BHARATI VIDYAPEETH DEEMED UNIVERSITY
COLLEGE OF ENGINEERING, PUNE - 43

DEPARTMENT OF COMPUTER ENGINEERING

Lab Manual
Programming Technologies and Tools Laboratory - II

B.Tech (Computer)

(Course 2020)

Sem-II

VISION OF THE INSTITUTE

“To be World Class Institute for Social Transformation through Dynamic Education”

MISSION OF THE INSTITUTE

- To provide quality technical education with advanced equipment’s, qualified faculty Members, infrastructure to meet needs of profession and society.
- To provide an environment conducive to innovation, creativity, research, and entrepreneurial leadership.
- To practice and promote professional ethics, transparency and accountability for social community, economic and environmental conditions.

VISION OF THE DEPARTMENT

“To pursue and excel in the Endeavour for creating globally recognized Computer Engineers through Quality education”.

MISSION OF THE DEPARTMENT

1. To impart engineering knowledge and skills confirming to a dynamic curriculum.

2.To develop professional, entrepreneurial & research competencies encompassing continuous intellectual growth.

3. To produce qualified graduates exhibiting societal and ethical responsibilities in working environment.

PROGRAM EDUCATIONAL OBJECTIVES

1. Demonstrate technical and professional competencies by applying engineering fundamentals, computing principles and technologies.
2. Learn, Practice, and grow as skilled professionals/ entrepreneur/researchers adapting to the evolving computing landscape.
3. Demonstrate professional attitude, ethics, understanding of social context and interpersonal skills leading to a successful career.

PROGRAM SPECIFIC OUTCOMES

PSO 1: To design, develop and implement computer programs on hardware towards solving problems.

PSO 2: To employ expertise and ethical practise through continuingintellectual growth and adapting to the working environment.

PROGRAM OUTCOMES

1. Apply the knowledge of mathematics, science, engineering fundamentals, and computing for the solution of complex engineering problems.
2. Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using computer engineering foundations, principles, and technologies.
3. Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues, and the consequent responsibilities relevant to the professional engineering practice.
7. Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and the need for sustainable development.
8. Apply ethical principles while committed to professional responsibilities and norms of the engineering practice.
9. Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10. Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Apply the engineering and management principles to one’s work, as a member and leader in a team.

12. Recognise the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

GENERAL INSTUCTIONS:

- Equipment in the lab is meant for the use of students. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care.
- Students are required to carry their reference materials, files and records with completed assignment while entering the lab.
- Students are supposed to occupy the systems allotted to them and are not supposed to talk or make noise in the lab.
- All the students should perform the given assignment individually.
- Lab can be used in free time/lunch hours by the students who need to use the systems should take prior permission from the lab in-charge.
- All the Students are instructed to carry their identity cards when entering the lab.
- Lab files need to be submitted on or before date of submission.
- Students are not supposed to use pen drives, compact drives or any other storage devices in the lab.
- For Laboratory related updates and assignments students should refer to the notice board in the Lab.

COURSE NAME: Programming Technologies and Tools Laboratory - II

WEEKLY PLAN:

Week No.	Practical/Assignment Name	Problem Definition
1	Object Oriented paradigm & Basic Concepts of OOP	Study OOP paradigm & OOP Concepts
2	Problem Statements without using class and objects	Implement Cpp Programs without using class & Objects
3	Class & Objects	Implement Programs using class & Objects
4	Call by value & Call by reference	Implement swapping of two numbers using call by value & call by reference
5	Scope Resolution Operator	Implement program using scope resolution operator
6	Friend Function	Implement program using Friend Function
7	Inheritance	
8	Function Overloading	Implement program using Function Overloading
9	Constructor & Destructor	Implement program using Constructor & Destructor
10	File Handling	Implement program using File Handling

EXAMINATION SCHEME

Term Work & Practical: 50
Marks Total: 50 Marks
Minimum Marks required: 18 Marks.

