#Solution-1:



#include <stdio.h> // Standard C library for input/output

#include <iostream> // Standard C++ library for input/output

using namespace std; // Using the standard namespace

class A { // Declaration of class A

public:

A(); // Default constructor

A(int); // Parameterized constructor

A(const A&); // Copy constructor

~A(); // Destructor

public:

void operator=(const A& rhs); // Assignment operator

void Print(); // Non-const print function

void PrintC() const; // Const print function

int x; // Public member variable

public:

int& X() { return x; } // Reference to x

};

// Implementation of class A member functions

A::A() : x(0) { // Default constructor, initializes x to 0

cout << "Hello from A::A() Default constructor" << endl; // Output message

}

A::A(int i) : x(i) { // Parameterized constructor, initializes x to the given value

cout << "Hello from A::A(int) constructor" << endl; // Output message

}

A::A(const A& a) : x(a.x) { // Copy constructor, copies x from the given object

cout << "Hello from A::A(const A&) constructor" << endl; // Output message

}

A::~A() { // Destructor, cleans up resources when the object is destroyed

cout << "Hello from A::A destructor" << endl; // Output message

}

void A::operator=(const A& rhs) { // Assignment operator, assigns x from the given object

x = rhs.x;

cout << "Hello from A::operator=" << endl; // Output message

}

void A::Print() { // Non-const print function

cout << "A::Print(), x " << x << endl; // Output message with the value of x

}

void A::PrintC() const { // Const print function

cout << "A::PrintC(), x " << x << endl; // Output message with the value of x

}

// Functions to demonstrate different ways of passing and manipulating objects of class A

void PassAByValue(A a) { // Pass by value, a copy of the object is passed

cout << "PassAByValue, a.x " << a.x << endl; // Output message with the value of a.x

a.x++; // Increment x of the passed object

a.Print(); // Call non-const print function

a.PrintC(); // Call const print function

}

void PassAByReference(A& a) { // Pass by reference, original object is passed

cout << "PassAByReference, a.x " << a.x << endl; // Output message with the value of a.x

a.x++; // Increment x of the passed object

a.Print(); // Call non-const print function

a.PrintC(); // Call const print function

}

void PassAByConstReference(const A& a) { // Pass by const reference, original object cannot be modified

cout << "PassAByReference, a.x " << a.x << endl; // Output message with the value of a.x

a.PrintC(); // Call const print function

//a.Print(); // Attempt to call non-const print function fails

// Compiler error from above line. Why? - Because the member function Print() is not const

}

void PassAByPointer(A\* a) { // Pass by pointer

cout << "PassAByPointer, a->x " << a->x << endl; // Output message with the value of a->x

a->x++; // Increment x of the pointed object

a->Print(); // Call non-const print function

a->PrintC(); // Call const print function

}

int main() {

cout << "Creating a0"; getchar(); // Output message, wait for user input

A a0; // Calls default constructor

cout << "Creating a1"; getchar(); // Output message, wait for user input

A a1(1); // Calls parameterized constructor

cout << "Creating a2"; getchar(); // Output message, wait for user input

A a2(a0); // Calls copy constructor

cout << "Creating a3"; getchar(); // Output message, wait for user input

A a3 = a0; // Also calls copy constructor

cout << "Assigning a3 = a1"; getchar(); // Output message, wait for user input

a3 = a1; // Calls assignment operator

// Call some of the "A" subroutines

cout << "PassAByValue(a1)"; getchar(); // Output message, wait for user input

PassAByValue(a1); // Demonstrates pass by value

cout << "After PassAByValue(a1)" << endl;

a1.Print(); // Original object not changed

cout << "PassAByReference(a1)"; getchar(); // Output message, wait for user input

PassAByReference(a1); // Demonstrates pass by reference

cout << "After PassAByReference(a1)" << endl;

a1.Print(); // Original object changed

cout << "PassAByConst(a1)"; getchar(); // Output message, wait for user input

PassAByConstReference(a1); // Demonstrates pass by const reference

cout << "After PassAByConstReference(a1)" << endl;

a1.Print(); // Original object not changed

cout << "PassAByPointer(&a1)"; getchar(); // Output message, wait for user input

PassAByPointer(&a1); // Demonstrates pass by pointer

cout << "After PassAByPointer(a1)" << endl;

a1.Print(); // Original object changed

cout << "a1.X() = 10"; getchar(); // Output message, wait for user input

a1.X() = 10; // Demonstrates using a member function to access and modify a member variable

a1.Print();

cout << "PassAByConstReference"; getchar(); // Output message, wait for user input

