";
}


int main()
{
  clrscr();
  Complex C1, C2, C3; //creates three objects C1,C2 and C3
  C1.set_value(1.0,2.0);
  C2.set_value(3.0,4.0);
  C3.set_value(0.0,0.0);
  cout<< endl<<"First Number is:"
  C1.print();
  cout<<endl<<"Second Number is: "
  C2.print();
  C3 =C1.add(C2);
  cout<<endl<<"Sum of two Number is: "
  C3.print();
  C3 = C1.multiply(C2);
  cout<<endl<<"Multiplication of two Numbers is: "
  C3.print();
  getch();
  return 0;
}
```

The output is:

`

```
First Number is: 1+2i
Second Number is: 3+4i
Sum of two Numbers is: 4+6i
Multiplication of two Numbers is:-5+10i
```

Value addition: Style Tips!!
Heading text : declaring data and member functions

Body text:
 	
When you define a class general practice is to keep every data in private section and member functions in public section. However this is not a rule or recommendation.
 	
class X
{
private:
 	
← data
 	
public:
 	
← functions
}

1. 4 Classes and Objects III

In the earlier sections we discussed the basic concepts about objects such as creation of objects, access specification and memory allocation. Now we will introduce the concepts like how to pass objects as a parameter like basic data types, how to return an object from a function and so on.

Learning Objectives

After this lesson you would be able to:

- Pass objects as function arguments.
- Return objects from functions.
- Create array of objects.

1.4.1 Passing Objects as Function Arguments

Previously we discussed the examples of functions having basic data types as arguments like int, float, char, and so on. t We can also pass objects as arguments to the functions like any other data type. In the present section we will discuss various methods to pass object

as an argument to a function. An object can be passed either by-value or by-reference just like other basic data types.

Using Pass-by-Value

When we pass the object by-value to a function a copy of the object is made. Changes made to that object inside function are actually made in that copy of the object and not in actual object. That's why changes are not reflected back in calling method. Let us consider the following example to illustrate passing the object by-value.

```
#include <iostream>
using namespace std;

class Byvalue
{
        int i,j;
public:
        void set_ij(int x, int y);
        void change(Byvalue o);
        void display();
};

void Byvalue:: set_ij(int x, int y)
{
        i= x;
        j=y;
}
void Byvalue:: change(By_value o)
{
        o.i =o.i+2;
        o.j=o.j*3
}

void Byvalue::display()
{

cout<<"\ni ="<<i;
cout<<"\nj="<<j;
}

int main()
{
   Byvalue O;
   O.set_ij(3,4);
   cout<<"\nBefore Change:";
   O.display();
   O.chage(O);
   cout<<"After Change.";
   O.display();
   return 0;
}
```

Output Is:

Before Change:
i =3
j=4
After Change:
i =3
j=4

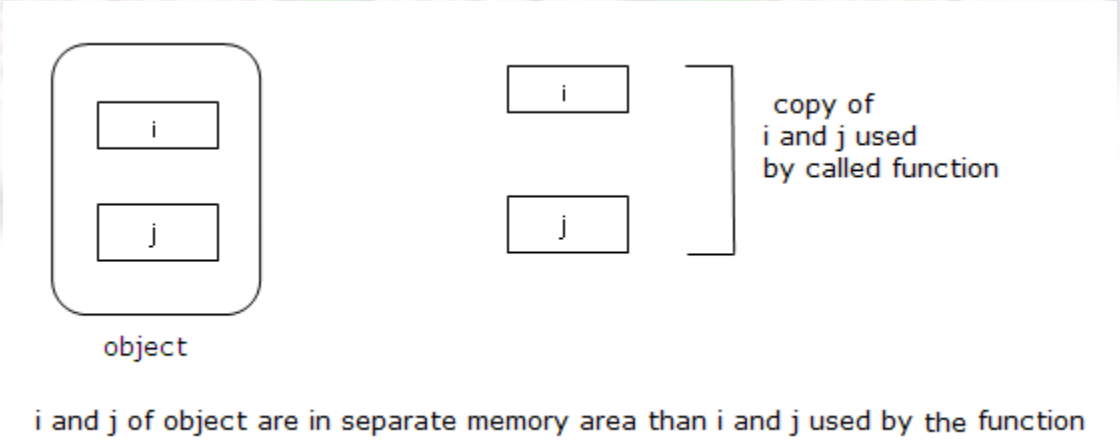
In the above program values displayed before and after calling the change() function are same. Since object O is being passed by-value, so changes made to the object inside the function body are not visible in main. Whatever changes are made inside function body their scope is limited to that function body only.

Value addition: Did you know?

Heading text : Passing objects by-value

Body text:

When an object is passed by-value, changes made to the object inside the function body are not visible in main. Whatever changes are made inside function body their scope is limited to that function body only. Since when an object is passed by-value its copy is made and changes are made in that copy not in the actual arguments.



Using Pass-by-Reference

When we pass the object by-reference to a function a copy of the object is not made. In this case address of the object is passed. So changes made to the object inside function body are made to the actual data of the object, so they are available in main method. Let us consider the following example to illustrate passing the object by-reference.