PROCEDURE OF EVALUATION

Each practical/assignment shall be assessed continuously on the scale of 25 marks for assignment. The distribution of marks as follows.And Practical, Oral Performance 25 marks

Sr. No	Evaluation Criteria	Marks for each Criteria	Rubrics
1	**Timely Submission**	**07**	➢ Punctuality reflects the work ethics. Students should reflect that work ethics by completing the lab assignments and reports in a timely manner without being reminded or warned.
2	**Presentation**	**06**	➢ Student are expected to write the technical document (lab report) in their own words. The presentation of the contents in the lab report should be complete, unambiguous, clear, understandable. The report should document approach/algorithm/design and code with proper explanation.
3	**Understanding**	**12**	➢ Correctness and Robustness of the code is expected. The Learners should have an in-depth knowledge of the practical assignment performed. The learner should be able to explain methodology used for designing anddeveloping the program/solution. He/she should clearly understand the purpose of the assignment and its outcome.

LABORATORY USAGE

Students use computers for executing the lab experiments, document the results and to prepare the technical documents for making the lab reports.

OBJECTIVE

The objective of this lab is to make students aware and practice implementation of various data structures by designing algorithms and implementing programs in C++.

PRACTICAL PRE-REQUISITE

- C Programming Language Skills

SOFTWARE REQUIREMENTS

- C++ compiler.

COURSE OUTCOMES

1. Choose and apply different Concepts of OOP
2. Demonstrate the use of functions to solve real world problem
3. Identify and apply the concept of Access Specifiers, Scope Resolution operator, Data Abstraction
4. Compare different types of inheritance to solve given problem.
5. Develop applications with constructor and polymorphism.
6. Develop OOP applications using file Handling and Exception handling.

HOW OUTCOMES ARE ASSESSED?

Outcome	Assignment Number	Level	Proficiency evaluated by
Choose and apply different Concepts of OOP	**1,2,3,4,5,6,7,8,9,10**	**3,2,2,3,3**	Performing Practical and reporting results
Demonstrate the use of functions to solve real world problem	**1,2,3,4**	**3,3,2,2,2,2,2,2,2,2**	Problem definition &Performing Practical and reporting results
Identify and apply the concept of Access Specifiers, Scope Resolution operator, Data Abstraction	**1,2,3,4,5,6,7,8,9,10**	**3,3,3,2**	Performing experiments and reporting results
Compare different types of inheritance to solve given problem.	**1,2,3,4,7**	**3,3,3,3,3,3,3,3,3,3**	Performing experiments and reporting results
Develop applications with constructor and polymorphism	**1,2,3,9**	**3,2,3,3,3**	Performing experiments and reporting results
Develop OOP applications using file Handling and Exception handling	**1,2,3,4,5,6,7,8,9,10**	**3,3,3,3,3**	Performing experiments and reporting results

CONTRIBUTION TO PROGRAM OUTCOME

Course Outcome	Program Outcomes												PSOs	
Choose and apply different Concepts of OOP	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
Demonstrate the use of functions to solve real world problem	3	2	1	2	1	1	1	1			1	1	3	
Identify and apply the concept of Access Specifiers, Scope Resolution operator, Data Abstraction	1	1												
Compare different types of inheritance to solve given problem.	1	2	2	1	1								2	
Develop applications with constructor and polymorphism	1	3	1	2			1						1	
Develop OOP applications using file Handling and Exception handling	2	2	1				1				1	1	1	

DESIGN EXPERIENCE GAINED

The students gain moderate design experience by creating algorithms using data structures and convert that algorithm to executable code.

LEARNING EXPERIENCE GAINED

The students learn both soft skills and technical skills while they are undergoing the practical sessions. The soft skills gained in terms of communication, presentation and behavior. While technical skills they gained in terms of programming.

