****

**LAB MANUAL**

**(OBJECT ORIENTED PROGRAMMING)**

## Submitted By:

**Name: Hafsa Sajid**

**Roll no: bsf-2201552**

**Submitted To:**

**Mr. Zafar Iqbal Karmani**

## *CONTENTS*

|  |  |  |
| --- | --- | --- |
| **Sr No.** | **Title** | **Pg No.** |
| 1. | Introduction | 3 |
| 2. | Functions | 6 |
| 3. | Introduction to OOP | 11 |
| 4. | Class | 13 |
| 5. | Class Objects | 14 |
| 6. | Encapsulation | 14 |
| 7. | Abstraction | 15 |
| 8. | Access Specifiers | 36 |
| 9. | Constructors | 39 |
| 10. | Destructors | 39 |
| 11. | Inheritance | 41 |
| 12. | Polymorphism | 50 |
| 13. | Exception | 59 |
| 14. | Abstract Class | 61 |
| 15. | Concreate Class | 61 |
| 16. | Friend Function | 63 |

**INTRODUCTION**

**T**he objective of this lab manual is to give students step-by-step examples to become familiar with programming concepts, design, and coding.

**F E AT U R E S**

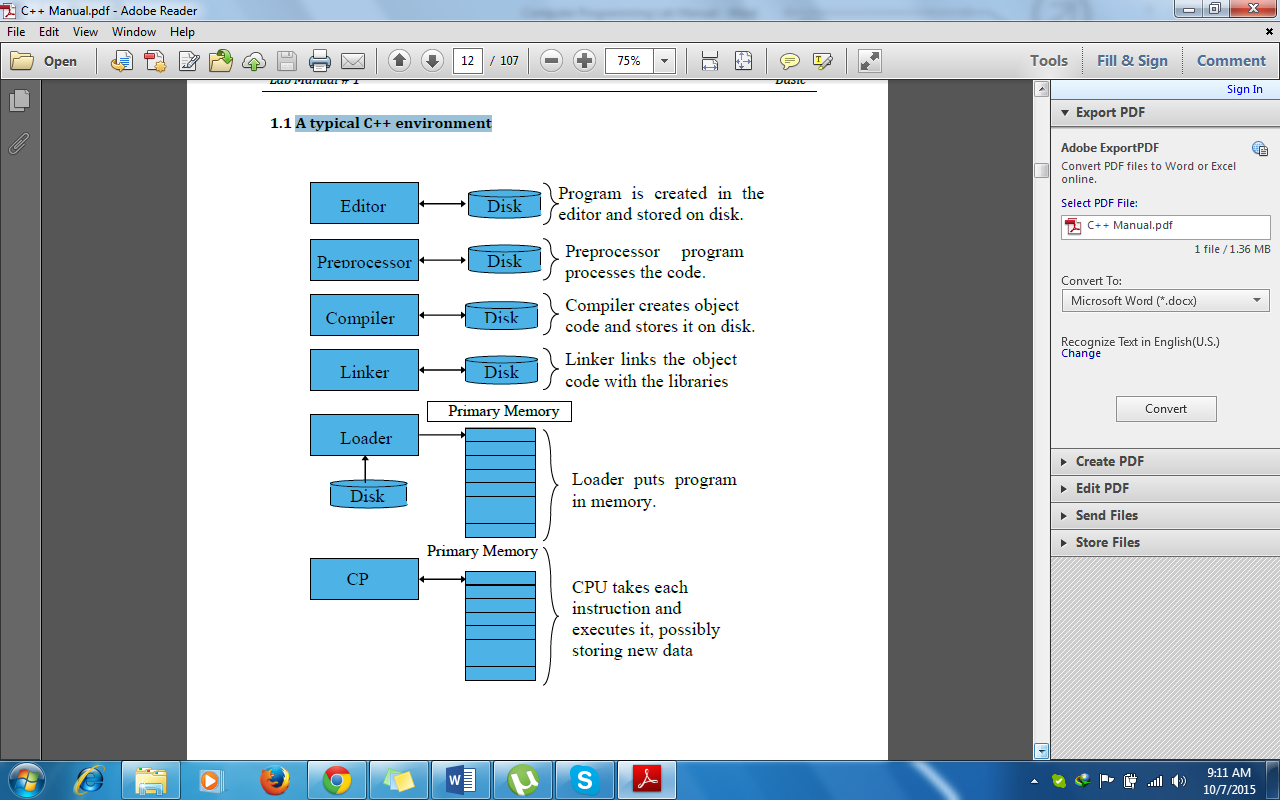
To ensure a successful experience for instructors and students alike, this lab manual includes the following features:

* **Lab Objectives**—Every lab has a brief description and list of learning objectives.
* **Materials required—**Every lab includes information on software you will need to complete the lab.
* **Completion Times**—Every lab has an estimated completion time so that you can plan your activities more accurately.
* **Activity Sections**—Labs are presented in manageable sections; where appropriate, additional Activity Background information is provided to illustrate the importance of a particular activity.
* **Step-by-Step Instructions**—Every lab provides steps to enhance technical proficiency; some labs include Critical Thinking exercises to challenge students.
* **Review Questions**—Some labs include review questions to help reinforce concepts presented in the lab.
* **SOFTWA R E REQUIREMENTS** —Computer running Windows Windows XP, Recommended compiler is Borland C++ 3.0 version, Turbo C++ 3.0 version or Microsoft Visual Studio 2008, 2010 etc.