```
#include <iostream>
using namespace std;
class ByReference
{
        int i,j;
public:
        void set_ij(int x, int y);
        void change(Byvalue &o);
        void display();
};

void ByReference:: set_ij(int x, int y)
{
        i= x;
        j=y;
}

void ByReference:: change(ByReference &o)
{
        o.i =o.i+2;
        o.j=o.j*3;
}

Void ByReference::display()
{

cout<<"\ni ="<<i;
cout<<"\nj="<<j;
}

int main()
{
   ByReference O;
   O.set_ij(3,4);
   cout<<"\nBefore Change:";
   O.display();
   O.chage(O);
   cout<<"\nAfter Change:";
   O.display();
   return 0;
}
```

Out put Is:

```
Before Change:
i =3
j=4
After Change:
i =5
j=12
```

In the above program we are passing object by reference, so changes are made directly at the place where actual argument i and j are stored.

If we write function change() as follows:

```
void ByReference:: change( )
{
        i = i+2;
        j= j*3;
}
```

This is also an example of pass-by-reference because in this case i and j refers to i and j of the object through which we are calling the function. Since address of the object through which we call a function is being passed implicitly as an argument to that function.

So a call to this function will be made by writing

O.change();

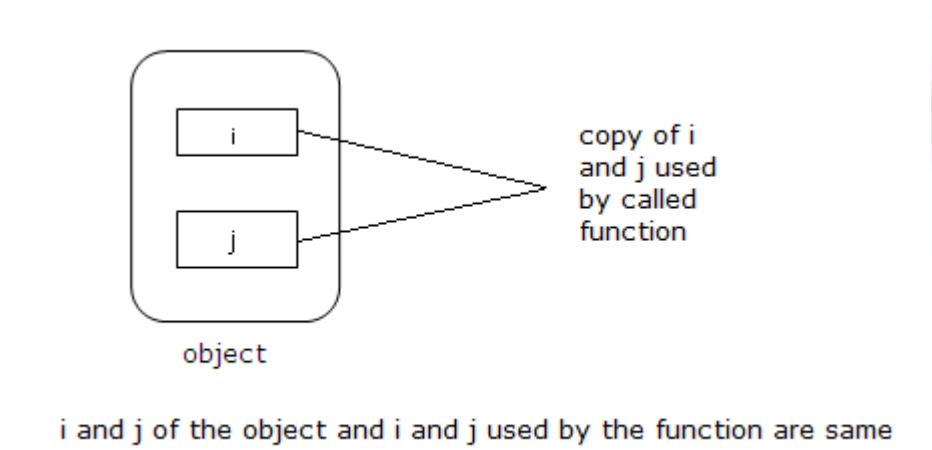
It will change the i and j of object O.

Value addition: Did you know?

Heading text : Passing objects by-reference

Body text:

When an object is passed by-reference, changes made to the object inside the function body are visible in main. Since when an object is passed by-value its copy is not made and changes are made in actual arguments.



1.4.2 Returning Objects from Functions

In the previous section we have described how to pass an object as an argument to a function. It is also possible to return object from a function just like simple data types.

Let us explain it through the following example:

```
#include <iostream>
using namespace std;

class ReturningObjects
{
        int i,j;
public:
        void set_ij(int x, int y);
        ReturningObjects add(ReturningObjects &o1);
        void display();
};

void ReturningObjects:: set_ij(int x, int y)
{
        i= x;
        j=y;
}

ReturningObjects  ReturningObjects:: add(ReturningObjects &o1)

{       ReturningObjects temp;
        temp.i =i+o1.i;
        temp.j=j+o1.j;
        return temp
}

void ReturningObjects::display()
{

cout<<endl<<"i ="<<i;
cout<<endl<<"j="<<j;
}


int main()
{
   ReturningObjects o1,o2,o3;
   o1.set_ij(3,4);
   cout<<endl<<"Values of object o1 are:";
   o1.display();
   o2.set_ij(5,6);
   cout<<endl<<"Values of object o2 are:";
   o2.display();
   o3=o1.add(o2);
   cout<<endl<<"Values of object o3 are:";
   o3.display();
   return 0;
}
```

Output is:

Values of object o1 are:
i=3
j=4

Values of object o2 are:
i=5
j=6

Values of object o3 are:
i=8
j=10

Value addition: FAQs

Heading text : Returning objects from function using reference

Body text:

Ques. Can I return an object from a function using reference?

Ans.
Yes. We can declare a function to return an object using reference type.
If object returned is very large than returning a reference will be more efficient than returning a copy of the object. If we return the object by reference than copying the object to a temporary memory location will not be required.

But we should not try to return references to local object because they are destroyed when control returns from the function. The following program will compile with the warning that reference to local variable t returned.

```
#include <iostream>
using namespace std;

class X
  {
       int i ,j;

    public:
       void Setij(int,int);
       X& Add_objects(X&);
       void display();
   };

void X:: Setij(int x, int y){i=x;j=y;}

X& X:: Add_objects(X& o1)
{
 X t;
 t.i=o1.i+i;
 t.j=o1.j+j;
 return t;  // illegal. Local object cannot be returned as reference.
```

```
    }

     void X:: display(){cout<<endl<<"i="<<i<<endl<<"j="<<j<<endl;}

    int main()
     {
       X o1,o2,o3,o4;
       o1.Setij(3,4);
       o2.Setij(6,8);
       //o3.Setij(4,9);
       o3=o1.Add_objects(o2);
       cout<<endl<<"i and j of object o3 are:"<<endl;
       o3.display();
       o4=o3.Add_objects(o3);
       cout<<endl<<"i and j of object o4 are:"<<endl;
       o4.display();
       getchar();
       return 0;
      }
```

Output is:

```
i and j of object o3 are:
i=9
j=12

i and j of object o4 are:
i=18
j=24
```

If we add function Assign_obj(X& o1) as given below:

```
        X& X ::Assign_obj(X& o1)
            {

             o1.i+=1;
             o1.j+=5;
             return o1;//OK
    }
```

And write the main as follows:

```
          int main()
              {
                X o1,o2;
                o1.Setij(3,4);
                cout<<endl<<"i and j of object o1 are:"<<endl;
                o1.display();
                o2=o2.Assign_obj(o1);
                cout<<endl<<"i and j of object o2 are:"<<endl;
                o2.display();
                getchar();
                return 0;
     }
```

There will be no warning and output will be given as follows:

Output is:

```
i and j of object o1 are:
i=3
j=4

i and j of object o2 are:
i=4
j=9
```

1.4.3 Creating Multiple Objects using Arrays

As we have discussed earlier also we can create more than one objects at the same time in a single statement, similarly we can create multiple objects using array. We can also create an array of objects.

Consider the following example:

```
class Student
{
        int rollno;
        char name[30];
        int age;
public:
        void get_data();
        void display_data();
}
```

In the above example we are defining a class Student to represent a student entity. If have to create ten objects either we use ten variables to represent ten objects as follows:

Student s1,s2,s3. . .s10.

Or we can create an array of ten objects of class students as follows:

Student s[10];

This statement is creating an array s of ten objects referred through index 0 to 9 as index of the array in C++ will always start from 0. For example s[1] stands for second object in the array.

Let us describe this concept with the help of the following example:

```
#include <iostream>
using namespace std;

class Person
{
```

```
        int Id;
        char Name[30];
        int age;
public:
        void get_data();
        void display();
}

void Person::get_data()
   {
      cout<<endl<<"Enter Id:";
      cin>>Id;
      cout<<endl<<"Enter Name:";
      cin>>Name;
      cout<<endl<<"Enter Age:";
      cin>>age;

   }
void Person::display()
   {
      cout<<endl<<"Id: "<<Id;
      cout<<endl<<"Name: "<<Name;
      cout<<endl<<"Age: "<<age;
      cout<<endl;
   }



int main()
{
   Person P[3];
   for(int i=0;i<3;++i)
      P[i].get_data();

   for(int i=0;i<3;++i)
        P[i].display();
   getchar();
   return 0;
}
```

Output is:

```
Enter Id: 1
Enter Name: Rahul
Enter Age: 25

Enter Id: 2
Enter Name: Siddharth
Enter Age: 20

Enter Id: 3
Enter Name: Prabhat
Enter Age: 29
```

```
Id: 1
Name: Rahul
Age: 25

Id: 2
Name: Siddharth
Age: 20

Id: 3
Name: Prabhat
Age: 29
```

1.4 Classes and Objects IV

In the previous sections we have discussed about creating objects, accessing members of the class through objects, passing objects to the function, returning objects from function and so on. When we were creating objects of a class, all of them were having their own copy of class data. But sometimes there is a requirement of a global data that can be accessed and modified by every object of that class. To achieve this static data and functions are used. In the present section we would discuss in detail about static data and static functions.

Learning Objectives

After this lesson you would be able to:

- Describe what is static data and static function?
- Use static data and static functions?
- Explain why static function can't access static data?

1.4.1 Static Data

Data members of the class can also be defined as static. A static data member possesses the following properties:

- Only One copy of the static data member is created and it is shared by all the objects of the class unlike other data members where each object has a separate copy.
- It is initialized outside the class.
- Static data member of the class exist even if class is not instantiated i.e. no object of class has been created.
- It exists throughout the life time of the program.

Static data members are used when all the objects of the class have to share a common data.

Value addition: Did you Know?
Heading text : static data Member

Body text:
 	
A static data member can be of any type except for void.
 	
class A
{
static void i;//wrong. Static data cannot be void.
}

```
#include <iostream>
using namespace std;

class sdata
{
   static int a;
   int b;
public:
  void increment_a();
  void display_a();
};

int sdata:: a=0;

void sdata::increment_a()
{
++a;
}

void sdata:: display_a()
{
cout<<endl<<"value of a ="<<a;
}

int main()
{
 sdata o1,o2,o3;
 o1.increment_a();
 o2.increment_a();
 o3.increment_a();
 o1.display_a();
 o2.display_a();
 o3.display_a();
 return 0;
}
```

Output is:

a=3

a=3
a=3

In the figure 48.1 o1, o2 and o3 are three objects of class sdata. Each object is having its individual copy of b i.e. separate memory allocation of b will be done for each object. But memory allocation for a will be done only once i.e. a single copy of a will be shared by every object of the class.

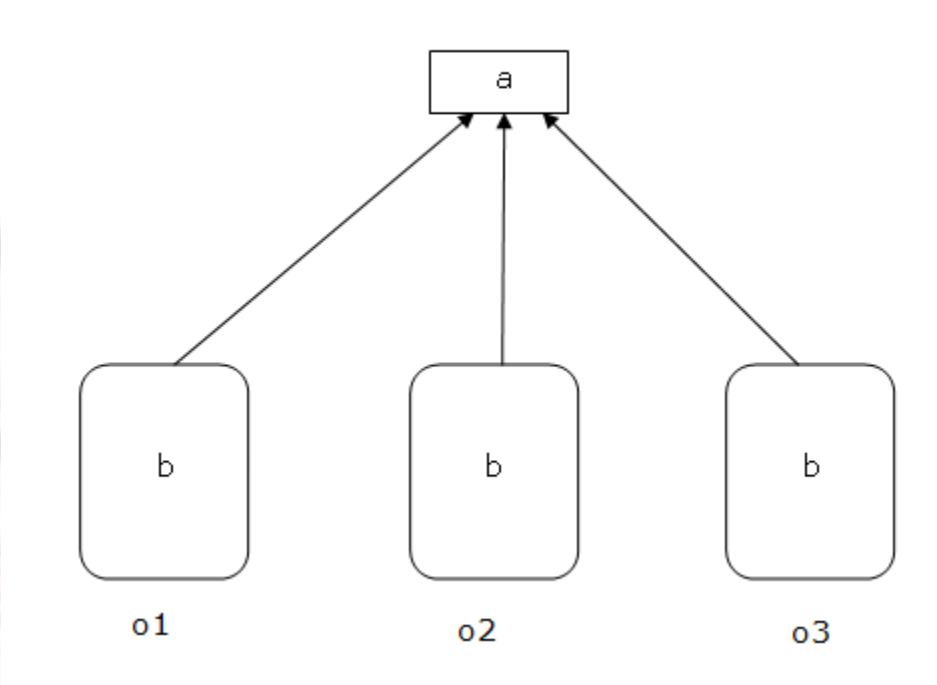


Figure 48.1 static data a shared by objects o1, o2 and o3

In the above program a is a static data member of class sdata. The statement

int sdata::a=0;

Initializes value of a to 0.

The statement o1.increment_a() increments the value of a by 1.

The other two statements

o2.increment_a() and o3.increment_a() also make the changes in the same a, since all three objects are accessing the same copy of a. so value displayed by all three objects are same i.e. 3.

Value addition: FAQs
Heading text : Can I count how many objects of a class has

been created?

Ans: Yes. This can be easily done with the help of static data. Consider the following program:

```
#include <iostream>
using namespace std;

class Count_object
{
   static int n;
public:
  void increment_n();
  void display_n();
};

void Count_object::increment_n()
{
++n;
}

void Count_object:: display_n()
{
cout<<endl<<"Number of objects ="<<n;
}

int Count_object::n=0;

int main()
{
 Count_object o1,o2,o3;
 o1.increment_n();
 o2.increment_n();
 o3.increment_n();
 o1.display_n();
 return 0;
 }
```

In the above program n is static data shared by all the objects. It is initialized by 0. Every object created increments its value by 1. So it will keep track of how many objects have been created.

Same program can be written more efficiently using constructors. Since have not been defined yet, we will discuss this again in later section using constructors.

1.4.2 Static Function

A function can also be made static just like data members of the class. If a function is made static, it can be invoked without making any object of that class.

Value addition: FAQs

Heading text : static data

Body text:

Ques: can we use static data of a class without class object like static function?
Ans: You can use static data without class objects only if they are defined in public section.

```
#include<iostream>
  using namespace std;

   class X
   {
     public:
       static int i;
    };

 int X::i=5;

int main()
 {
  X O;
  cout<<endl<<"i="<<X::i;
  getchar();
  return 0;

 }
```

Output is:

```
i=5;
```

Consider the following example:

```
#include<iostream>
using namespace std;


class X
```

```
{
   static int a;
   int b;
public:

  void increment_a();
  static void display_a();
  void display();
};

int X::a=0;

void X::increment_a()
{
++a;
}

void X:: display_a()
{
cout<<endl<<"value of a ="<<a;
}

void X:: display()
{
cout<<endl<<"value of a ="<<a;
cout<<endl<<"value of b ="<<b;

}

int main()
{
 X o1;
 o1.increment_a();
 X::display_a();
 getchar();
 return 0;
}
```

Value addition: FAQs
Heading text : What is this pointer?
This pointer contains address of the class. This is passed as an implicit argument to the member function of the class. ▪ **this** pointer stores the address of the class instance, to enable member functions of the class to access the data members. ▪ **this** pointer is not counted for calculating the size of the object. ▪ **this** pointers are not modifiable. Consider the following example:

```
class thispointer // class for explaining this pointer
{
   int x;
 public:
   //Function using this pointer

//this is implicitly passed to getdata() as well as setdata()

   int getdata()
   {          return this->x;// this is equivalent to return x;
   }
  //Function without using this pointer
  void setdata(int n)
  {
     x = n;//this is equivalent to this->x=n;
  }
};
```

Thus, a member function can gain the access of data member by either using this pointer or not.

Value addition: FAQs

Heading text : static function

Body text:

Ques: Why static function cannot access non-static data?
Ans: Static functions do not have access to this pointer. This pointer contains address of the object through which a function is being invoked. Without this pointer static function cannot access object specific data.

Here we are calling display_a() as
X::display_a();

Where X is class name and :: is scope resolution operator. So you don't need an object to call a static function. But there is a restriction on static function . A static function can only access static data members of that class.

If we define a static function display1() in class X as follows:

```
static void display1()
 {
   cout<<endl<<"a="<<a;
   cout<<endl<<"b="<<b;//Error

 }
```

Value addition: Did you know?

Heading text : static data

A static data member can also be made constant. You may specify a constant initializer in the static data member's declaration. You still need to define the static member outside the class even if you initialize it in class.

