**Assignment Number: 6**

**Solution 1:**

#include <iostream>

using namespace std;

class Rectangle {

private:

double length;

double width;

public:

Rectangle(double l, double w) {

length = l;

width = w;

}

double calculateArea() {

return length \* width;

}

double calculatePerimeter() {

return 2 \* (length + width);

}

};

int main() {

Rectangle rect(5, 3);

cout << "Area: " << rect.calculateArea() << endl;

cout << "Perimeter: " << rect.calculatePerimeter() << endl;

return 0;

}

**Solution 2:**

#include <iostream>

#include <string>

using namespace std;

class Student {

private:

string name;

int rollNumber;

double marks[3];

public:

void inputDetails() {

cout << "Enter student name: ";

getline(cin, name);

cout << "Enter roll number: ";

cin >> rollNumber;

cout << "Enter marks in three subjects: ";

for (int i = 0; i < 3; ++i) {

cin >> marks[i];

}

}

void displayDetails() {

cout << "Student Name: " << name << endl;

cout << "Roll Number: " << rollNumber << endl;

cout << "Marks in three subjects: ";

for (int i = 0; i < 3; ++i) {

cout << marks[i] << " ";

}

cout << endl;

}

double calculateAverageMarks() {

double sum = 0;

for (int i = 0; i < 3; ++i) {

sum += marks[i];

}

return sum / 3.0;

}

};

int main() {

Student student;

student.inputDetails();

student.displayDetails();

cout << "Average Marks: " << student.calculateAverageMarks() << endl;

return 0;

}

**Solution 3:**

#include <iostream>

#include <string>

using namespace std;

class BankAccount {

private:

string accountNumber;

string accountHolderName;

double balance;

public:

BankAccount(string accNum, string accHolder, double initialBalance) {

accountNumber = accNum;

accountHolderName = accHolder;

balance = initialBalance;

}

void deposit(double amount) {

balance += amount;

cout << "Deposit of $" << amount << " successful." << endl;

}

void withdraw(double amount) {

if (amount > balance) {

cout << "Insufficient funds. Withdrawal failed." << endl;

} else {

balance -= amount;

cout << "Withdrawal of $" << amount << " successful." << endl;

}

}

void displayAccountDetails() {

cout << "Account Number: " << accountNumber << endl;

cout << "Account Holder Name: " << accountHolderName << endl;

cout << "Balance: $" << balance << endl;

}

};

int main() {

BankAccount account("123456789", "abc", 1000.0);

cout << "Initial Account Details:" << endl;

account.displayAccountDetails();

cout << endl;

account.deposit(500.0);

account.withdraw(200.0);

cout << "\nUpdated Account Details:" << endl;

account.displayAccountDetails();

return 0;

}

**Solution 4:**

#include <iostream>

#include <string>

using namespace std;

class Car {

private:

string make;

string model;

int year;

public:

Car(string carMake, string carModel, int carYear) {

make = carMake;

model = carModel;

year = carYear;

}

void setMake(string carMake) {

make = carMake;

}

void setModel(string carModel) {

model = carModel;

}

void setYear(int carYear) {

year = carYear;

}

string getMake() {

return make;

}

string getModel() {

return model;

}

int getYear() {

return year;

}

};

int main() {

Car myCar("Toyota", "Corolla", 2020);

cout << "Initial Car Details:" << endl;

cout << "Make: " << myCar.getMake() << endl;

cout << "Model: " << myCar.getModel() << endl;

cout << "Year: " << myCar.getYear() << endl;

myCar.setMake("Honda");

myCar.setModel("Civic");

myCar.setYear(2019);

cout << "\nUpdated Car Details:" << endl;

cout << "Make: " << myCar.getMake() << endl;

cout << "Model: " << myCar.getModel() << endl;

cout << "Year: " << myCar.getYear() << endl;

return 0;

}

**Solution 5:**

#include <iostream>

using namespace std;

class Time {

private:

int hours;

int minutes;

int seconds;

public:

Time(int h = 0, int m = 0, int s = 0) {

hours = h;

minutes = m;

seconds = s;

}

Time addTime(Time t) {

Time sum;

sum.seconds = seconds + t.seconds;

sum.minutes = minutes + t.minutes + sum.seconds / 60;

sum.hours = hours + t.hours + sum.minutes / 60;

sum.minutes %= 60;

sum.seconds %= 60;

return sum;

}

void displayTime() {

cout << "Time: " << hours << " hours, " << minutes << " minutes, " << seconds << " seconds" << endl;

}

};

int main() {

Time time1(3, 45, 30);

Time time2(1, 20, 45);

cout << "Time 1: ";

time1.displayTime();

cout << "Time 2: ";

time2.displayTime();

Time sum = time1.addTime(time2);

cout << "\nSum of Time 1 and Time 2: ";

sum.displayTime();

return 0;

}

**Solution 6:**

#include <iostream>

#include <string>

using namespace std;

class Employee {

private:

int employeeID;

string name;

double salary;

public:

Employee(int id, string empName, double empSalary) {

employeeID = id;

name = empName;

salary = empSalary;

}

void giveRaise(double raiseAmount) {

salary += raiseAmount;

cout << name << "'s salary has been raised by $" << raiseAmount << endl;

}

void displayDetails() {

cout << "Employee ID: " << employeeID << endl;

cout << "Name: " << name << endl;

cout << "Salary: $" << salary << endl;

}

};

int main() {

Employee emp1(101, "AC", 50000);

cout << "Initial Details:" << endl;

emp1.displayDetails();

emp1.giveRaise(5000);

cout << "\nDetails after Raise:" << endl;

emp1.displayDetails();

return 0;

}

**Solution 7:**

#include <iostream>

using namespace std;

class Circle {

private:

double radius;

public:

Circle(double r) {

radius = r;

}

double calculateArea() {

return 3.14 \* radius \* radius;

}

double calculateCircumference() {

return 2 \* 3.14 \* radius;

}

};

int main() {

Circle circle(5);

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

cout << "Circumference of the circle: " << circle.calculateCircumference() << endl;

return 0;

}

**Solution 8:**

#include <iostream>

using namespace std;

class Complex {

private:

double real;

double imag;

public:

Complex(double r, double i) {

real = r;

imag = i;

}

Complex add(Complex c) {

double newReal = real + c.real;

double newImag = imag + c.imag;

return Complex(newReal, newImag);

}

Complex subtract(Complex c) {

double newReal = real - c.real;

double newImag = imag - c.imag;

return Complex(newReal, newImag);

}

Complex multiply(Complex c) {

double newReal = (real \* c.real) - (imag \* c.imag);

double newImag = (real \* c.imag) + (imag \* c.real);

return Complex(newReal, newImag);

}

Complex divide(Complex c) {

double denominator = (c.real \* c.real) + (c.imag \* c.imag);

double newReal = ((real \* c.real) + (imag \* c.imag)) / denominator;

double newImag = ((imag \* c.real) - (real \* c.imag)) / denominator;

return Complex(newReal, newImag);

}

void display() {

cout << real << " + " << imag << "i" << endl;

}

};

int main() {

Complex c1(3, 4);

Complex c2(1, 2);

Complex sum = c1.add(c2);

Complex difference = c1.subtract(c2);

Complex product = c1.multiply(c2);

Complex quotient = c1.divide(c2);

cout << "Sum: ";

sum.display();

cout << "Difference: ";

difference.display();

cout << "Product: ";

product.display();

cout << "Quotient: ";

quotient.display();

return 0;

}

**Solution 9:**

#include <iostream>

#include <string>

using namespace std;

class Book {

private:

string title;

string author;

int publicationYear;

public:

Book(string t, string a, int year) {

title = t;

author = a;

publicationYear = year;

}

int comparePublicationYear(Book b) {

if (publicationYear < b.publicationYear) {

return -1;

} else if (publicationYear > b.publicationYear) {

return 1;

} else {

return 0;