PassAByConstReference(20); // This compiles because it accepts a const reference to an anonymous object

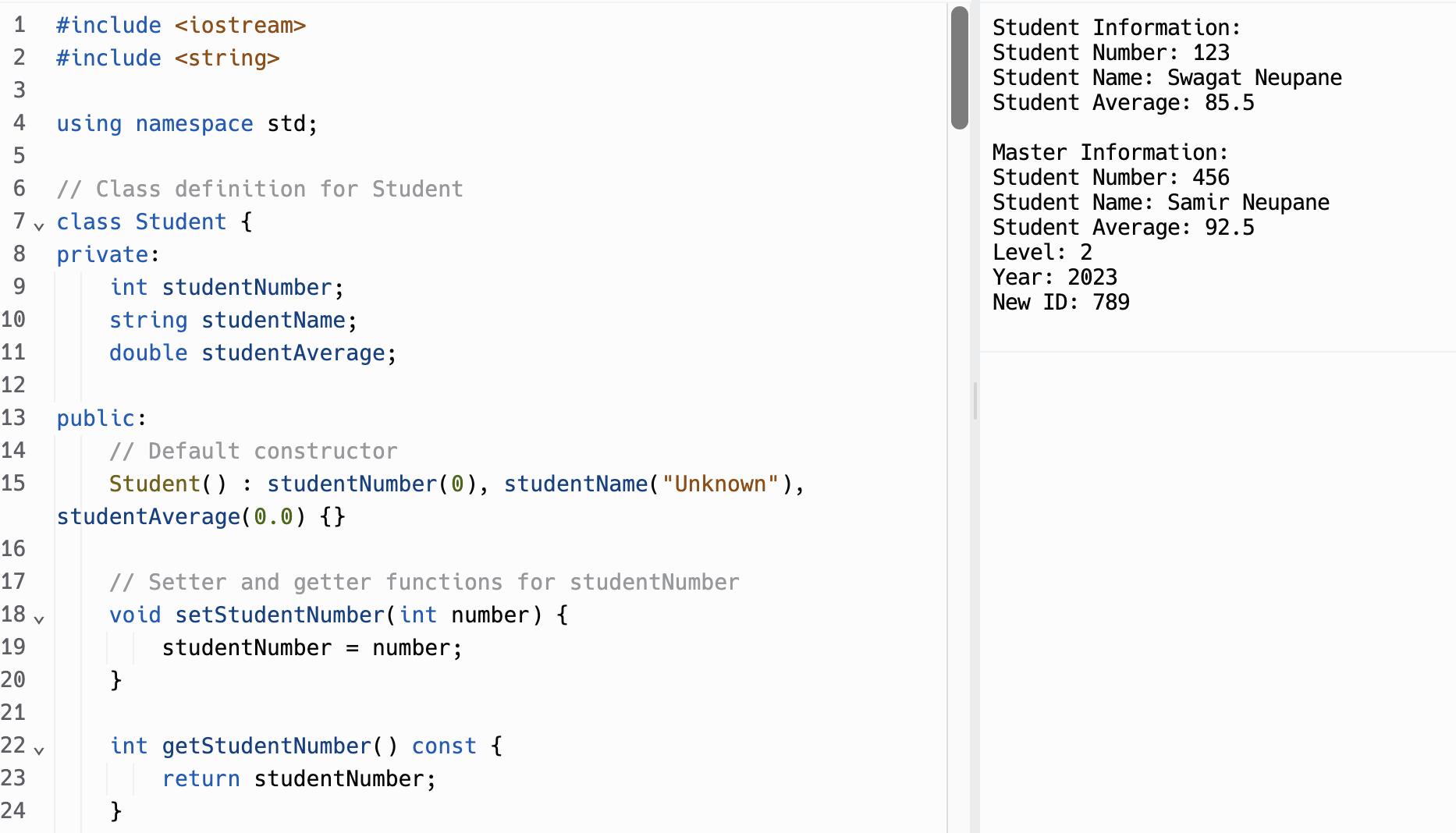
// Why does the above compile? What does it do? - It compiles because it accepts a const reference to an anonymous object.

