



**FALL 2023**

Lab Manual

**OBJECT ORIENTED PROGRAMMING**

CSC-213

Lab Manual

**OBJECT ORIENTED PROGRAMMING**

**Semester : Fall 2023**

**Program : BS**

**Course Title and Name : CSC 213**

**Credits : 1**

**Faculty : Mehar Khan Niazi**

**Student Name : Sadia Muddasir**

**Student ID : 20221-33243**

**Total Marks : 100**

**Obtained Marks :**

**Submitted Date :5-oct-23**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Lab No.** | **Experiment Title** | **Date** |
| **1** | **Transition from C to C++**  **●** Understanding the shift from C to C++  ● Setting up the development environment (IDE,  compiler)  ● Basic syntax and structure transition  ● Hands-on experience with cin, cout, std::  Namespace, variables, loops, and conditions |  |
| **2** | **Overview of programming paradigms/ Introduction to Classes and Objects**  ● Importance of OOP, Why OOP, Procedural  Programing vs OOP  ● Basic syntax and structure of a class in C++  ● Introduction to classes, objects  ● Access Specifiers in Classes |  |
| **3** | **Functions and Constructors**   * Introduction to Constructors * Default Constructors * Passing and Returning Objects to/from Functions * Private Functions and Variable Scopes |  |
| **4** | **Overloading**   * Method Overloading * Constructor Overloading (Parameterized Constructors) * Introduction to Copy Constructors |  |

**Week 4**

**Lab: Overloading**

**Lab Objective:**

The objective of this lab is to provide students with a comprehensive understanding of overloading in C++, including method overloading, constructor overloading (Parameterized Constructor), and introduction to copy constructors.

**Tools/Software Requirement:**

* Dev-C++

**Description:**

**Introduction to Method Overloading in C++:**

Method overloading is the process of overloading a method with the same name but different parameters. C++ provides this method of overloading features. Method overloading allows users to use the same name as another, but the parameters passed to the methods should differ. The return type of methods can be the same or different. This lab manual will discuss method overloading in C++ with its working and examples.

**Syntax**:

**int sample(a){**

**}**

**int sample(int a , int b) {**

**}**

**float sample(float a, float b) {**

**}**

Here the sample is the name of the method. This method has different arguments. Return types used for these methods are different. We can use the same or different return types.

**Examples to Implement Methods Overloading in C++:**

**Example 1: Program to implement the method overloading with a different number of arguments.**

**#include <iostream>**

**using namespace std;**

**class addition {**

**public:**

**int addMethod(int x, int y) {**

**return x + y;**

**}**

**int addMethod(int x, int y, int z) {**

**return x + y + z;**

**}**

**};**

**int main(void) {**

**addition add;**

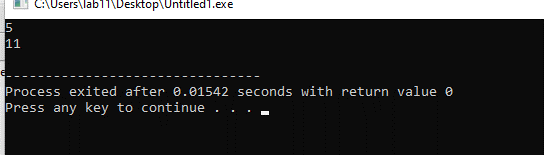
**cout << add.addMethod(2, 3) << endl;**

**cout << add.addMethod(2, 3, 6) << endl;**

**return 0;**

**}**

**Output & Explanation/Reason:**



**Reason:**

This code means that you can have multiple methods in a class with the same name as long as they have different parameter lists. This is known as function overloading, and it allows you to use the same method name for similar operations with different argument variations.

**Example 2: Program to implement the method overloading with a different number of arguments and different return types.**

**#include <iostream>**

**using namespace std;**

**class addition {**

**public:**

**int addMethod(int x, int y) {**

**return x + y;**

**}**

**float addMethod(float x, float y, float z) {**

**return x + y + z;**

**}**

**};**

**+**

**int main(void) {**

**addition add;**

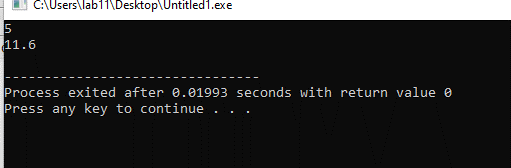
**cout << add.addMethod(2, 3) << endl;**

**cout << add.addMethod(2.2, 3.3, 6.1) << endl;**

**return 0;**

**}**

**Output & Explanation/Reason:**



**Reason:** This code make a class of addition with two different "addMethod" functions. The first "addMethod" adds two integers and returns their sum, while the second "addMethod" adds three floating-point numbers and returns their sum. In the "main" function, an object of the "addition" class is created, and both versions of "addMethod" are called with different sets of numbers. The results are printed using "cout." This demonstrates the concept of function overloading, where you can have multiple functions with the same name but different parameters in a class, making it flexible to perform similar operations with different types of data.