}

}

void display() {

cout << "Title: " << title << endl;

cout << "Author: " << author << endl;

cout << "Publication Year: " << publicationYear << endl;

}

};

int main() {

Book book1("The Great Gatsby", "F. Scott Fitzgerald", 1925);

Book book2("To Kill a Mockingbird", "Harper Lee", 1960);

int result = book1.comparePublicationYear(book2);

cout << "Book 1:" << endl;

book1.display();

cout << endl;

cout << "Book 2:" << endl;

book2.display();

cout << endl;

if (result < 0) {

cout << "Book 1 was published before Book 2." << endl;

} else if (result > 0) {

cout << "Book 1 was published after Book 2." << endl;

} else {

cout << "Both books were published in the same year." << endl;

}

return 0;

}

**Solution 10:**

#include <iostream>

#include <vector>

using namespace std;

class Matrix {

private:

vector<vector<int>> mat;

int rows;

int cols;

public:

Matrix(int m, int n) : rows(m), cols(n) {

mat.resize(rows, vector<int>(cols, 0));

}

void setValue(int i, int j, int value) {

if (i >= 0 && i < rows && j >= 0 && j < cols) {

mat[i][j] = value;

} else {

cout << "Invalid index!" << endl;

}

}

int getValue(int i, int j) {

if (i >= 0 && i < rows && j >= 0 && j < cols) {

return mat[i][j];

} else {

cout << "Invalid index!" << endl;

return -1; // Assuming -1 represents an invalid value

}

}

Matrix add(Matrix& other) {

if (rows != other.rows || cols != other.cols) {

cout << "Matrices cannot be added. Dimensions mismatch." << endl;

return Matrix(0, 0); // Return empty matrix

}

Matrix result(rows, cols);

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

result.mat[i][j] = mat[i][j] + other.mat[i][j];

}

}

return result;

}

Matrix subtract(Matrix& other) {

if (rows != other.rows || cols != other.cols) {

cout << "Matrices cannot be subtracted. Dimensions mismatch." << endl;

return Matrix(0, 0); // Return empty matrix

}

Matrix result(rows, cols);

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

result.mat[i][j] = mat[i][j] - other.mat[i][j];

}

}

return result;

}

Matrix multiply(Matrix& other) {

if (cols != other.rows) {

cout << "Matrices cannot be multiplied. Inner dimensions mismatch." << endl;

return Matrix(0, 0);

}

Matrix result(rows, other.cols);

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < other.cols; ++j) {

for (int k = 0; k < cols; ++k) {

result.mat[i][j] += mat[i][k] \* other.mat[k][j];

}

}

}

return result;

}

void display() {

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

cout << mat[i][j] << " ";

}

cout << endl;

}

}

};

int main() {

Matrix A(2, 3);

Matrix B(2, 3);

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

A.setValue(i, j, i + j);

B.setValue(i, j, i - j);

}

}

cout << "Matrix A:" << endl;

A.display();

cout << endl;

cout << "Matrix B:" << endl;

B.display();

cout << endl;

cout << "Sum of matrices A and B:" << endl;

Matrix sum = A.add(B);

sum.display();

cout << endl;

cout << "Difference of matrix B from A:" << endl;

Matrix diff = A.subtract(B);

diff.display();

cout << endl;

cout << "Product of matrices A and B:" << endl;

Matrix product = A.multiply(B);

product.display();

cout << endl;

return 0;

}

**Solution 11:**

#include <iostream>

using namespace std;

class Date {

private:

int day;

int month;

int year;

public:

Date(int d, int m, int y) : day(d), month(m), year(y) {}

int compare(const Date& other) const {

if (year != other.year) {

return year - other.year;

}

if (month != other.month) {

return month - other.month;

}

return day - other.day;

}

int difference(const Date& other) const {

const int days\_per\_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int days = 0;

if (year == other.year) {

if (month == other.month) {

days += abs(day - other.day);

} else {

days += days\_per\_month[month - 1] - day;

for (int i = month; i < other.month - 1; ++i) {

days += days\_per\_month[i];