**COMPLETING THE LAB ASSIGNMENTS**

* Some lab assignments require written answers to complete an exercise, while others are programming assignments that require you to work with a C++ compiler.
* Check with your instructor for instructions on completing the written assignments. For example, you can print pages directly from the appropriate editor, and then write directly on the page.
* To complete the programming assignments, use the compiler that your instructor recommends or requires.
* Print all the documentation assigned, including program code, program prompts, input, and output displayed on the screen, input files, and output files.
* You can submit your written answers and the printed documentation with a lab cover sheet for grading.
* If your instructor requires an electronic copy of your work, e-mail the completed assignment to your instructor or include a removable disk with your work.
* Your instructor will tell you what is needed, but be sure to submit the .cpp.
* To provide program documentation, compile and run your program, copy the prompts, input, and output (if appropriate), and paste them as a block comment at the end of your program.
* Use the Copy and Paste features of your C++ program development kit to do so. After you paste the comment in the program, either print the program file from your text editor or submit the program file to your instructor electronically.

**A typical C++ environment**



**Functions**

**Functions**

* + In C++, a function is a group of statements that is given a name, and which can be called from some point of the program.
  + In function, a large program is divided into subprograms and each subprogram is solved separately and at the end, all the solutions of these subprograms are joined to find the solution of the large program. These subprograms are called sub-routines.

**Syntax of Function**

Return-type name (parameter1, parameter2, ...)

{

Statements;

}

**Local Variables and Global Variables**

* + **Local Variables:** The variables that are declared within the main function but can’t access by everyone is called local variables.
  + **Global Variables:** The variables that are declared outside the main function are called global variables. These functions can be accessed by anyone and can be used in the program where it is needed.

## Program # 01

**Write a C++ program to implement a simple calculator with addition, subtraction, multiplication, and division operations**.

#include<iostream>

using namespace std;

void sum(int x,char op,int y)

{

int sum;

sum=x+y;

cout<<"the sum of nums is : "<<sum<<endl;

}

void sub(int x,char op,int y)

{

int sub;

sub=x-y;

cout<<"the sub of nums is : "<<sub<<endl;

}

void mul(int x,char op,int y)

{

int mul;

mul=x\*y;

cout<<"the mul of nums is : "<<mul<<endl;

}

void div(int x,char op,int y)

{

int div;

div=x/y;

cout<<"the div of nums is : "<<div<<endl;

}

int main(){

int select;

cout<<"main menu"<<endl;

cout<<"press 1 for sum"<<endl;

cout<<"press 2 for sub"<<endl;

cout<<"press 3 for mul"<<endl;

cout<<"press 4 for div"<<endl;

cin>>select;

int a,b;

char op;

cout<<"enter 1st num : ";

cin>>a;

cout<<"enter 2nd num : ";

cin>>b;

if(select == 1)

{

sum(a, '+', b);

}

else if(select == 2)

{

sub(a, '-', b);

}

else if(select == 3)

{

mul(a, '\*', b);

}

else if(select == 4)

{

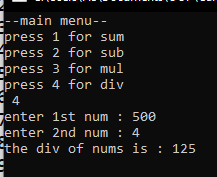
div(a, '/', b);

}

Return 0;

}

**Output:**

****

**Introduction**

**To OOP**

## Object-Oriented Programming:

Object-Oriented Programming (OOP) in C++ is a programming paradigm that uses classes and objects to structure code. It emphasizes concepts such as:

* **Classes and Objects**
* **Encapsulation**
* **Abstraction**
* **Inheritance**
* **Polymorphism**

**What is programming paradigm?**

A programming paradigm is a fundamental style or approach to programming that guides the structure and organization of code. It defines the principles and methods used to design and implement computer programs.

Examples of programming paradigms include procedural programming, object-oriented programming (OOP), functional programming, and declarative programming. Each paradigm has its own set of principles and features that influence how developers write and organize code.

**Classes**

**Classes**

* A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" (plan) for creating objects.
* A way to map real world objects into programming constructs
* A C++ class is composed of methods and variables where:
* Attributes/properties are mapped with variables
* Behaviors/operations are mapped with methods

**Object in classes:**

In C++, object is anything tangible that hold data and behavior. To use objects in a class:

* Define a class with member variables and functions.
* Create objects by declaring variables of the class type.
* Access member variables and call member functions using the dot operator (‘.’).

**Access Specifiers:**

* Access specifiers define how the members (attributes and methods) of a class can be accessed.
* In C++, there are three access specifiers:
* **public** - members are accessible from outside the class
* **private** - members cannot be accessed (or viewed) from outside the class
* **protected** - members cannot be accessed from outside the class, however, they can be accessed in inherited classes

**Encapsulation:**

Encapsulation is a process of combining data members and functions in a single unit called class. This is to prevent the access to the data directly, the access to them is provided through the functions of the class. It is one of the popular features of Object-Oriented Programming (OOP) that helps in data hiding.