**Example 3: If we try to pass the float numbers to the int return type, it will not accept the argument and will throw an error.**

**#include <iostream>**

**using namespace std;**

**class addition {**

**public:**

**int addMethod(int x, int y) {**

**return x + y;**

**}**

**float addMethod(int x, int y) {**

**return x + y;**

**}**

**};**

**int main(void) {**

**addition add;**

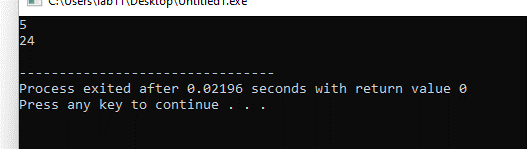
**cout << add.addMethod(2, 3) << endl;**

**cout << add.addMethod(21, 3) << endl;**

**return 0;**

**}**

**Output & Explanation/Reason:**



**Reason:**

This code defines a C++ class with two "addMethod" functions. One adds integers and the other adds floats. In the "main" function, it uses the correct version of the function based on the data types of the input values and prints the results.

|  |
| --- |
| **Note:** Types of arguments and the number of arguments passed to the method should not be the same. |

**Constructor Overloading:**

Constructors can be overloaded in a similar way as function overloading.

Overloaded constructors have the same name (name of the class) but the different number of arguments. Depending upon the number and type of arguments passed, the corresponding constructor is called.

**Examples to Implement Constructor Overloading in C++:**

**Example 1: Constructor overloading**

// C++ program to demonstrate constructor overloading

#include <iostream>

using namespace std;

class Person {

private:

int age;

public:

// 1. Constructor with no arguments

Person() {

age = 20;

}

// 2. Constructor with an argument

Person(int a) {

age = a;

}

int getAge() {

return age;

}

};

int main() {

Person person1, person2(45);

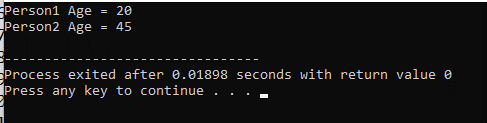
cout << "Person1 Age = " << person1.getAge() << endl;

cout << "Person2 Age = " << person2.getAge() << endl;

return 0;

}

**Output & Explanation/Reason:**



**Reason**:  
This code defines a Person class with two constructors—one for a default age of 20 and another for specifying a custom age when creating a person object. In the "main" function, it creates two "Person" objects, one with the default age and another with a custom age of 45. It then retrieves and prints the ages of both objects, demonstrating the use of constructors to initialize object attributes differently.

**Example 2: Constructor overloading**

// C++ program to demonstrate constructor overloading

#include <iostream>

using namespace std;

class Room {

private:

double length;

double breadth;

public:

// 1. Constructor with no arguments

Room() {

length = 6.9;

breadth = 4.2;

}

// 2. Constructor with two arguments

Room(double l, double b) {

length = l;

breadth = b;

}

// 3. Constructor with one argument

Room(double len) {

length = len;

breadth = 7.2;

}

double calculateArea() {

return length \* breadth;

}

};

int main() {

Room room1, room2(8.2, 6.6), room3(8.2);

cout << "When no argument is passed: " << endl;

cout << "Area of room = " << room1.calculateArea() << endl;

cout << "\nWhen (8.2, 6.6) is passed." << endl;

cout << "Area of room = " << room2.calculateArea() << endl;

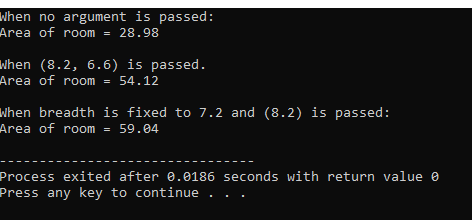
cout << "\nWhen breadth is fixed to 7.2 and (8.2) is passed:" << endl;

cout << "Area of room = " << room3.calculateArea() << endl;

return 0;

}

**Output & Explanation/Reason:**



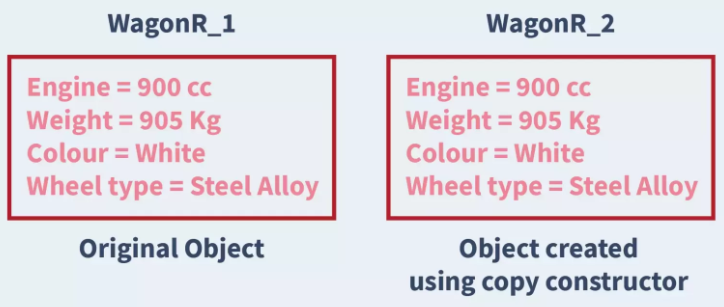
**Reason:**

In this code the "main" function, creates three Room objects: "room1" using the default constructor, "room2" with custom length and breadth, and "room3" with a custom length and a fixed breadth. It then calculates and prints the area of each room using the "calculateArea" method. This code demonstrates constructor overloading, allowing you to create Room objects with different parameter combinations.