```
  #include<iostream>
   using namespace std;

    class X
    {
      static const int i = 10;
      public:
      void display(){cout<<endl<<"i="<<i;}
    };

const int X::i; // define outside class.

int main()
 {
  X O;
  O.display()
  getchar();
  return 0;
}
```

Output is:

```
i=10
```

This will not compile and give an error message. Since b is a non-static data, it cannot be accessed by static function display1(). Even if we call this function with object, it cannot use non-static data of a class, since a static function does not have access to this pointer. This pointer contains the address of the object through which we make a call to the function. If this pointer is not available, object specific data cannot be accessed.

So if we write

o.display1(); //o is an object of class X.

Even if we call function display1() with object o it cannot access b. since address of o is not available inside display().Anon-static function can have access to static as well as non static data of the class, as this pointer is available inside non-static member function of the class. In the above program function display() prints the value of a which is a static data as well as b, which is non-static data of the class.

1.6 Constructors and Destructors

In previous chapters we have discussed the basic concepts about classes and objects like creating objects, accessing class data through objects, passing objects as arguments to the function, creating array of objects and so on.

Earlier we defined member functions to initialize class data and made explicit call to those functions. In the present section we would discuss about constructors which are used for automatic initialization of objects. We would also be discussing about destructors which are inverse of constructors and used for automatic destruction of objects.

Learning Objectives

After this lesson you would be able to:

- Define constructor and Destructor.
- Explain default constructor and overloaded constructors.
- Design classes using constructors and destructors.
- Perform dynamic memory allocation using constructors

1.6.1 Constructor

A **constructor** is a special member function of the class having the same name as class name. It is used to initialize the objects of the classes. No explicit call is required to invoke constructors unlike other member functions. They are automatically invoked when objects of that class are created. Constructors do not have any return type. Consider the following example:

```
class Example
{
public:
        Example();// Constructor
}
```

Example() is a constructor of class Example. Note that it has been declared in public section because constructors can not be the private members of the class, since they are invoked outside the class whenever an object of that class is created. If constructor would have been private it would not be possible to invoke it outside the class.

We can either have a default constructor or constructor with parameters.

Default Constructor :

A constructor with no arguments is called default constructor.

If no other constructor has been defined by the programmer, the compiler will generate its own default constructor for that class. The compiler generated default constructor initializes the member variables of the class using their respective default constructors.

If no user-defined constructor exists for a class X and one is needed, the compiler implicitly declares a constructor X::X().The compiler will implicitly *define* X() when the compiler uses this constructor to create an object of type X. The constructor will have no argument and a null body.

X(){};//Default constructor generated by compiler

No default constructor is created for a class that has any constant or reference type members.

Like other member functions, constructors are declared within a class declaration. They can be defined inline or external to the class declaration.

Value addition: BEWARE?

Heading text : Constructor

Never declare a constructor in private section of the class.

```
class X
      {
         int i,j;
         X( ){i=0,j=0;}// constructor cannot be declared in private section
    public:
         .
         .
         .
    }
```

Consider the following program showing the implementation of the default constructor of class X.

```
class X
{
      int i,j;
public:
      X( ){i=0,j=0;}//default constructor with inline definition

};
```

This constructor is defined inside the class itself. This definition can be provided outside the class as discussed below.

Constructor defined outside the class:

Definition of a constructor can be given outside the class just like other member functions of the class.

```
class X
{
      int i,j;
public:
      X( );//default constructor
      void display();

};


X::X()
{
i=0;
J=0;
}

void X::display()
{
 cout<<endl<<" i="<<i;
 cout<<endl<<" j="<<j;
}


#include<iostream>
using namespace std;

void main()
{
clrscr();
X O; //invokes default constructor
cout<<endl<<"output is ";
O.display();
getch();
return 0;
}
```

The output is:

```
i=0
j=0
```

In the function main() the statement

```
X O;
```

Invokes the default constructor of the class X and initialize values of i and j to 0 for the object O;

Constructor with Parameters :

Constructor can also take arguments like other member functions of the class.

The following example will explain how to initialize the object using parameterized constructor:

```
class Example
{
      int i,j;
public:
      Example(int,int); //a counstructor with two parameters
      void display();

 };


Example:: Example (int x,int y)
 {
   i=x;
   j=y;
 }

Example::void display()
  {
    cout<<"i="<<i;
    cout<<"j="<<j
  }

#include<iostream>
using namespace std;

void main()
{
Example e(10,20); //a constructor is called with two parameter
cout<<"output is ";
e.display();
getchar();
return 0;
}
```

The output is:

```
i=10
j=20
```

In the above program we are creating an object e and passing two parameters while creating object. This will invoke the constructor automatically and assign the values being passed to i and j.

Constructor with Default argument:

Like other member function in C++ constructors can also have default arguments. Default arguments are those arguments whose default values are provided during function declaration itself. It is not necessary for a programmer to specify those values while calling such a function. If values are not provided while making a call to that function, default values are taken.

Value addition: Did you know?

Heading text : Default Constructor

Body text:

Constructor with all the arguments with default values also works as default constructor.

Consider the following example:

```
class X
   {
        int i,j;
     public:
        X( int a=2,int b=5){i=a,j=b;};//constructor with default value

   };
```

In the above class X constructor have been defined with two parameters and both of them have default values. This constructor will also be invoked when no argument will be passed while creating an object of this class. This is shown as follows:

```
X O;
```

This statement will assign default values 2 and 5 to i and j respectively.

Consider the following example:

```
#include<iostream>
using namespace std;

class X
{
     int i,j;
public:
     X( int a,int b=5){i=a,j=b;}//constructor with a default value

};

int main()
{
X  Obj1(1,2);//initialize i to1 and j to 2
X  Obj2(1);// initialize i to1 and j to 5
```

```
return 0;
}
```

In the class X constructor has been defined with one argument b having a default value 5.

In the statement Obj1(1,2) two arguments have been passed to the constructor. It will initialize i to 1 and j to 2. But in the second statement Obj2(1) only one argument is being passed so 1 will be assigned to i and j will take default value 5 which is provided at declaration time.

Overloaded Constructors:

Since arguments can be passed in constructors they can also be overloaded just like other member functions of the class. We can overload constructor by passing different type of arguments or by passing different number of arguments in it.

To illustrate this concept let us consider the following example:

```
#include<iostream>
using namespace std;

class X
{
      int i,j;
public:
      X(); // default Constructor(no arguments)
      X(int); // constructor with one parameter
      X(int,int); // constructor with two parameter
      void display();

 };

X:: X ( )
 {
   i=10;
   j=10;
 }

X:: X (int x,int y)
 {
   i=x;
   j=y;
 }

X:: X (int x)
 {
   i=x;
   j=x;

//i and j have been initialized with value x.
 }

int main()
```

```
{

X e1; // invokes default constructor X()
X e2(10); // invokes constructor X(int)
X e3(10,20)// // invokes constructor X(int,int)
return 0;
}
```

In class X we have been defined three constructors. The constructor X() is a default constructor, X(int) is a constructor with one argument and X(int,int) is a constructor with two arguments.

All three definitions exist in the class. Depending on number of arguments being passed appropriate constructor will be invoked.

Let us modify complex class discussed previously using constructors.

Program: Complex Class using Constructors

```
#include<iostream>
using namespace std;

class Complex
{
        Double Real;
        Double Imag;
  public:
        Complex();
        Complex(double, double);
        Complex add(Complex);
        Complex Multiply(Complex);
        void print();

};

Complex::Complex()
{
   Real=0.0;
   Imag=0.0;
}
Complex::Complex(double r, double i)
{
   Real  = r;
   Imag= i;
}
Complex Complex::add( Complex C)
{
   Complex t;
   t.Real=Real+C.Real;
   t.Imagl=Imag+C.Imag;
   return t;
```

```
}
Complex Complex::multiply( Complex C)
{
   Complex t;
   t.Real=Real*C.Real-Imag*C.Imag;
   t.Imag= Real*C.Imag+Imag*C.Real;
   return t;

}
void Complex::print()
{

   Cout<<Real<<"+"<<Imag<<"i";
}

int main()
{
  clrscr();
  Complex C1(1.0,2.0), C2(3.0,4.0), C3; //creates three objects C1,C2 and C3
  cout<<"\nfirst Number is:"
  C1.print();
  cout<<"\nSecond Number is: "
  C2.print();
  C3 = C1.add(C2);
  cout<<"\nSum of two Numbers is: "
  C3.print();
  C3 = C1.multiply(C2);
  cout<<"\nMultiplication of two Numbers is: "
  C3.print();
  getch();
  return 0;
}
```

The output is:

First Number is: 1+2i

Second Number is: 3+4i

Sum of two Numbers is: 4+6i

Multiplication of two Numbers is:-5+10i

Value addition: Program
Heading text : Counting how many objects of a class ~~has~~ have been created.
The following program is a modified version of the program given in section 1.4.1. This demonstrates the use of static data to count number of total objects of a class

created so far using constructor.

```
#include <iostream>
using namespace std;

class Count_object
{
   static int n;
public:
  Count_object(){++n;}//increments n by 1;
  static void display_n();
};


void Count_object:: display_n()
{
cout<<endl<<"Number of objects: ="<<n;
}

int Count_object::n=0;

int main()
{
 Count_object o1,o2,o3;
 Count_object()::display_n();//static function can be invoked  without object.
 return 0;
 }
```

Output:

Number of objects:3

In the above program n is static data shared by all the objects. It is initialized by 0. When an object is created constructor is automatically invoked and increments its value by 1. Since three objects have been created n will be incremented thrice. So n will keep track of number of objects created so far.

Value addition: Beware!!

Heading text : Constructor overloading

Don't overload constructors using default parameters.

```
#include<iostream>
```

```
using namespace std;

class X
  {   int a,b;
    public:
      X(){a=0;b=0;}//constructor
      X(int i){a=b=i;}
      X(int i,int j=10){a=i;b=j;}// overloaded constructor with default parameter
      void display(){cout<<endl<<"a="<<a;cout<<endl<<"b="<<b;}
  };

 int main()
 {
   X O(4);//error
   O.display();
   getchar();
   return 0;
 }
```

The definitions X(int i) and X(int i ,int j=20) will be equivalent as they can be called with one parameter int. This program will not compile and complain that call to constructor X(int) is ambiguous.

So don't mix overloading and default parameters together.

Value addition: Did you know?

Heading text : Constructor

Body text:

You cannot declare a constructor as virtual or static, nor can you declare a constructor as const.

Dynamic Memory Allocation using Constructor:

Constructors can also be used for dynamic allocation of memory along with the initialization of object. So objects can be dynamically constructed with the help of constructors. Dynamic allocation of memory is performed using new operator. Consider the following example:

```
#include<iostream>
using namespace std;


class Student
{
        char* Name;
        int Rollno,size;
```

```
  public:
        X(int s=10)
          {
           size=s;
           Name=new char[size];
          }
        X(char *n, int r)
          {
            Name=new char[strlen(n)+1];
            strcpy(Name,n);
            Rollno=r;
          }

        void print()
          {
            cout<<endl<<"Name: "<<Name;
            cout<<endl<<"Rollno: "<<Rollno;

          }

};



int main()
{
  clrscr();
  X s1("Ajay",1), s2("Rajan",2), s3("Sakshi",3); //creates three objects C1,C2 and C3
  cout<<endl<<"First Student is:"<<endl;
  s1.print();
  cout<<endl<<"Second Student is:"<<endl;
  s2.print();
  cout<<endl<<"Third Student is:"<<endl;
  s3.print();
  getchar();
  return 0;
}
```

The output is:

First Student is:

Name: Ajay
Rollno: 1

Second Student is:

Name: Rajan
Rollno: 2

Third Student is:

Name: Sakshi
Rollno: 3

1.6.2 Destructor

Destructors are also the member functions of a class. They are the inverse of constructor functions. They also have the same name as class name preceded by tilde (**~**). For example, the destructor for class X is declared as ~X()