**Abstraction:**

Abstraction is one of the features of Object-Oriented Programming, where you show only relevant details to the user and hide irrelevant details. **For example**, when you send an email to someone you just click send and you get the success message, what actually happens when you click send, how data is transmitted over network to the recipient is hidden from you (because it is irrelevant to you).

## Example Program

**Write a program to create a class of car having multiple functions**

#include<iostream>

#include<string.h>

using namespace std;

int main();

class car{

//data members

private:

int totalcars;

public:

string name,model;

int height;

void inputdata()

{

cout<<"car name"<<endl;

cin>>name;

cout<<"car model"<<endl;

cin>>model;

cout<<"height of car"<<endl;

cin>>height;

}

void setcars(){

cout<<"total no. of cars"<<endl;

cin>>totalcars;

}

void display(){

cout<<"the name of car is "<<name<<endl;

cout<<"the model is "<<model<<endl;

cout<<"it's height is"<<height<<endl;

}

};

main()

{

car t1;

t1.inputdata();

t1.setcars();

t1.display();

return 0;

}

**Output**

## Program # 01

**Write a C++ program to find the sum of two numbers entered by the user**.

#include <iostream>

using namespace std;

class Add {

private:

int num1;

int num2;

int sum;

public:

void inputNos() {

cout << "Enter the first number: ";

cin >> num1;

cout << "Enter the second number: ";

cin >> num2;

}

void calcSum() {

sum = num1 + num2;

cout << "Sum of " << num1 << " and " << num2 << " is: " << sum <<endl;

}

};

int main() {

Add calcu;

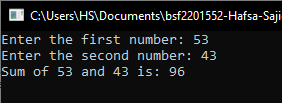
calcu.inputNos();

calcu.calcSum();

return 0;

}

**Output:**



## Program # 02

**Write a C++ program that calculates the factorial of a given positive integer**.

#include<iostream>

using namespace std;

class Fact{

private:

int num;

int factorial=1;

public:

void calFact()

{

cout<<"Enter a num"<<endl;

cin>>num;

if(num== 0 || num == 1){

factorial = 1;

}

else{

for (int i = 1; i <= num; i++) {

factorial = factorial \* i;

}

}

}

void show() {

cout<<"Factorial is: "<<factorial;

}

};

int main()

{

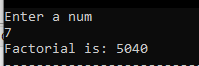
Fact fac;

fac.calFact();

fac.show();

}

**Output:**

****

## Program # 03

**Write a C++ program to check if a number is even or odd.**

#include <iostream>

using namespace std;

int main() {

cout << "Enter an integer: ";

int num;

cin >> num;

if (num % 2 == 0) {

cout << num << " is even." << endl;

} else {

cout << num << " is odd." << endl;

}

return 0;

}

**Output:**

****

## Program # 04

**Write a C++ program to find the largest among three numbers entered by the user**

#include <iostream>

using namespace std;

class Find {

public:

int findLargest(int num1, int num2, int num3) {

int largest = num1;

if (num2 > largest) {

largest = num2;

}

if (num3 > largest) {

largest = num3;

}

return largest;

}

};

int main() {

Find f;

int num1, num2, num3;

cout << "Enter the first no: ";

cin >> num1;

cout << "Enter the second no: ";

cin >> num2;

cout << "Enter the third no: ";

cin >> num3;

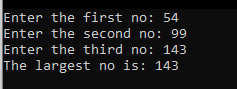
int largest = f.findLargest(num1, num2, num3);

cout << "The largest no is: " << largest <<endl;

return 0;

}

**Output**



## Program # 05

**Write a C++ program to reverse a string entered by the user.**

#include <iostream>

#include <string>

using namespace std;

class Reverse {

public:

string reverseString(const string& input) {

string reversed;

for (int i = input.length() - 1; i >= 0; i--) {

reversed += input[i];

}

return reversed;

}

};

int main() {

Reverse rev;

string input;

cout << "Enter a string: ";

cin >> input;

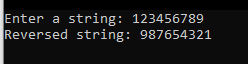
string reversed = rev.reverseString(input);

cout << "Reversed string: " << reversed <<endl;

return 0;

}

**Output**

****

## Program # 06

**Write a C++ program that converts temperature from Fahrenheit to Celsius**

#include<iostream>

using namespace std;

class temp{

private:

int F;

int celsius;

public:

void inputdata(){

cout<<"enter the temperatur in fahrenheit scale:";

cin>>F;

}

void displayoutput(){

celsius=(F-32)\*5/9;

cout<<"the temperature in celsius scale:"<<celsius<<endl;

}

};

main(){

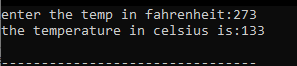
temp T;

T.inputdata();

T.displayoutput();

}

**Output:**

****

## Program # 07

**Write a C++ program to implement a simple calculator with addition, subtraction, multiplication, and division operations**.

