 Create an abstract base class Account that has a **pure virtual function** display\_balance(). The class should have a protected member variable to store the account balance.

 Implement two derived classes:

* SavingsAccount: This class should override the display\_balance() function to display the balance with a bonus amount added.
* CurrentAccount: This class should override the display\_balance() function to display the balance after deducting a maintenance fee.

 Use pointers of the base class Account to demonstrate **runtime polymorphism** and call the appropriate display\_balance() function for each account type.

#include <iostream>

using namespace std;

// Base class

class Account {

protected:

double balance;

public:

void bal() { // Default constructor to initialize balance

balance = 1000; // Default balance

}

// Pure virtual function making Account an abstract class

virtual void display\_balance() = 0;

};

// Derived class for Savings Account

class SavingsAccount : public Account {

private:

double bonus = 200.0; // Bonus for savings account

public:

void display\_balance() {

bal();// Override pure virtual function

cout << "Savings Account Balance: " << (balance + bonus) << endl;

}

};

// Derived class for Current Account

class CurrentAccount : public Account {

private:

double maintenance\_fee = 50.0; // Maintenance fee for current account

public:

void display\_balance() {

bal();// Override pure virtual function

cout << "Current Account Balance: " << (balance - maintenance\_fee) << endl;

}

};

int main() {

Account \*accountPtr; // Pointer of type Account

// Create a SavingsAccount object and use pointer to call display\_balance

SavingsAccount savings;

accountPtr = &savings;

accountPtr->display\_balance(); // Calls SavingsAccount's display\_balance()

// Create a CurrentAccount object and use pointer to call display\_balance

CurrentAccount current;

accountPtr = &current;

accountPtr->display\_balance(); // Calls CurrentAccount's display\_balance()

return 0;

}

Create a base class named **Shape** that includes a pure virtual function called **calculateArea()**. This function will be responsible for calculating the area of the shapes.

You will derive two classes from the **Shape** class:

1. **Square**: Implement the **calculateArea()** function to compute the area of a square using the formula:

Area=dimension\*dimension

**Circle**: Implement the **calculateArea()** function to compute the area of a circle using the formula:

Area=π\*dimension\*dimension

#include<iostream>

using namespace std;

class Shape

{

protected:

float dimension;

public:

void getDimension()

{

cin >> dimension;

}

// pure virtual Function

virtual float calculateArea()=0;

};

class Square : public Shape

{

public:

float calculateArea()

{

return dimension \* dimension;

}

};

class Circle : public Shape

{

public:

float calculateArea()

{

return 3.14\*dimension\*dimension;

}

};

int main()

{

Shape \*s1;

Square s;

Circle c;

cout<<"enter dimension";

s.getDimension();

s1=&s;

cout<<"area of square is"<<s1->calculateArea();

s1=&c;

cout<<"area of cirle";

cout<<s1->calculateArea()<<endl;

return 0;

}

Write a C++ program that demonstrates **function overloading** by creating a class named Adder. The class should have the following overloaded functions named add:

1. **add(int a, int b)**: This function takes two integers as parameters and returns their sum.
2. **add(float a, float b)**: This function takes two floating-point numbers as parameters and returns their sum.
3. **add(int a, int b, int c)**: This function takes three integers as parameters and returns their sum.

In the main() function, create an instance of the Adder class and call each of the overloaded add() functions to demonstrate the concept of function overloading.

#include <iostream>

using namespace std;

class adder {

public:

// Function to add and display the sum of two integers

void add(int x, int y) {

cout << "Sum of x and y is " << x + y << endl;

}

// Function to add and display the sum of two doubles

void add(double x, double y) {

cout << "Sum of x and y is " << x + y << endl;

}

void add(int x, int y, int z) {

cout << "Sum of x and y is " << x + y + z << endl;

}

};

int main() {

adder obj1;

obj1.add(7,7);

// func() is called with double value

obj1.add(9.132,10.890);

// func() is called with 3 int values and adds them

obj1.add(10,10,10);

return 0;

}

* Implement a program to handle exception it prints appropriate error messages, should anything go wrong. The expected behavior is defined as follows: If the compute function runs fine with the given arguments, then print the result of the function call. If it fails to allocate the memory that it needs, print Not enough memory. If any other standard C++ exception occurs, print Exception: S where S is the exception's error message. If any non-standard exception occurs, print Other Exception
* #include <iostream>
* #include <stdexcept> // For standard exceptions like runtime\_error
* using namespace std;
* // Simple function that might throw exceptions
* int compute(int a, int b) {
* if (a == 0) {
* throw bad\_alloc(); // Simulate memory allocation failure
* }
* if (b == 0) {
* throw runtime\_error("Division by zero"); // Simulate a standard exception
* }
* if (a == -1) {
* throw "Non-standard exception"; // Simulate a non-standard exception
* }
* return a / b;
* }
* int main() {
* int a, b;
* cout << "Enter two integers: ";
* cin >> a >> b;
* try {
* int result = compute(a, b);
* cout << "Result: " << result << endl;
* }
* catch (const bad\_alloc&) {
* cout << "Not enough memory" << endl; // Handle memory allocation failure
* }
* catch (const runtime\_error& e) {
* cout << "Exception: " << e.what() << endl; // Handle standard exceptions with error message
* }
* catch (...) {
* cout << "Other Exception" << endl; // Catch any other non-standard exception
* }
* return 0;
* }
* Write a C++ Program for exception handling if age is greater than or equal to 18 then it display message access granted you are old enough otherwise throw an exception.
* #include<iostream>  
  using namespace std;
* Int main() {
* try {  
    int age = 15;  
    if (age >= 18) {  
      cout << "Access granted - you are old enough.";  
    } else {  
      throw (age);  
    } }
* //variable myNum holds the value of the thrown exception, which is age   
  catch (int myNum) {  
    cout << "Access denied - You must be at least 18 years old.\n";  
    cout << "Age is: " << myNum;  
  } return 0; }
* Explain file handling with example
* Analyze the result of the following code we are executing program in text editor?

#include <iostream>

#include <fstream>

using namespace std;

int main () {

ofstream filestream("testout.txt");

 if (filestream.is\_open())

{

filestream << "Programming in C++\n";

filestream << "Object Oriented Programming\n";

filestream.close();

}

else

cout <<"File opening is fail.";

return 0;

}

* Explain Inheritance with its types
* Develop a C++ program to create a base class Vehicle with attributes like manufacturer and model. Create two derived classes: Car and Bike. Each derived class should have its specific attribute like numberOfDoors for Car and engineCapacity for Bike.
* Explain polymorphism with its types.
* Explain Compile time and run time polymorphism with example.
* Explain Function overloading with example
* Explain virtual function with example
* Explain abstract class with example