**What is a Copy Constructor?**

A constructor is used to initialize an object. A copy constructor is a member function of a class that initializes an object with an existing object of the same class. In other words, it creates a copy of an already existing object and stores it in a new object. Consider an object of class Car, WagonR\_1. A copy constructor can create another object of this car, which will be the same as WagonR\_1, say WagonR\_2.

New Object, i.e., WagonR\_2, is the exact copy of the existing object.



**Syntax to create a Copy Constructor in C++:**

className(className &objectName);

The following code describes an example of a simple Copy Constructor in C++:

sampleClass(sampleClass &obj) {

dataMember1 = obj.dataMember1;

dataMember2 = obj.dataMember2;

}

**Example 1: Copy Constructor**

#include <iostream>

using namespace std;

// declare a class

class Wall {

private:

double length;

double height;

public:

// initialize variables with parameterized constructor

Wall(double len, double hgt) {

length = len;

height = hgt;

}

// copy constructor with a Wall object as parameter

// copies data of the obj parameter

Wall(Wall &obj) {

length = obj.length;

height = obj.height;

}

double calculateArea() {

return length \* height;

}

};

int main() {

// create an object of Wall class

Wall wall1(10.5, 8.6);

// copy contents of wall1 to wall2

Wall wall2 = wall1;

// print areas of wall1 and wall2

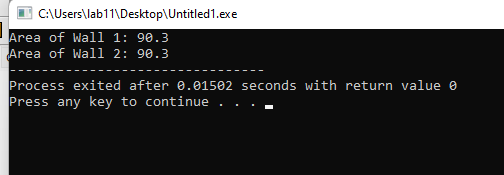
cout << "Area of Wall 1: " << wall1.calculateArea() << endl;

cout << "Area of Wall 2: " << wall2.calculateArea();

return 0;

}

**Output & Explanation/Reason:**



**Reason:**

This code defines a Wall class that represents walls with length and height attributes. It demonstrates the use of a parameterized constructor to initialize these attributes when creating a "Wall" object. Additionally, it showcases a copy constructor that allows you to create a new "Wall" object (wall2) by copying the data from an existing one (wall1). It calculates and prints the areas of both wall1 and wall2, highlighting how the copy constructor duplicates the attributes of the original object. Essentially, this code illustrates the concept of constructor overloading and object copying in C++ classes, which is useful for managing and replicating object data.

**Lab Tasks:**

**Requirements (Code, Output and Reason)**

**Task 1: Method Overloading**

1. Create a class `MathOperations` with two methods for addition: `int add(int x, int y)` and `double add(double x, double y)`. Explain the concept of method overloading in simple terms.

**Code**:

#include <iostream>

using namespace std;

class MathOperations {

public:

int add(int x, int y) {

return x + y;

}

double add(double x, double y) {

return x + y;

}

};

int main() {

MathOperations math;

int resultInt = math.add(5, 3);

double resultDouble = math.add(2.5, 3.7);

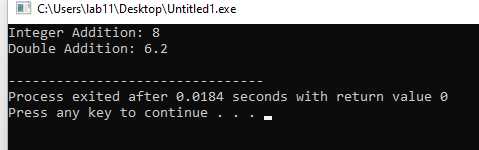
cout << "Integer Addition: " << resultInt << endl;

cout << "Double Addition: " << resultDouble << endl;

return 0;

}

**Output**:



**Reason:**

In this code, we've defined a class called "MathOperations" with two functions named "add." The first "add" function works with integers, adding two of them together and returning the result as an integer. The second add function operates on doubles, adding two double values and returning the result as a double. After then in the main part of the code, we create an instance of the "MathOperations" class and use both versions of the add function to perform addition, both with integers and doubles.

1. In the `MathOperations` class, write a parameterized constructor that initializes the object. Can you have multiple constructors in a class? Explain briefly.