#include<iostream>

using namespace std;

class calculator{

private:

int x;

int y;

int sum;

int subs;

int mult;

int divi;

public:

void inputdata(){

cout<<"enter the first integer:";

cin>>x;

cout<<"enter the second integer:";

cin>>y;

}

void displaysum(){

sum=x+y;

cout<<"the sum of two integer:"<<sum<<endl;

}

void displaysub(){

subs=x-y;

cout<<"the sub of two integers:"<<subs<<endl;

}

void displaymult(){

mult=x\*y;

cout<<"the mult of two numbers:"<<mult<<endl;

}

void displaydivi(){

divi=x/y;

cout<<"div of two numbers:"<<divi<<endl;

}

};

main(){

calculator C;

int choose;

cout<<"..............MAIN MENU........."<<endl;

cout<<"press 1 to sum the integers:"<<endl;

cout<<"press 2 to substract the integers:"<<endl;

cout<<"press 3 to multiply the integers:"<<endl;

cout<<"press 4 to divide the integers:"<<endl;

cin>>choose;

C.inputdata();

if(choose==1){

C.displaysum();

}

else if(choose==2){

C.displaysub();

}

else if(choose==3){

C.displaymult();

}

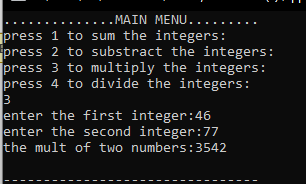
else if(choose==4){

C.displaydivi();

}

}

**Output**

****

## Program # 08

**Write a C++ program to generate a Fibonacci sequence of a given length**

#include<iostream>

using namespace std;

class fibo{

private:

int n;

int t1=0;

int t2=1;

int nextterm=0;

public:

void inputdata(){

cout<<"enter the integer:";

cin>>n;

}

void displayoutput(){

cout<<"....The fibonacci series:..."<<endl;

for(int i=1;i<=n;++i){

if(i==1){

cout<<t1<<", ";

continue;

}

if(i==2){

cout<<t2<<", ";

continue;

}

nextterm=t1+t2;

t1=t2;

t2=nextterm;

cout<<nextterm<<", ";

}

}

};

main(){

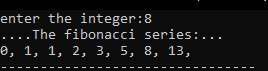
fibo F;

F.inputdata();

F.displayoutput();

}

**Output**

****

## Program #09

**Write a C++ program to find the prime factors of a number**

#include<iostream>

using namespace std;

class prime{

private:

int n;

int count;

public:

void inputdata(){

cout<<"enter the integer:";

cin>>n;

}

void displayoutput(){

for(int a=1;a<=n;a++){

if(n%a==0){

count++;

}

}

if(count==2){

cout<<"number is prime:"<<endl;

}

else{

cout<<"number is not prime:"<<endl;

}

}

};

main(){

prime P;

P.inputdata();

P.displayoutput();

}

**Output**

****

## Program # 10

**Write a C++ program to determine whether a given year is a leap year or not.**

#include<iostream>

using namespace std;

class leapyear{

private:

int year;

public:

void input(){

cout<<"enter the year:";

cin>>year;

}

void displayoutput(){

if((year%4==0 && year%100==0) || (year%400==0))

cout<<"year is leap:";

else

cout<<"year is not leap:";

}

};

main(){

leapyear Y;

Y.input();

Y.displayoutput();

}

**Output**

## Program # 11

**Define a class to represent a bank account. Include the following**

**members:**

**Data members:**

**1) Name of the depositor**

**2) Account number**

**3) Type of account**

**4) Balance amount in the account.**

**Member functions:**

**1) To assign initial values**

**2) To deposit an amount**

**3) To withdraw an amount after checking the balance**

**4) To display name and balance.**

**Write a main program to test the program.**

#include<iostream>

using namespace std;

class bank\_account{

public:

char name[30];

int num;

char type[30];

int balance;

void initial\_value()

{

cout<<"The name of the person: "<<endl;

cin.getline(name, 30);

cout<<"Account Type is: "<<endl;

cin.getline(type, 30);

cout<<"Account num is: "<<endl;

cin>>num;

cout<<"Account Balance is : "<<endl;

cin>>balance;

}

void deposite(int amount)

{

balance += amount;

cout<<" Amount after deposite: "<<balance<<endl;

}

void withdraw(int amount){

if(amount>balance){

cout<<"Amount is not enough"<<endl;

}

else{

balance -=amount;

cout<<"Amount after with draw"<<balance;

}

system("pause");

}

void display()

{

cout<<"The name of depositor : "<<name<<endl;

cout<<"Account number is: "<<num<<endl;

cout<<"Account type is: "<<type<<endl;

cout<<"Account Balance is : "<<balance<<endl;

}

};

main(){

bank\_account H;

int amount;

int choose;

H.initial\_value();

cout<<"-------------Atm----------"<<endl;

cout<<"press 1 to deposite"<<endl;

cout<<"press 2 to with draw"<<endl;

cin>>choose;

switch(choose){

case 1:

cout<<"account Deposite: "<<endl;

cin>>amount;