```
class X
{

public:
      X( );//default constructor

     ~X();//Destructor
};
```

Destructor must also be the public member of the class. A destructor also no return type just like constructor.

A destructor is called for a object when that object goes out of scope or is explicitly deleted.

Value addition: Did you know?
Heading text : Constant objects

Body text:
 	
Objects can also be created using key word const.
 	
Consider the following program:
 	
#include<iostream>
 	
using namespace std;
 	
#include<iostream>
using namespace std;
 	
class X
{ int a,b;
public:
X();//constructor
X(int x,int y){a=x;b=y;}// constructor with two parameter
void change(){a=a/2;}
};

```
int main()
{
  const X O(5,10);// O is a constant object.
  O.change();//Error! value of O cannot be changed.
  return 0;
}
```

In the above program statement const X O(5,10);// creates a constant object. Once an object has been defined as constant its value cannot be modified. So the statement O.change(); is invalid and this program will not compile.

```
#include<iostream>
using namespace std;

class X
{
      int a;
  public:
      X( ){a=0;}
      X(int n){a=n;}

     ~X(){cout<<endl<<" Destroying objects….";}//Destructor
};


int main()
{

  {

  X o1;
  cout<<endl<<"Inside Block.";

  }
cout<<endl<<"Outside Block.";
getchar();
return 0;
}
```

Output:

```
Inside Block.
Destroying Objects….
Outside Block.
```

Just like constructors If no user-defined destructor exists in a class and one is required, then compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of the class.

Value addition: Did you know?
Heading text : Destructor
Body text: Destructors cannot be overloaded. Since they do not take any argument.

We can also invoke a destructor explicitly to destroy objects when an object is created with new operator. This is required only when dynamic allocation is being used for some of the data members of the class. To free that space before the program termination an explicit call to the constructor can be made using delete operator. The following example demonstrates this:

```
#include<iostream>
#include<string>
using namespace std;

class X
{
      char *A;
      int size;
  public:
      X(int n )
         {
         size=n; A=new char[n];
         }
      X(int n, char array[])
         {
         size=n;
         A=new char[n];
         strcpy(A,array);
         }
     ~X()
         {delete[] A;cout<<endl<<"Destroying……"}//Destructor
};

int main()
{

  X  * O= new X(12,"Destructors");
  delete O;
  cout<<endl<<"Memory used by object is now available.";
  getchar();
  return 0;
}
```

Output is:

Destroying……"

Memory used by object is now available.

Value addition: Did you know?
Heading text : Destructor
Body text: Constructors and destructors follow the same access rules as member functions. For example, if you declare a constructor with protected access, only derived classes and friends can use it to create class objects.

Summary

- Object Oriented Programming is a method of implementation in which programs are organized as cooperative collection of objects.
- An Abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects.
- Encapsulation is a way to group data into a single entity and hide its internal state and working.
- In C++ Polymorphism provides an ability to use an operator or function in more than one ways.
- Inheritance is a process of deriving new classes from already existing classes. One of the most important advantages of Inheritance is code reusability.
- Class is a collection of data as well as associated functions which operate on that data.
- Data and functions defined in the class are called data members and member functions.
- Member functions either can be defined inside class or outside the class.
- Functions defined inside class become Inline by default.
- An object will be an instance of its class.
- All objects of the class share the same properties.
- Class is also defined as collection of similar type of objects.
- Members of a class whether data or functions can be accessed only through the object of that class.
- Memory space for an object is allocated when they are declared.
- Data and functions defined in the class can be either public or private.
- If we do not use any keyword (neither public nor private) everything in class will be private by default.
- Public data and functions can be directly accessed outside the class with class objects.
- Private members (data or functions) can be accessed only inside public member functions of the class.
- Objects can also be passed as a parameter to a function just like simple data types.
- If Object is passed by-value Changes made to that object inside function are not reflected back in calling method.

- If Object is passed by-reference changes made to that object inside function are reflected back in calling method.
- Objects can also be returned from functions like simple data types.
- We can also create an array of objects just like we create array of basic data types like int, char and so on.
- For a static data member of a class a single copy is created and this is shared by all objects of the class.
- Static data can be accessed without the class object.
- Static data exists throughout the Life time of a program.
- Static function of a class can also be accessed without class objects.
- Static function cannot access non-static data of a class since they do not have access to this pointer.
- A non-static function can access static as well as non-static data of a class.
- Constructors have same name as class name.
- Constructors are used for automatic initialization of objects.
- Constructors can have default arguments and can not be overlodad

- Destructors also have same name as class name and preceded by tilde(~).
- Constructors and destructors do not have return types nor can they return values.
- Destructors are invoked when an object of the class goes out of scope.
- Destructors do not take any arguments. So destructors cannot be overloaded.
- Constructors and Destructors can be defined inline or outside the class.

Exercises

1.1 What are the basic elements of Object Oriented Programming?

1.2 "Abstraction and encapsulation are complementary to each other." Justify this statement.

1.3 What is Inheritance? What are its advantages?

1.4 Write short notes on the following:

i) Data abstraction

ii) Encapsulation

iii) Polymorphism

1.5 Differentiate between run-time and compile-time polymorphism.

1.6 Define classes and objects.

1.7 What is the basic difference between structures and classes in C++?

1.8 Design a class student with name, age, course, address, and college name. Also define member functions to input values for a student, to display details of a student and to modify details of a student.

1.9 Discuss various methods for defining member functions of a class with the help of suitable examples.

1.10 For Exercise 1.8 define the function main and create three objects of class Student.

1.11 Describe the public and private access specifier in a class with the help of suitable

examples.

1.12 State whether the following statements are true or false?

i. Default access specifier of a class is public.

ii. Public member function of a class can access all type of class data whether public or private.

iii. A private member function of a class cannot be directly accessed outside the class.

1.13 Find the error(s) if any in the following code. Also correct the error(s) and give the output.