LIST OF PRACTICAL ASSIGNMENTS:

1.Study Of Object Oriented paradigm & Basic Concepts of Object Oriented Programming
2. Write a program for following problem Statements without using class and objects i) Write a program to check Whether number is perfect or not using C++ ii) Write a program to check whether number is palindrome or not using C++ iii) Write a program to find Fibonacci series using C++
3. Write a Program for following problem Statementsusing Class & Objects i) Design Calculator ii) Read and print student's details
4.Write a Program to implement concept of call by value & call by reference in C++
5.Write a Program to implement Concept of Scope resolution Operator in C++
6.Write a Program to implement Concept of Friend Function in C++
7.Write a Program to implement Concept of Inheritance in C++
8.Write a Program to implement Concept of Function Overloading in C++
9.Write a Program to implement Concept of Constructor & Destructor in C++
10.Write a Program to implement Concept of File handling in C++

ASSIGNMENT NO.1

Aim: **Study of Object Oriented paradigm & Basic Concepts of Object Oriented Programming**

Theory:

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.

Many of the most widely used programming languages (such as C++, Object Pascal, Java, Python etc.) are multi-paradigm programming languages that support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include Java, C++, C#, Python, PHP, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, and Smalltalk.

There are a few principle concepts that form the foundation of object-oriented programming −

Object

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class

When you define a class, you define a blueprint for an object. 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.

Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

Questions:

- What are the advantages of OOP ?
- What are the basic concepts of OOP ?
- What is the difference between Procedural Programming and OOP ?

ASSIGNMENT NO.2

Aim:

- **Write a program to check whether number is perfect or not using C++**
- **Write a program to check whether number is palindrome or not using C++**
- **Write a program to find Fibonacci series using C++**

1.PERFECT NUMBER

A positive number is said to be perfect number if it is equal to the sum of its divisors.

For example,
The divisors of 6 are 1,2 and 3 and 1+2+3 = 6. Thus 6 is a perfect number.
The divisors of 28 are 1,2,4,7 and 14 and 1+2+4+7+14 = 28. Thus 28 is a perfect number.

Program:

/*check whether Perfect Number*/

```
#include <iostream>

using namespace std;

int main()
{
        intnum ,sum=0 ,i;

        cout<<"ENTER A NUMBER: ";
        cin>>num;

        for(i=1;i<=(num/2);i++)
       {
                if(num % i==0)
               {
                        sum = sum + i;
               }
       }
```

```
        cout<<endl;

         if(sum==num)
        {
                 cout<<num<<" IS A PERFECT NUMBER";
         }
         else
         {
                cout<<num<<" IS NOT A PERFECT NUMBER";
         }

         return 0;
}
```

2.PALINDROME

This program takes an integer from user and that integer is reversed.If the reversed integer is equal to the integer entered by user then, that number is a palindrome if not that number is not a palindrome.

Program:

/*check whether Palindrome Number*/

```
#include<iostream>
usingnamespacestd;

int main()
{
int n, num, digit, rev = 0;

cout<<"Enter a positive number: ";
cin>>num;

     n = num;

do
     {
digit = num % 10;
rev = (rev * 10) + digit;
num = num / 10;
     } while (num != 0);
cout<<" The reverse of the number is: "<< rev <<endl;

if (n == rev)
cout<<" The number is a palindrome";
else
cout<<" The number is not a palindrome";

return0;
}
```

3.FIBONACCI

Fibonacci Series is in the form of 0, 1, 1, 2, 3, 5, 8, 13, 21, ...To find this series we add two previous terms/digits and get next term/number.

Program:

/*print Fibonacci series*/

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

void main()
{
inti,no, first=0, second=1,next;
clrscr();
first=0;
second=1;
cout<<"Enter nubmer of terms for Series: ";
cin>>no;
cout<<"Fibonacci series are: \n";
for(i=0;i<no;i++)
        {
        cout<<"\n"<<first;
        next= first + second;
        first= second;
        second=next;
        }
getch();
}
```

ASSIGNMENT NO.3

Aim: **Write a Program for following problem Statements using Class & Objects**

- **Design Calculator**
- **Read and print student's details**

1.CALCULATOR

Program:

```
/*design a calculator*/

#include <iostream>
using namespace std;

class test
{
   public:intx,y,sum;
           floatfsum;
   void first()
   {
      cout<<"Enter first number\n";
   }
   void second()
   {
      cout<<"Enter second number\n";
   }
   int add()
   {
      sum=x+y;
      cout<<sum;
   }
   int minus()
   {
      sum=x-y;
      cout<<sum;
   }
   Intmult()
   {
      sum=x*y;
      cout<<sum;
   }
   int divide()
```

```
    {
        sum=x/y;
        cout<<fsum;
    }
};
```

2.READ AND PRINT STUDENTS' DETAILS

<u>Program:</u>

/*read and print students' details*/

```
#include <iostream>
usingnamespacestd;

class student
{
        private:
                char name[30];
                introllNo;
                int  total;
                floatperc;
        public:
                //member function to get student's details
                voidgetDetails(void);
                //member function to print student's details
                voidputDetails(void);
};


//member function definition, outside of the class
void student::getDetails(void){
        cout<< "Enter name: " ;
        cin>> name;
        cout<< "Enter roll number: ";
        cin>>rollNo;
        cout<< "Enter total marks outof 500: ";
        cin>> total;

        perc=(float)total/500*100;
}

//member function definition, outside of the class
void student::putDetails(void){
        cout<< "Student details:\n";
```

```
	cout<< "Name:"<< name << ",Roll Number:" <<rollNo<< ",Total:" << total << ",Percentage:"
<<perc;
}

int main()
{
	studentstd;			//object creation

	std.getDetails();
	std.putDetails();

	return 0;
}
```

Questions:

- How to create object in C++ ?
- How classes reduce complexity of codes ?

ASSIGNMENT NO.4

Aim: **Write a Program to implement Concept of call by value & call by reference in C++**

Theory:

CALL BY VALUE

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
// function definition to swap the values.
void swap(int x,int y)
{
int temp;

temp= x;
   x = y;
   y = temp;

return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example –

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

// function declaration
void swap(int x, int y);

int main () {
   // local variable declaration:
int a = 100;
int b = 200;

cout<< "Before swap, value of a :" << a <<endl;
cout<< "Before swap, value of b :" << b <<endl;
```

```
   // calling a function to swap the values.
swap(a, b);

cout<< "After swap, value of a :" << a <<endl;
cout<< "After swap, value of b :" << b <<endl;

return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result −

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

CALL BY REFERENCE

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int&x, int&y)
 {
int temp;
temp = x;
   x = y;
   y = temp;
return;
}
```

For now, let us call the function **swap()** by passing values by reference as in the following example −

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

// function declaration
void swap(int&x, int&y);

int main () {
   // local variable declaration:
int a = 100;
int b = 200;

cout<< "Before swap, value of a :" << a <<endl;
cout<< "Before swap, value of b :" << b <<endl;

   /* calling a function to swap the values using variable reference.*/
swap(a, b);

cout<< "After swap, value of a :" << a <<endl;
cout<< "After swap, value of b :" << b <<endl;

return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result −

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

ASSIGNMENT NO.5

Aim: **Write a Program to implement concept of Scope resolution operator in C++**

Theory:

In C++, scope resolution operator is **::**. It is used for following purposes.

1) To access a global variable when there is a local variable with same name:

Program:

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

int x;  // Global x

int main()
{
  int x = 10; // Local x
  cout<< "Value of global x is " << ::x;
  cout<< "\nValue of local x is " << x;
  return 0;
}
```

2) To define a function outside a class.

Program:

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

class A
{
public:

   // Only declaration
   void fun();
};

// Definition outside class using ::
void A::fun()
{
   cout<< "fun() called";
}

int main()
{
  A a;
  a.fun();
  return 0;
}
```

3) To access a class's static variables.