// It creates a temporary object of class A with value 20 and passes it to the function PassAByConstReference.

return 0; // Indicates successful termination

}

#Solution-2:



#include <iostream>

#include <string>

using namespace std;

// Class definition for Student

class Student {

private:

int studentNumber;

string studentName;

double studentAverage;

public:

// Default constructor

Student() : studentNumber(0), studentName("Unknown"), studentAverage(0.0) {}

// Setter and getter functions for studentNumber

void setStudentNumber(int number) {

studentNumber = number;

}

int getStudentNumber() const {

return studentNumber;

}

// Setter and getter functions for studentName

void setStudentName(const string& name) {

studentName = name;

}

string getStudentName() const {

return studentName;

}

// Setter and getter functions for studentAverage

void setStudentAverage(double average) {

studentAverage = average;

}

double getStudentAverage() const {

return studentAverage;

}

// Print function to display the information

void Print() const {

cout << "Student Number: " << studentNumber << endl;

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

cout << "Student Average: " << studentAverage << endl;

}

};

// Class definition for GraduateStudent, inherits from Student

class GraduateStudent : public Student {

private:

int level;

int year;

public:

// Constructor

GraduateStudent() : level(0), year(0) {}

// Setter and getter functions for level

void setLevel(int l) {

level = l;

}

int getLevel() const {

return level;

}

// Setter and getter functions for year

void setYear(int y) {

year = y;

}

int getYear() const {

return year;

}

// Print function to display the information

void Print() const {

// Call base class Print function

Student::Print();

cout << "Level: " << level << endl;

cout << "Year: " << year << endl;

}

};

// Class definition for Master, inherits from GraduateStudent

class Master : public GraduateStudent {

private:

int newId;

public:

// Constructor

Master() : newId(0) {}

// Setter and getter functions for newId

void setNewId(int id) {

newId = id;

}

int getNewId() const {

return newId;

}

// Print function to display the information

void Print() const {

// Call base class Print function

GraduateStudent::Print();

cout << "New ID: " << newId << endl;

}

};

int main() {

// Declare object of type Student

Student studentObj;

studentObj.setStudentNumber(123);

studentObj.setStudentName("Swagat Neupane");

studentObj.setStudentAverage(85.5);

// Print information for the student object

cout << "Student Information:" << endl;

studentObj.Print();

cout << endl;

// Declare object of type Master

Master masterObj;

masterObj.setStudentNumber(456);

masterObj.setStudentName("Samir Neupane");

masterObj.setStudentAverage(92.5);

masterObj.setLevel(2);

masterObj.setYear(2023);

masterObj.setNewId(789);

// Print information for the master object

cout << "Master Information:" << endl;

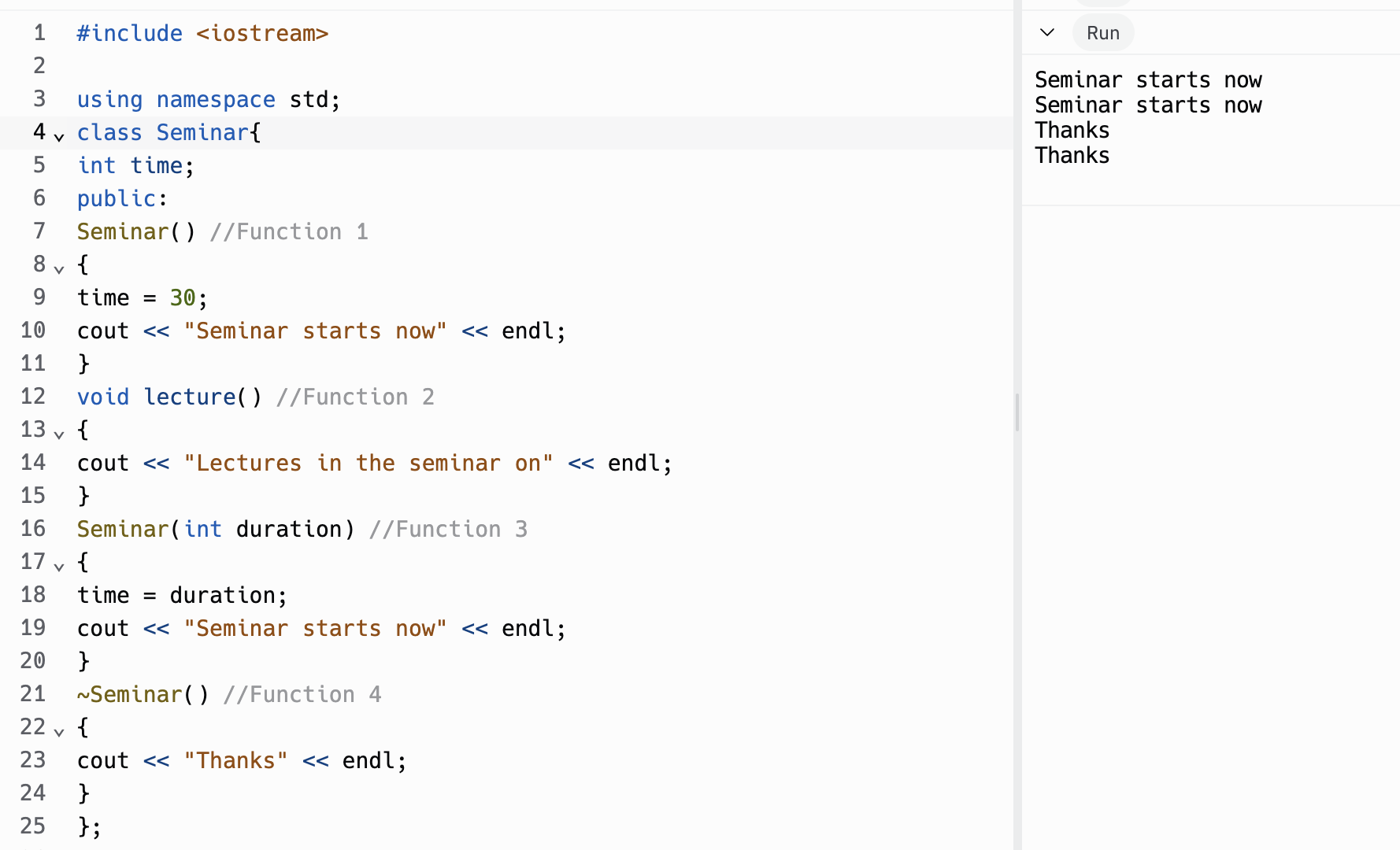
masterObj.Print();