}

days += other.day;

}

} else {

int start\_month = month;

int end\_month = other.month;

days += days\_per\_month[start\_month - 1] - day;

for (int i = start\_month; i < 12; ++i) {

days += days\_per\_month[i];

}

for (int i = year + 1; i < other.year; ++i) {

if ((i % 4 == 0 && i % 100 != 0) || (i % 400 == 0)) {

days += 366; // Leap year

} else {

days += 365; // Non-leap year

}

}

for (int i = 0; i < end\_month - 1; ++i) {

days += days\_per\_month[i];

}

days += other.day;

}

return days;

}

};

int main() {

Date date1(1, 1, 2022);

Date date2(1, 1, 2023);

int cmp = date1.compare(date2);

if (cmp < 0) {

cout << "Date 1 is before Date 2" << endl;

} else if (cmp > 0) {

cout << "Date 1 is after Date 2" << endl;

} else {

cout << "Date 1 is equal to Date 2" << endl;

}

int diff = date1.difference(date2);

cout << "Difference in days: " << diff << endl;

return 0;

}

**Solution 12:**

#include <iostream>

#include <cmath>

using namespace std;

class Vector {

private:

double x;

double y;

double z;

public:

Vector(double x\_val, double y\_val, double z\_val) : x(x\_val), y(y\_val), z(z\_val) {}

double dotProduct(const Vector& other) const {

return x \* other.x + y \* other.y + z \* other.z;

}

Vector crossProduct(const Vector& other) const {

double cross\_x = y \* other.z - z \* other.y;

double cross\_y = z \* other.x - x \* other.z;

double cross\_z = x \* other.y - y \* other.x;

return Vector(cross\_x, cross\_y, cross\_z);

}

void display() const {

cout << "(" << x << ", " << y << ", " << z << ")" << endl;

}

};

int main() {

Vector v1(1.0, 2.0, 3.0);

Vector v2(4.0, 5.0, 6.0);

cout << "Vector 1: ";

v1.display();

cout << "Vector 2: ";

v2.display();

double dot\_product = v1.dotProduct(v2);

cout << "Dot Product: " << dot\_product << endl;

Vector cross\_product = v1.crossProduct(v2);

cout << "Cross Product: ";

cross\_product.display();

return 0;

}

**Solution 13:**

#include <iostream>

#include <vector>

#include <stdexcept>

using namespace std;

class Account {

private:

string accountNumber;

double balance;

public:

Account(const string& accNumber, double initialBalance) : accountNumber(accNumber), balance(initialBalance) {}

void deposit(double amount) {

if (amount < 0) {

throw invalid\_argument("Deposit amount must be positive.");

}

balance += amount;

}

void withdraw(double amount) {

if (amount < 0) {

throw invalid\_argument("Withdrawal amount must be positive.");

}

if (amount > balance) {

throw runtime\_error("Insufficient funds.");

}

balance -= amount;

}

double getBalance() const {

return balance;

}

};

class Bank {

private:

vector<Account> accounts;

public:

void addAccount(const string& accNumber, double initialBalance) {

accounts.push\_back(Account(accNumber, initialBalance));

}

void deposit(const string& accNumber, double amount) {

for (auto& account : accounts) {

if (accountNumber == accNumber) {

account.deposit(amount);

return;

}

}

throw invalid\_argument("Account not found.");

}

void withdraw(const string& accNumber, double amount) {

for (auto& account : accounts) {

if (accountNumber == accNumber) {

account.withdraw(amount);

return;

}

}

throw invalid\_argument("Account not found.");

}

double getBalance(const string& accNumber) const {

for (const auto& account : accounts) {

if (accountNumber == accNumber) {

return account.getBalance();

}

}

throw invalid\_argument("Account not found.");

}

};

int main() {

Bank bank;

bank.addAccount("123456", 1000);

bank.addAccount("789012", 2000);

try {

bank.deposit("123456", 500);

bank.withdraw("789012", 300);