Program:

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

class Test
{
    static int x;
public:
    static int y;

    // Local parameter 'a' hides class member
    // 'a', but we can access it using ::
    void func(int x)
    {
      // We can access class's static variable
      // even if there is a local variable
      cout<< "Value of static x is " << Test::x;

      cout<< "\nValue of local x is " << x;
    }
};

// In C++, static members must be explicitly defined
// like this
int Test::x = 1;
int Test::y = 2;

int main()
{
    Test obj;
    int x = 3 ;
    obj.func(x);

    cout<< "\nTest::y = " << Test::y;
    return 0;
  }
```

4) In case of multiple Inheritance:

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

Program:

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

class A
{
protected:
    int x;
public:
    A() { x = 10; }
};

class B
{
protected:
    int x;
public:
    B() { x = 20; }
};

class C: public A, public B
{
public:
   void fun()
   {
      cout<< "A's x is " << A::x;
      cout<< "\nB's x is " << B::x;
   }
};

int main()
{
    C c;
    c.fun();
    return 0;}
```

ASSIGNMENT NO.6

Aim: **Write a Program to implement Concept of Friend Function in C++**

Theory:

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows −

Program:

```
class Box
{
double width;
public:
double length;
friend void printWidth( Box box );
voidsetWidth( double wid );
};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne −

```
friend class ClassTwo;
```

Consider the following program −

```
#include <iostream>

using namespace std;
```

```
class Box {
double width;

public:
friend void printWidth( Box box );
voidsetWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid ) {
width = wid;
}

// Note: printWidth() is not a member function of any class.
voidprintWidth( Box box ) {
   /* Because printWidth() is a friend of Box, it can
directly access any member of this class */
cout<< "Width of box : " <<box.width<<endl;
}

// Main function for the program
int main() {
   Box box;

   // set box width without member function
box.setWidth(10.0);

   // Use friend function to print the wdith.
printWidth( box );

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Width of box : 10
```

ASSIGNMENT NO.7

Aim:Write a Program to implement Concept of Inheritance in C++

Theory:

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base**class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include<iostream>

usingnamespacestd;

// Base class
classShape{
public:
voidsetWidth(int w){
width= w;
}
voidsetHeight(int h){
height= h;
}
protected:
int width;
int height;
};

// Derived class
classRectangle:publicShape{
public:
intgetArea(){
return(width * height);
}
};


int main(void)
{
RectangleRect;

Rect.setWidth(5);
Rect.setHeight(7);

// Print the area of the object.
cout<<"Total area: "<<Rect.getArea()<<endl;

return0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way –

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.
-

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private**members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private**members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax −

class derived-class: access baseA, access baseB....

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −

```
#include<iostream>

usingnamespacestd;

// Base class Shape
classShape
{
public:
voidsetWidth(int w)
{
width= w;
}
voidsetHeight(int h)
{
height= h;
}

protected:
int width;
int height;
};

// Base class PaintCost
classPaintCost
{
public:
intgetCost(int area)
{
return area *70;
}
};

// Derived class
ClassRectangle:publicShape,publicPaintCost
```

```
{
public:
intgetArea()
{
return(width * height);
}
};

int main(void)
{
RectangleRect;
int area;

Rect.setWidth(5);
Rect.setHeight(7);

area=Rect.getArea();

// Print the area of the object.
cout<<"Total area: "<<Rect.getArea()<<endl;

// Print the total cost of painting
cout<<"Total paint cost: $"<<Rect.getCost(area)<<endl;

return0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35

Total paint cost: $2450
```

ASSIGNMENT NO.8

Aim: Write a Program to implement concept of Function Overloading in C++

Theory:

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

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

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

classprintData
 {
public:
void print(inti)
{
cout<< "Printing int: " <<i<<endl;
      }
void print(double  f)
 {
cout<< "Printing float: " << f <<endl;
      }
void print(char* c)
 {
cout<< "Printing character: " << c <<endl;
      }
};

int main(void)
 {
printDatapd;

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

   // Call print to print float
pd.print(500.263);
```

```
   // Call print to print character
pd.print("Hello C++");

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Printing int: 5

Printing float: 500.263

Printing character: Hello C++
```