return 0;

}

#Solution-3:

a.



b. In Object-Oriented Programming (OOP), Function 4, which is the destructor ~Seminar(), is referred to as the destructor. It gets invoked or called automatically when an object of the class is destroyed or goes out of scope. The destructor is responsible for releasing resources, performing cleanup, or any necessary finalization actions.

c. Function 1 and Function 3 together illustrate the concept of constructor overloading. Constructor overloading allows a class to have multiple constructors with different parameter lists. In this case, the class Seminar has two constructors with different parameters: one with no parameters (Function 1) and another with an int parameter (Function 3). This allows flexibility when creating objects of the class, depending on the specific needs and requirements.

#Solution-4:



#include <iostream>

#include <cstring>

using namespace std;

class Test {

private:

char paper[20];

int marks;

public:

Test() // Function 1

{

strcpy(paper, "Computer");

marks = 0;

}

Test(const char p[]) // Function 2

{

strcpy(paper, p);

marks = 0;

}

Test(int m) // Function 3

{

strcpy(paper, "Computer");

marks = m;

}

Test(const char p[], int m) // Function 4

{

strcpy(paper, p);

marks = m;

}

void Print() const {

cout << "Paper: " << paper << endl;

cout << "Marks: " << marks << endl;

}

};

int main() {

// Execute Function 1

Test test1;

cout << "Result of Function 1:" << endl;

test1.Print();

cout << endl;

// Execute Function 2 with a specified paper name ("Math")

Test test2("Math");

cout << "Result of Function 2:" << endl;

test2.Print();

cout << endl;

// Execute Function 3 with a specified marks value (80)

Test test3(80);

cout << "Result of Function 3:" << endl;

test3.Print();

cout << endl;

// Execute Function 4 with specified paper name ("Science") and marks value (90)

Test test4("Science", 90);

cout << "Result of Function 4:" << endl;