} catch (const exception& e) {

cerr << "Error: " << e.what() << endl;

}

try {

cout << "Balance of account 123456: " << bank.getBalance("123456") << endl;

cout << "Balance of account 789012: " << bank.getBalance("789012") << endl;

} catch (const exception& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Solution 14:**

#include <iostream>

using namespace std;

class Triangle {

private:

double side1, side2, side3;

public:

Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

bool isEquilateral() const {

return side1 == side2 && side2 == side3;

}

bool isIsosceles() const {

return side1 == side2 || side1 == side3 || side2 == side3;

}

bool isScalene() const {

return side1 != side2 && side1 != side3 && side2 != side3;

}

};

int main() {

double s1, s2, s3;

cout << "Enter the lengths of the sides of the triangle: ";

cin >> s1 >> s2 >> s3;

Triangle triangle(s1, s2, s3);

if (triangle.isEquilateral()) {

cout << "The triangle is equilateral." << endl;

} else if (triangle.isIsosceles()) {

cout << "The triangle is isosceles." << endl;

} else if (triangle.isScalene()) {

cout << "The triangle is scalene." << endl;

}

return 0;

}

**Solution 15:**

#include <iostream>

using namespace std;

const int MAX\_SIZE = 100;

class Stack {

private:

int data[MAX\_SIZE];

int top;

public:

Stack() : top(-1) {}

void push(int element) {

if (top == MAX\_SIZE - 1) {

cout << "Stack Overflow: Cannot push element onto full stack." << endl;

return;

}

data[++top] = element;

}

int pop() {

if (isEmpty()) {

cout << "Stack Underflow: Cannot pop element from empty stack." << endl;

return -1;

}

return data[top--];

}

bool isEmpty() const {

return top == -1;

}

};

int main() {

Stack stack;

stack.push(10);

stack.push(20);

stack.push(30);

while (!stack.isEmpty()) {

cout << "Popped: " << stack.pop() << endl;

}

cout << "Attempting to pop from an empty stack..." << endl;

cout << "Popped: " << stack.pop() << endl; // This should display an error message

return 0;

}

**Solution 16:**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

class BankAccount {

private:

string accountNumber;

string accountHolderName;

double balance;

public:

BankAccount(string accNum, string accHolder, double initialBalance)

: accountNumber(accNum), accountHolderName(accHolder), balance(initialBalance) {}

double getBalance() const {

return balance;

}

};

class BankManager {

private:

vector<BankAccount> accounts;

public:

void addAccount(const BankAccount& account) {

accounts.push\_back(account);

}

double getTotalBalance() const {

double totalBalance = 0;

for (const auto& acc : accounts) {

totalBalance += acc.getBalance();

}

return totalBalance;

}

BankAccount findAccountWithHighestBalance() const {

if (accounts.empty()) {

cerr << "No accounts available." << endl;

return BankAccount("", "", 0);

}

auto maxBalanceAccount = accounts[0];

for (size\_t i = 1; i < accounts.size(); ++i) {

if (accounts[i].getBalance() > maxBalanceAccount.getBalance()) {

maxBalanceAccount = accounts[i];

}

}

return maxBalanceAccount;

}

};

int main() {

BankManager bankManager;

bankManager.addAccount(BankAccount("123456", "Alice", 1000.0));

bankManager.addAccount(BankAccount("789012", "Bob", 1500.0));

bankManager.addAccount(BankAccount("345678", "Charlie", 2000.0));

cout << "Total balance of all accounts: $" << bankManager.getTotalBalance() << endl;

BankAccount highestBalanceAccount = bankManager.findAccountWithHighestBalance();

cout << "Account with the highest balance: " << highestBalanceAccount.getBalance() << endl;

return 0;

}

**Solution 17:**

#include <iostream>

#include <cmath>

#include <vector>

using namespace std;

struct Point {

double x, y;

Point(double \_x, double \_y) : x(\_x), y(\_y) {}

};

class Polygon {

private:

vector<Point> vertices;

public:

Polygon(const vector<Point>& \_vertices) : vertices(\_vertices) {}

double perimeter() const {

double totalPerimeter = 0.0;

size\_t numVertices = vertices.size();

for (size\_t i = 0; i < numVertices; ++i) {

size\_t nextIndex = (i + 1) % numVertices;

double dx = vertices[nextIndex].x - vertices[i].x;

double dy = vertices[nextIndex].y - vertices[i].y;

totalPerimeter += sqrt(dx \* dx + dy \* dy);

}

return totalPerimeter;

}

double area() const {

double totalArea = 0.0;

size\_t numVertices = vertices.size();

for (size\_t i = 0; i < numVertices; ++i) {

size\_t nextIndex = (i + 1) % numVertices;

totalArea += (vertices[i].x \* vertices[nextIndex].y) - (vertices[nextIndex].x \* vertices[i].y);

}

return abs(totalArea) / 2.0;

}

};

int main() {

vector<Point> vertices = {Point(0, 0), Point(4, 0), Point(4, 3), Point(0, 3)};

Polygon polygon(vertices);

cout << "Perimeter: " << polygon.perimeter() << endl;

cout << "Area: " << polygon.area() << endl;

return 0;

}

**Solution 18:**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100

class Queue {

private:

int arr[MAX\_SIZE];

int front;

int rear;

public:

Queue() {

front = -1;

rear = -1;

}

bool isEmpty() {

return front == -1 && rear == -1;

}

bool isFull() {

return (rear + 1) % MAX\_SIZE == front;

}

void enqueue(int value) {

if (isFull()) {

cout << "Queue is full. Cannot enqueue element." << endl;

return;

} else if (isEmpty()) {

front = rear = 0;

} else {

rear = (rear + 1) % MAX\_SIZE;

}

arr[rear] = value;

cout << "Enqueued element: " << value << endl;

}

void dequeue() {

if (isEmpty()) {

cout << "Queue is empty. Cannot dequeue element." << endl;

return;

} else if (front == rear) {

front = rear = -1;

} else {

front = (front + 1) % MAX\_SIZE;

}

cout << "Dequeued element." << endl;

}

};

int main() {

Queue queue;

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

queue.enqueue(4);

queue.enqueue(5);

queue.dequeue();

queue.dequeue();

queue.dequeue();

queue.dequeue();

queue.dequeue();

return 0;

}

**Solution 19:**

#include <iostream>

#include <cmath>

using namespace std;

class Point {

private:

double x;

double y;

public:

Point(double xCoord, double yCoord) : x(xCoord), y(yCoord) {}

double distanceTo(const Point& other) const {

double dx = x - other.x;

double dy = y - other.y;

return sqrt(dx \* dx + dy \* dy);

}

void display() const {

cout << "(" << x << ", " << y << ")";

}

};

int main() {

Point p1(1.0, 2.0);

Point p2(4.0, 6.0);

cout << "Distance between ";

p1.display();

cout << " and ";

p2.display();

cout << " is: " << p1.distanceTo(p2) << endl;

return 0;

}

**Solution 20:**

#include <iostream>

#include <unordered\_map>

#include <string>

using namespace std;

class Dictionary {

private:

unordered\_map<string, string> words;

public:

void addWord(const string& word, const string& meaning) {

words[word] = meaning;

}

string searchWord(const string& word) const {

auto it = words.find(word);

if (it != words.end()) {

return it->second;

} else {

return "Word not found";

}

}

};

int main() {

Dictionary dict;

dict.addWord("apple", "a fruit");

dict.addWord("book", "a written or printed work consisting of pages glued or sewn together along one side and bound in covers.");

dict.addWord("computer", "an electronic device for storing and processing data");

cout << "Meaning of 'apple': " << dict.searchWord("apple") << endl;

cout << "Meaning of 'book': " << dict.searchWord("book") << endl;

cout << "Meaning of 'computer': " << dict.searchWord("computer") << endl;

cout << "Meaning of 'cat': " << dict.searchWord("cat") << endl; // Word not found

return 0;

}