ASSIGNMENT NO.9

Aim: Write a Program to implement Concept of Constructor & Destructor in C++

Theory:

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor −

```
#include <iostream>


using namespace std;


class Line {

public:

voidsetLength( double len );

doublegetLength( void );

Line();  // This is the constructor

private:

double length;

};


// Member functions definitions including constructor

Line::Line(void) {

cout<< "Object is being created" <<endl;

}
```

```
void Line::setLength( double len ) {
length = len;
}
double Line::getLength( void ) {
return length;
}

// Main function for the program
int main() {
   Line line;

   // set line length
line.setLength(6.0);
cout<< "Length of line : " <<line.getLength() <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Object is being created
Length of line :
```

Parameterized Constructor

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example −

```
#include <iostream>
```

```
using namespace std;

class Line {

public:

voidsetLength( double len );

doublegetLength( void );

Line(double len);  // This is the constructor


private:

double length;

};


// Member functions definitions including constructor

Line::Line( double len) {

cout<< "Object is being created, length = " <<len<<endl;

length = len;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}


// Main function for the program

int main() {
   Line line(10.0);

   // get initially set length.
cout<< "Length of line : " <<line.getLength() <<endl;

   // set line length again
line.setLength(6.0);
cout<< "Length of line : " <<line.getLength() <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Using Initialization Lists to Initialize Fields

In case of parameterized constructor, you can use following syntax to initialize the fields −

```
Line::Line( double len): length(len) {
cout<< "Object is being created, length = " <<len<<endl;
}
```

Above syntax is equal to the following syntax −

```
Line::Line( double len) {
cout<< "Object is being created, length = " <<len<<endl;
length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows −

```
C::C( double a, double b, double c): X(a), Y(b), Z(c) {

   ....

}
```

The Class Destructor

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor −

```
#include <iostream>


using namespace std;

class Line {

public:

voidsetLength( double len );

doublegetLength( void );

Line();   // This is the constructor declaration

      ~Line();  // This is the destructor: declaration


private:

double length;

};
```

```
// Member functions definitions including constructor
Line::Line(void) {
cout<< "Object is being created" <<endl;
}
Line::Line(void) {
cout<< "Object is being deleted" <<endl;
}
void Line::setLength( double len ) {
length = len;
}
double Line::getLength( void ) {
return length;
}

// Main function for the program
int main() {
   Line line;

   // set line length
line.setLength(6.0);
cout<< "Length of line : " <<line.getLength() <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Object is being created

Length of line : 6

Object is being deleted
```

ASSIGNMENT NO.10

Aim:Write a Program to implement Concept of File handling in C++

Theory:

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. Andifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.

5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstreamoutfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstreamafile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

```
#include<fstream>
#include<iostream>
usingnamespacestd;

int main (){
char data[100];

// open a file in write mode.
ofstreamoutfile;
outfile.open("afile.dat");

cout<<"Writing to the file"<<endl;
cout<<"Enter your name: ";
cin.getline(data,100);

// write inputted data into the file.
outfile<< data <<endl;

cout<<"Enter your age: ";
cin>> data;
cin.ignore();

// again write inputted data into the file.
outfile<< data <<endl;

// close the opened file.
outfile.close();

// open a file in read mode.
ifstreaminfile;
infile.open("afile.dat");

cout<<"Reading from the file"<<endl;
infile>> data;

// write the data at the screen.
cout<< data <<endl;

// again read the data from the file and display it.
infile>> data;
cout<< data <<endl;

// close the opened file.
infile.close();

return0;
}
```

When the above code is compiled and executed, it produces the following sample input and output −

```
$./a.out

Writing to the file

Enter your name: Zara

Enter your age: 9

Reading from the file

Zara

9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)

fileObject.seekg( n );

// position n bytes forward in fileObject

fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject

fileObject.seekg( n, ios::end );

// position at end of fileObject

fileObject.seekg( 0, ios::end );
```