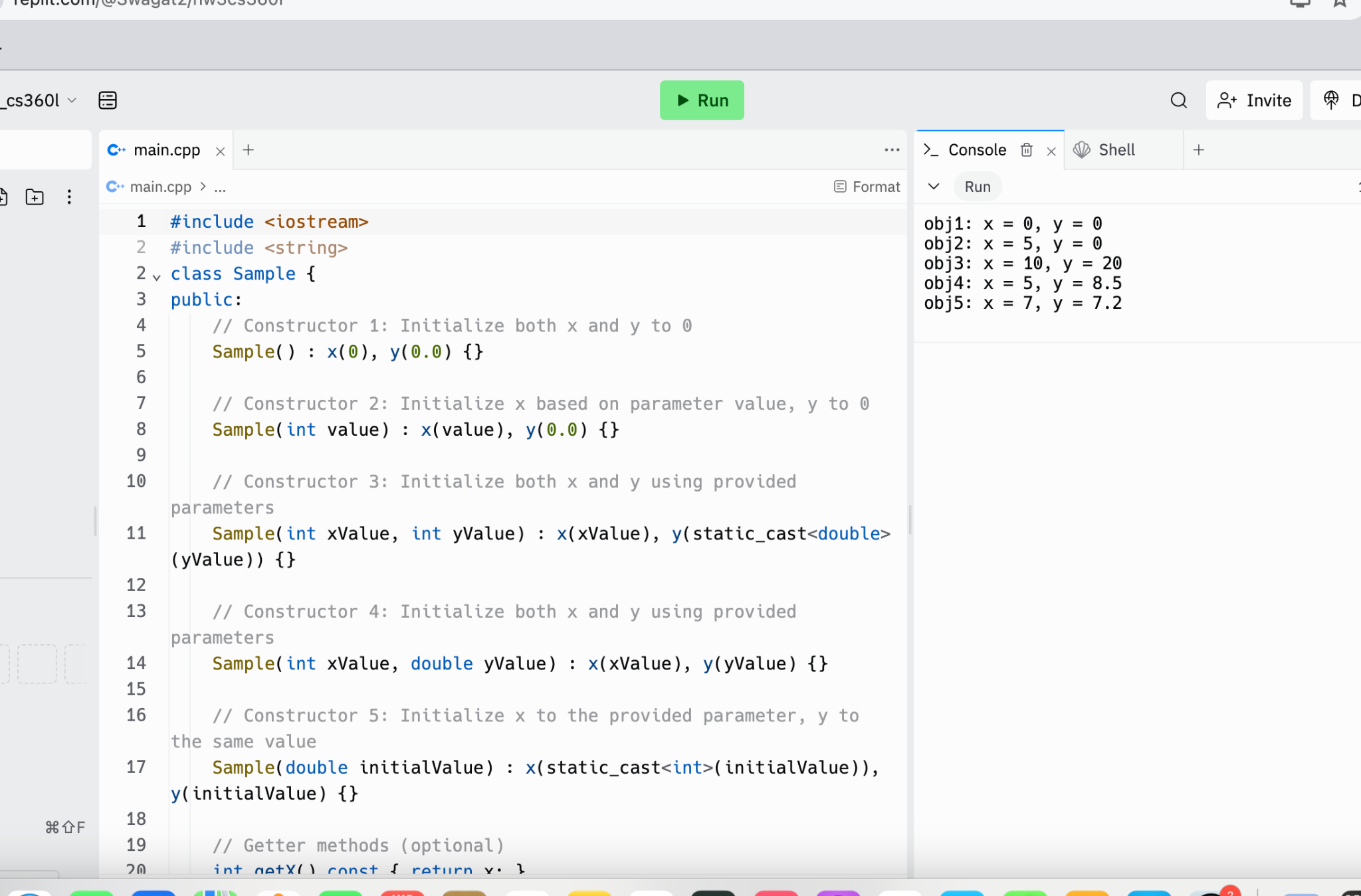
test4.Print();