H.deposite(amount);

break;

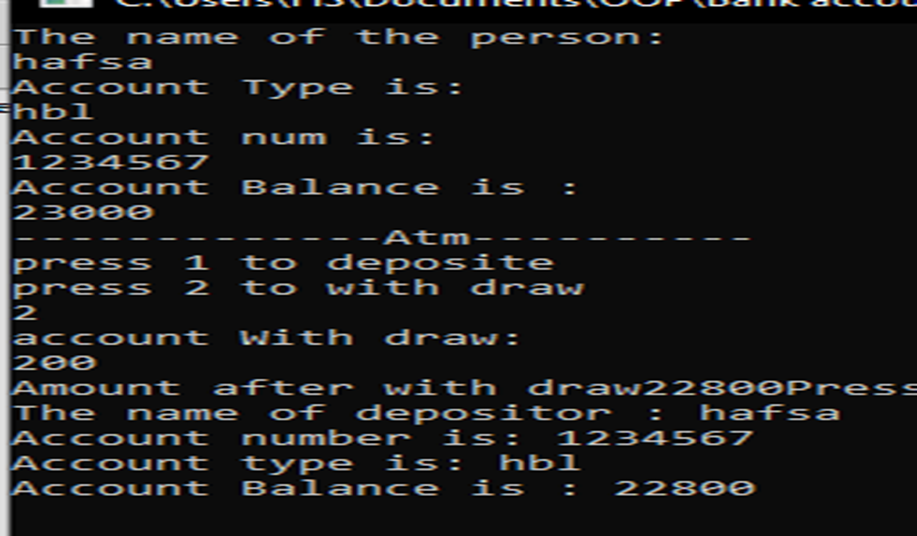
case 2:

cout<<"account With draw: "<<endl;

cin>>amount;

H.withdraw(amount);

}

****H.display();

}

**Output:**

## Access control

Access control, in the context of programming and object-oriented languages like C++, Java, and others, refers to the rules and mechanisms that regulate how classes, methods, and data members can be accessed and used. It ensures the encapsulation and protection of data and functionality. There are three primary access control modifiers:

**Public:** Members (methods and data) declared as public are accessible from any part of the program. This access modifier provides the least restriction and is often used for interface elements meant to be accessed externally.

**Private:** Members declared as private are only accessible from within the class in which they are defined. This provides strong encapsulation and data hiding.

**Protected:** Members declared as protected are accessible within the class itself and its subclasses (derived classes). This access control modifier is used to provide limited access to derived classes while still maintaining some level of encapsulation

## Program

**Define a base class Base and a derived class Derived. Test the accessibility of different members (public, private, protected) from the derived class**

#include <iostream>

using namespace std;

class Base {

public:

int publicVar;

void publicFunction() {

cout << "Base::publicFunction() called" << endl;

}

protected:

int protectedVar;

void protectedFunction() {

cout << "Base::protectedFunction() called" << endl;

}

};

class Derived : public Base {

public:

void accessBaseMembers() {

cout << "Derived class accessing Base class members:" << endl;

cout << "publicVar: " << publicVar << endl;

cout << "protectedVar: " << protectedVar << endl;

publicFunction();

protectedFunction();

}

};

int main() {

Derived derivedObj;

derivedObj.accessBaseMembers();

cout << "Accessing Base class members from the Main function:" << endl;

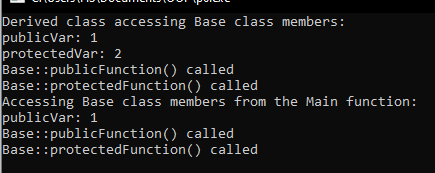
cout << "publicVar: " << derivedObj.publicVar << endl;

derivedObj.publicFunction();

return 0;

}

**Output**

****

## Constructor

In object-oriented programming (OOP), a constructor is a special method or function within a class that is automatically called when an object of the class is created. Constructors are used to initialize the object's state or set its initial properties.

**Types of constructors:**

Three types of constructors: **default** (no parameters), **parameterized** (with parameters) and **Copy Constructor**.

## Destructor

Destructor is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime. It has same name as the class. It is preceded by a tilde ( ~ ). Destructor does not take arguments, it can never be overloaded. A class can have no more than one destructor. The destructor cannot have any parameters. The class destructor is automatically invoked when the instance of that class goes out of scope.

## Program

**Create a base class Vehicle with attributes like speed and a derived class Car that inherits from Vehicle. Implement constructors for both classes and display the speed of the car.**

#include <iostream>

using namespace std;

class Vehicle {

protected:

int speed;

public:

Vehicle(int s) : speed(s) {}

};

class Car : public Vehicle {

public:

Car(int s) : Vehicle(s) {}

void displaySpeed() {

cout << "Car's speed: " << speed << " mph" <<endl;

}

};

int main() {

Car car(60);

car.displaySpeed();

return 0;

}

**Output**

****

## Inheritance:

Inheritance is a fundamental concept in object-oriented programming that allows a new class (derived or child class) to inherit properties and behaviours from an existing class (base or parent class). This enables code reusability and the creation of hierarchical relationships among classes.