```
Class X
        {
        private:
                int a,b;
                void update(int a, int b);

        public:
                void read(){
                                cout<<"Enter the value of a:";
                                cin>>a;
                                cout<<"Enter the value of b:";
                                cin>>b;
                              }
                 void print(){
                                cout<<"a="<<a<<" "<<"b="<<b;
                              }
        }
```

```
#include<iostream>
using namespace std;

int main()
{
  X O;
  O.a=10;
  O.b=20;
  O.print();
  return 0;
}
```

1.14 Explain with the help of suitable example, How objects are passed by-value and by-reference to a function?

1.15 Is it possible to return objects from functions? Explain how?.

1.16 Is it possible to create an array containing class objects? Explain with the help of

suitable example.

1.17 State whether the following statements are true or false.

i. Objects can only be passed by reference.

ii. Objects can also be returned from functions.

iii. An array of objects cannot be created.

iv. Every object has a separate copy for its data as well as functions.

1.18 What is the purpose of using static data in a class? Explain with the help of suitable example.

1.19 Why static function cannot use non-static data? Justify your answer.

1.20 Is it possible for static function to access non-static data, if it is invoked with a class object? Explain with the help of suitable example.

1.21 tate whether the following statements are True or False.

i. Non-static function cannot access static data.

ii. Static data can be accessed by every object of the class.

iii. Static data and static function both can be accessed without making class objects.

1.22 Find the error(s) if any in the following code. Also correct the error(s) and give the output.

i)

```
#include<iostream>
using namespace std;

class X
        {
        private:
                int a;
                static int b;

        public:
                 void set(int i){a=I;}
                   static void display()
                        {
                        cout<<" The value of a:"<<a;
                        cout<<"The value of b:"<<b;
                        }

        }

        X:: b=0;
```

```
int main()
{
  X O;
  O.set(5);
  O.display();
  return 0;
}
```

ii)

```
#include<iostream>
 using namespace std;

  class X
  {
    static const int i = 5;

  };

const int X::i;

int main()
 {
  X O;
  cout<<"i="<<X::i;
  getchar();
  return 0;

 }
```

1.23 Define Constructor and Destructor.

1.24 What is default constructor? Explain.

1.25 Can you declare a constructor in private section? Justify your answer.

1.26 Can you overload constructors? Explain with the help of suitable example.

1.27 Why do we need destructors?

1.28 Why destructors cannot be overloaded? Explain.

1.29 State whether the following statements are true or false.

i) It is necessary to define a constructor in every class.

ii) Destructor can be declared in private section.

iii) Destructors can also be overloaded.

iv) Destructors can also be declared as virtual.

v) Constructor can be a static member of the class.

vi) Constructors and destructors can be defined Inline.

Glossary

Array: collection of similar type of data accessed by a single name using an index.
Basic data types: built-in data type of a language.
Class: A class is a collection of data as well as associated functions which operate on that data.
Const : A const data cannot be changed throughout its life time.
Data: collection of information
delete: it is an operator used to free the space used in dynamic initialization of the objects.
Function: Function groups a set of statement into a unit and give it a name which can be invoked any number of times whenever required.
Friend: it is a key word of the language. A friend class or friend function can access private data of the another class.
Inline Function: An Inline function is a function whose code is inserted at the place of call by the preprocessor before compilation.
Keyword: They are reserved words of the language.
Life time: life time of a program is the duration for which a program is running.
new: it is an operator used for dynamic initialization of the objects.
Object: Object is an instance of some class. It is also defined as a class variable.
Private: It is also a key word and defines access right for various members of a class. Private members are accessed differently from Public members.
Procedural Programming: Program is written as a collection of actions (a procedure) that are carried out in a sequence, one after the other.
Public: It's a key word and defines how to access various members of a class. Public members can be directly accessed outside the class.
Robustness: Robustness is the quality of being unfailing in nature.
Scope: The area where a particular data can be accessed defines its scope.

This pointer: this pointer contains the address of the object through which we invoke the function and is implicitly passed to that function.

Virtual Function: A function defined with the keyword virtual and accessed through base class pointer, and redefined in various subclasses of that base class is called a virtual function.

void: it's a key word. When it is applied with a data type it can contain any type of data.

References

[1] **Bjarne Stroustrup**, The C++ Programming Language, , Addison Wesley Publishing Company.

[2] **E. Balaguruswamy,** Object Oriented Programming with C++ (4th ed.)**,** Tata McGraw Hill

[3] **Grady Booch**, Object Oriented Design with Application, The Benjamin/Cumings Publishing Company.

[4] **Herbert Schildt**, C++:The complete Reference(3rd ed),Tata McGraw Hill

[5] **Robert Lafore**, Object Oriented Programming in Microsoft C++, Galgotia Publications

[6] **Stanly B. Lippman**, Josee Lajoie, C++ Primer, 3rd Edition, Addison Wesley