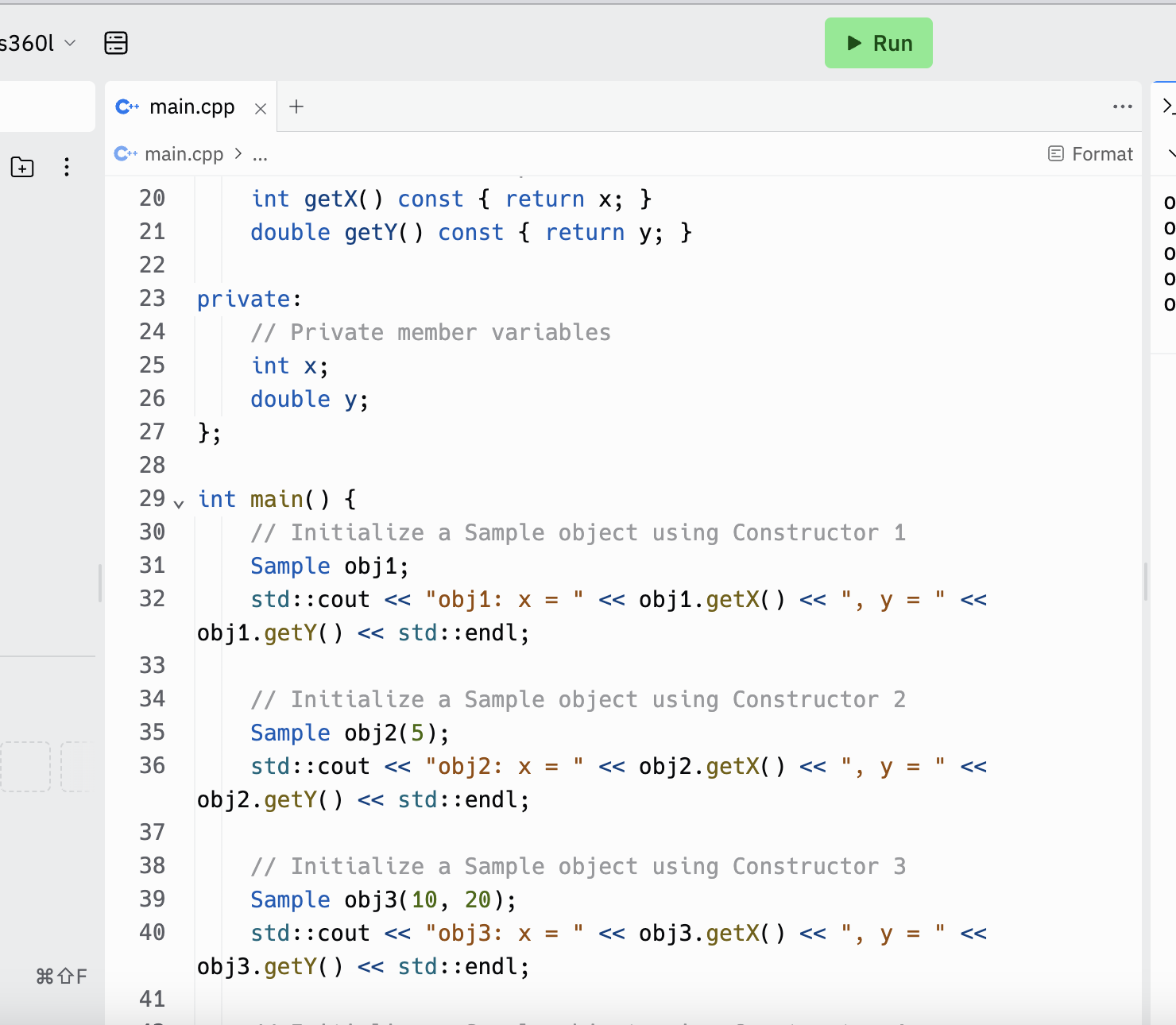
cout << endl;

return 0;

}

#Solution-5:







#include <iostream>

class Sample {

public:

// Constructor 1: Initialize both x and y to 0

Sample() : x(0), y(0.0) {}

// Constructor 2: Initialize x based on parameter value, y to 0

Sample(int value) : x(value), y(0.0) {}

// Constructor 3: Initialize both x and y using provided parameters

Sample(int xValue, int yValue) : x(xValue), y(static\_cast<double>(yValue)) {}

// Constructor 4: Initialize both x and y using provided parameters

Sample(int xValue, double yValue) : x(xValue), y(yValue) {}

// Constructor 5: Initialize x to the provided parameter, y to the same value

Sample(double initialValue) : x(static\_cast<int>(initialValue)), y(initialValue) {}

// Getter methods (optional)

int getX() const { return x; }

double getY() const { return y; }

private:

// Private member variables

int x;

double y;

};

int main() {

// Initialize a Sample object using Constructor 1

Sample obj1;

std::cout << "obj1: x = " << obj1.getX() << ", y = " << obj1.getY() << std::endl;

// Initialize a Sample object using Constructor 2

Sample obj2(5);

std::cout << "obj2: x = " << obj2.getX() << ", y = " << obj2.getY() << std::endl;

// Initialize a Sample object using Constructor 3

Sample obj3(10, 20);

std::cout << "obj3: x = " << obj3.getX() << ", y = " << obj3.getY() << std::endl;

// Initialize a Sample object using Constructor 4

Sample obj4(5, 8.5);

std::cout << "obj4: x = " << obj4.getX() << ", y = " << obj4.getY() << std::endl;

// Initialize a Sample object using Constructor 5

Sample obj5(7.2);

std::cout << "obj5: x = " << obj5.getX() << ", y = " << obj5.getY() << std::endl;

// Rest of your code goes here...

return 0;

}