**Types of inheritance:**

**Single Inheritance:**

Single inheritance involves a derived class inheriting from a single base class.

**Example:**

Dog class inherit from animal class.

**Multiple Inheritance:**

Multiple inheritance allows a derived class to inherit from multiple base classes.

**Example:**

A class inherit from two other classes.

**Multilevel Inheritance:**

Multilevel inheritance involves a chain of inheritance where a derived class becomes the base class for another class.

**Example:**

Parent class inherit from grand-parent class and child class from parent class.

**Hierarchical Inheritance:**

Hierarchical inheritance occurs when multiple classes inherit from a single base class.

**Hybrid Inheritance:**

Hybrid inheritance is a combination of different types of inheritance within a single program.

## Program # 01

**Write a program to inherit a class from another class.**

#include<iostream>

using namespace std;

class Person{

public:

char name[20];

int age;

void Eating();

void in(){

cout<<"Enter the name : "<<endl;

cin>>name;

cout<<"Enter the age : "<<endl;

cin>>age;

}

void out(){

cout<<"the name of person is : "<<name<<endl;

cout<<"the age of person is : "<<age<<endl;

}

};

class Student: public Person{

public:

int student\_id = 20;

char grade = 'A';

void Study(){

cout<<"the student id is : "<<student\_id<<endl;

cout<<"his grades are : "<<grade<<endl;

}

};

main(){

Student e;

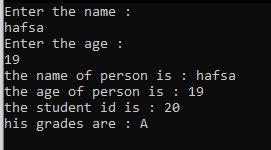
e.in();

e.out();

e.Study();

}

**Output**



## Program #2

**Write a program to inherit two classes manager and engineer from an employee base class**.

#include<iostream>

using namespace std;

class Employee{

public:

int id;

int salary;

void in(){

cout<<"the employee id is: "<<endl;

cin>>id;

cout<<"the salary of employee is"<<endl;

cin>>salary;

}

};

class Manager:public Employee{

int id;

int salary;

string depart="HEC management";

public:

void management(){

cout<<"the manager manages the office work he works in "<<depart<<endl;

}

};

class Engineer:public Employee{

string site="office";

public:

void visitsite(){

cout<<"the engineer visit the "<<site<<" site";

}

};

main(){

Manager m;

m.in();

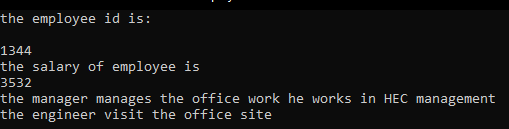
m.management();

Engineer n;

n.visitsite();

}

**Output**

****

## Program #3

**Define a base class Shape with members to store the dimensions of a shape and a derived class Circle that inherits from Shape. Add a method to calculate the area of the circle.**

#include <iostream>

#include <cmath>

using namespace std;

class Shape {

public:

double calculateArea() {

return 0.0;

}

};

class Circle : public Shape {

public:

double radius;

double calculateArea() {

return M\_PI \* radius \* radius;

}

};

int main() {

Circle circle;

circle.radius = 5.0;

double area = circle.calculateArea();

cout << "Area of the circle is: " << area << endl;

return 0;

}

**Output**

****

## Program #4

**Create a base class Person with attributes name and age. Create a derived class Student which inherits from Person and has additional attributes studentId and grade. Implement appropriate methods.**

#include <iostream>

#include <string>

using namespace std;

class Person {

public:

string name;

int age;

void displayInfo() {

cout << "Name: " << name << ", Age: " << age;

}

};

class Student : public Person {

public:

string studentId;

double grade;

void displayStudentInfo() {

displayInfo();

cout << ", Student ID: " << studentId << ", Grade: " << grade << endl;

}

};

int main() {

Student student;

student.name = "Hafsa";

student.age = 19;

student.studentId = "12345";

student.grade = 95.5;

cout << "Student Information: ";

student.displayStudentInfo();

return 0;

}

**Output **

## Polymorphism

Polymorphism in object-oriented programming (OOP) is a concept that allows objects of different classes to be treated as objects of a common superclass. It enables you to write code that can work with objects in a more general way, without needing to know their specific types. Polymorphism is typically achieved through method overriding and overloading or abstract classes, making it easier to create flexible and extensible code.

**Types of Polymorphism:**

**1.** **Compile-time Polymorphism (Static Binding or Early Binding):**

**Function Overloading:**

When multiple functions in the same scope have the same name but different parameters.

**Operator Overloading:**

Extending the functionality of operators for user-defined data types.

**2. Run-time Polymorphism (Dynamic Binding or Late Binding):**

**Method Overriding:**

When a subclass provides a specific implementation of a method that is already defined in its superclass.

**Virtual Function:**

A virtual function is a function declared in a base class and marked as "virtual" that can be **overridden** by derived classes. When objects of different derived classes are treated as objects of the base class, calling a virtual function on these objects invokes the specific implementation from the derived class. This enables different behaviours based on the actual object's type, making it a key feature of polymorphism.

**Pure Virtual Function:**

A pure virtual function in object-oriented programming (OOP) is a function declared in an abstract base class with no implementation. It's used to define a common interface that derived classes must implement, ensuring they provide their own specific implementations. This concept enables polymorphism and is particularly useful for designing frameworks and interfaces in OOP.

## Program #1

**Writ a program with Base class Shape method (draw function overloading ) 2 function and Drive class circle (2funt) or triangle(2 function), Any one function is overload. Displays function**

#include<iostream>

using namespace std;

class shape{

public:

void draw(int x){

cout<<"draw any shape."<<endl;

}

void draw(double x){

cout<<"draw a circle shape."<<endl;

}

void draw(int x,int y){

cout<<"draw a triangle shape."<<endl;

}

void setColor(){

cout<<"choose any color"<<endl;

}

};

class circle: public shape{

public:

void radius(int x){

cout<<"the radius of a cicle is"<<x<<endl;

}

void radius(double x){

cout<<"the radius of cicle is d/2"<<x<<endl;

}

void radius(int x,int y){

cout<<"function overloaded values are"<<x<<" "<<y<<endl;

}

void diameter(){

cout<<"find the diameter of a cicle"<<endl;

}

};

class triangle: public shape{

public:

void height(int x){

cout<<"height of triangle is "<<x<<endl;

}

void height(double x){

cout<<"height of triangle is "<<x<<endl;

}

void height(int x,int y){

cout<<"height of triangle is x,y as "<<x<<" "<<y<<endl;

}

void base(){

cout<<"triangle has a base"<<endl;

}

};

main(){

circle c;

c.radius(23);

c.radius(54,47);

c.diameter();

triangle t;

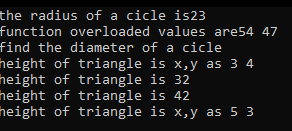
t.height(3,4);

t.height(32);

return 0;

}

**Output:**

****

## Program#2

**Inherit a function using virtual class function**

#include<iostream>

using namespace std;

class base{

public:

virtual void print(){

cout<<"print the base class"<<endl;

}

void show(){

cout<<"show the base class"<<endl;

}

};

class derived : public base{

public:

void print(){

cout<<"print the derived class"<<endl;

}

void show(){

cout<<"show the derived class"<<endl;

}

};

main(){

base \*pt;

derived d;

pt = &d;

//virtual function, runtime

pt->print();

//non virtual function, compile time

pt->show();

}

**Output**



## Program #3

**Define a base class Shape with a virtual function calculateArea(). Derive classes Circle and Rectangle from Shape and override calculateArea() for each shape.**

#include <iostream>

#include <cmath>

using namespace std;

class Shape {

public:

virtual double calculateArea() {

return 0.0;

}

};

class Circle : public Shape {

private:

double radius;

public:

void setRadius(double r) {

radius = r;

}

double calculateArea(){

return 3.14 \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

void setDimensions(double w, double h) {

width = w;

height = h;

}

double calculateArea(){

return width \* height;

}

};

int main() {

Circle circle;

circle.setRadius(5.0);

Rectangle rectangle;

rectangle.setDimensions(4.0, 6.0);

cout << "Area of the circle: " << circle.calculateArea() << endl;

cout << "Area of the rectangle: " << rectangle.calculateArea() << endl;

return 0;

}

**Output**

****

## Program #4

**Create an abstract base class BankAccount with a pure virtual function calculateInterest(). Derive two classes SavingsAccount and CurrentAccount from BankAccount and implement calculateInterest() accordingly.**

#include <iostream>

using namespace std;

class BankAccount {

public:

virtual void calculateInterest() = 0; // Pure virtual function

};

class SavingsAccount : public BankAccount {

public:

void calculateInterest(){

cout << "Interest calculated for Savings Account." <<endl;

}

};

class CurrentAccount : public BankAccount {

public:

void calculateInterest(){

cout << "Interest calculated for Current Account." <<endl;

}

};

int main() {

BankAccount\* account1 = new SavingsAccount();

BankAccount\* account2 = new CurrentAccount();

account1->calculateInterest();

account2->calculateInterest();

return 0;

}

**Output**

****

## Program #5

**Create a base class Animal with a method makeSound(). Derive two classes Dog and Cat from Animal and override the makeSound() method to represent the sound each animal makes**

#include <iostream>

using namespace std;

class Animal {

public:

void makeSound() {

cout << "Animal makes a generic sound" << endl;

}

};

class Dog : public Animal {

public:

void makeSound() {

cout << "Dog barks" << endl;

}

};

class Cat : public Animal {

public:

void makeSound() {

cout << "Cat meows" << endl;

}

};

int main() {

Animal animal1;

Dog dog;

Cat cat;

animal1.makeSound();

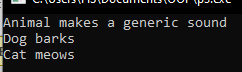
dog.makeSound();

cat.makeSound();

return 0;

}

**Output**



## Exception

Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. C++ provides following specialized keywords for this purpose.

• **try:** represents a block of code that can throw an exception.

• **catch**: represents a block of code that is executed when a particular exception is thrown.

• **throw:** Used to throw an exception. Also used to list the exceptions that a function throws, but doesn’t handle itself.

## Program

**Write a program to calculate the square of a number using exception.**

#include<iostream>

#include<cmath>

using namespace std;

int main(){

try{

int a;

cout<<"enter a num : "<<endl;

cin>>a;

if(a<0){

cout<<"negative num"<<endl;

throw"negative num";

}

if(a==0){

cout<<"invalid"<<endl;

throw a;

}

int sq = sqrt(a);

cout<<a<<" is square of "<<sq<<endl;

}

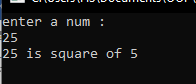
catch(...){

cout<<"error!!"<<endl;

}

}

**Output:**

****

## Abstract Class

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in class.

**Syntax**

class AB {

public:

virtual void f() = 0;

};

## Concreate class

A concrete class is an ordinary class which has no purely virtual functions and hence can be instantiated. It has an implementation for all of its methods. There cannot have any unimplemented methods.

**Example**

#include <iostream>

using namespace std;

class Shape {

public:

**virtual void draw()= 0;**

void printDes (){

cout << "This is a shape." << endl;

}};

class Circle : public Shape {

public:

void draw(){

cout << "Drawing a circle." << endl;

}};

class Square : public Shape {

public:

void draw(){

cout << "Drawing a square." << endl;

}};

int main() {

Circle circle;

Square square;

circle.draw();

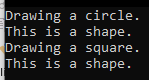
circle.printDes ();

square.draw();

square.printDes ();

return 0;

} **output**

****

## Friend function

A friend function in C++ is a function that is not a member of a class but has access to the class's private and protected members. It is declared inside the class with the keyword **"friend"** preceding the function prototype.

**Purpose of using friend functions:**

The purpose of using friend functions in a C++ class is to allow specific non-member functions to access the private and protected members of the class without having to make those members public. This can be useful in situations where you want to provide access to certain functions without exposing the internal details of the class to the outside world. It can also be used to achieve operator overloading, where non-member functions are given access to private members of a class to perform operations.

**Example**

#include<iostream>

using namespace std;

class B;

class A{

int a;

public:

A(){

a= 10;

}

friend void show(A,B);//function declared

};

class B{

int b;

public:

B(){

b = 12;

}

friend void show(A,B);//function declared

};

void show(A x, B y){//function define

int r;

r= x.a + y.b;

cout<<"value of class A : "<<x.a<<endl;

cout<<"value of class B : "<<y.b<<endl;

cout<<"sum of class A+B : "<<r<<endl;

}

int main(){

A o1;

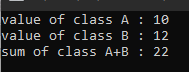
B o2;

show(o1,o2);

return 0;

}

**Output**

****

## File Handling

In C++, file handling is implemented through the use of the `<fstream>` library, which provides classes and functions for working with files. The key components of file handling in C++ include the following:

**1.** **File Streams:** C++ provides three classes for working with files: `ifstream` (input file stream) for reading from files, `ofstream` (output file stream) for writing to files, and `fstream` (file stream) for both reading and writing.

**2.** **File Opening:** Files are opened using the `open()` method of file stream objects, specifying the file name and the mode (e.g., input, output, append) as parameters.

**3.** **File Reading and Writing**: Once a file is opened, data can be read from or written to the file using standard input/output operations such as `<<` and `>>` for formatted input/output, and `write()` and `read()` for binary input/output.

**1. ofstream (Output File Stream):** This class is used to create and write to files. It is used when you want to write data to a file.

**Example:**

#include <iostream>

#include <fstream>

using namespace std;

int main() {

ofstream outputFile("output.txt"); // Create an output file stream

if (outputFile.is\_open()) {

outputFile << "Hello, this is a sample output to a file." << endl;

outputFile.close(); // Close the file

cout << "Data has been written to the file." << endl;

} else {

cout << "Unable to open the file." << endl;

}

return 0;

} **Output**



**2. ifstream (Input File Stream):** This class is used to read from files. It is used when you want to read data from a file.

**Example:**

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main() {

ifstream inputFile("input.txt"); // Create an input file stream

string line;

if (inputFile.is\_open()) {

while (getline(inputFile, line)) {

cout << line << endl; // Output the content of the file

}

inputFile.close(); // Close the file

} else {

cout << "Unable to open the file." << endl;

}

return 0;

} **Output**



**3. fstream (File Stream):** This class is used for both reading from and writing to files. It provides the functionality of both `ofstream` and `ifstream`.

**Example:**

#include <iostream>

#include <fstream>

using namespace std;

int main() {

fstream file("data.txt", ios::out | ios::in); // Create a file stream for both input and output

if (file.is\_open()) {

file << "This is a sample text." << endl; // Write to the file

file.seekg(0); // Move the file pointer to the beginning

string content;

getline(file, content); // Read from the file

cout << "Content of the file: " << content << endl;

file.close(); // Close the file

} else {

cout << "Unable to open the file." << endl;

}

****return 0;

} **Output**