**Code:**

#include <iostream>

using namespace std;

class MathOperations {

public:

MathOperations() {

}

MathOperations(int initialValue) {

value = initialValue;

}

int add(int x, int y) {

return x + y;

}

double add(double x, double y) {

return x + y;

}

int getValue() {

return value;

}

private:

int value;

};

int main() {

MathOperations math;

cout << "Default Value: " << math.getValue() << endl;

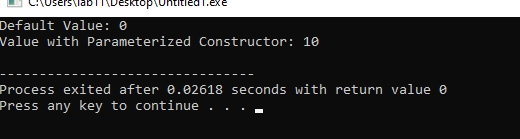
MathOperations mathWithInitialValue(10);

cout << "Value with Parameterized Constructor: " << mathWithInitialValue.getValue() << endl;

return 0;

}

**Output**:



**Reason**:

In this code I’ve added a parameterized constructor MathOperations(int initialValue) to the MathOperations class. This constructor allows you to initialize the object with a specified initial value. In the main function, we demonstrate both the default constructor and the parameterized constructor by creating two instances of the MathOperations class. Multiple constructors in a class provide flexibility in how objects can be initialized, allowing you to create objects with different initial states or configurations.

1. In the `MathOperations` class, create an object and demonstrate the use of method overloading to perform addition with both integers and doubles.

**Code**:

#include <iostream>

using namespace std;

class MathOperations {

public:

int add(int x, int y) {

return x + y;

}

double add(double x, double y) {

return x + y;

}

};

int main() {

MathOperations math;

int resultInt = math.add(5, 3);

double resultDouble = math.add(2.5, 3.7);

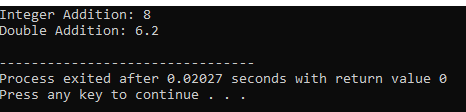
cout << "Integer Addition: " << resultInt << endl;

cout << "Double Addition: " << resultDouble << endl;

return 0;

}

**Output:**

****

**Reason:**

In this code I’ve created a class of the MathOperations class named "math." Then, we use the add method to perform addition with both integers and doubles, showcasing method overloading. The resultInt and resultDouble variables store the results of these addition operations, and we print them to the console. This demonstrates how you can use method overloading to perform different types of addition with the same method name, depending on the data types of the arguments.

**Task 2: Constructor Overloading (Parameterized Constructors)**

1. Create a class `Person` with private data members `name` and `age`. Implement a parameterized constructor that takes name and age as arguments and a default constructor that sets name to "Unknown" and age to 0. Why is constructor overloading useful?

**Code**:

#include <iostream>

using namespace std;

class Person {

private:

string name;

int age;

public:

Person(string n, int a) {

name = n;

age = a;

}

Person() {

name = "Unknown";

age = 0;

}

void displayInfo() {

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

cout << "Age: " << age << endl;

}

};

int main() {

Person person1("Sadia", 19);

cout << "Person 1:" << endl;

person1.displayInfo();

Person person2;

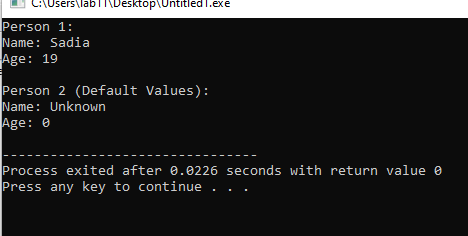
cout << "\nPerson 2 (Default Values):" << endl;

person2.displayInfo();

return 0;

}

**Output**:



**Reason:**

This code defines a Person class with private attributes name and age. It showcases constructor overloading, which lets you create Person objects with custom or default values. There's a parameterized constructor for custom values and a default constructor for setting standard values. In the main function, two Person objects are created: one with custom data and another with default data, demonstrating how constructor overloading provides flexibility in object initialization.

2. In the `Person` class, create two objects, one using the parameterized constructor and another using the default constructor. Display the values of both objects. Share the code and the result.

**Code:**

#include <iostream>

using namespace std;

class Person {

private:

string name;

int age;

public:

Person(string n, int a) {

name = n;

age = a;

}

Person() {

name = "Unknown";

age = 0;

}

void displayInfo() {

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

cout << "Age: " << age << endl;

}

};

int main() {

Person person1("Sadia", 19);

cout << "Person 1:" << endl;

person1.displayInfo();

Person person2;

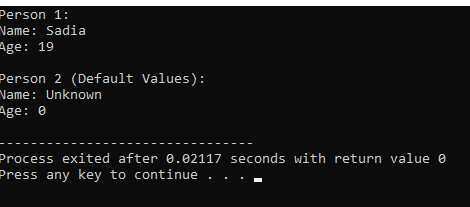
cout << "\nPerson 2 (Default Values):" << endl;

person2.displayInfo();

return 0;

}

**Output:**

****

**Reason:**In this code, we create two Person objects: person1 using the parameterized constructor with custom values ("Sadia" and 19) and person2 using the default constructor. We then display the values of both objects, showing that person1 has the custom data, while person2 has the default values.

**Task 3: Copy Constructors**

1. Create a class `Book` with private data members `title` and `pages`. Initialize one with values, and then implement a copy constructor that copies the values of one `Book` object to another. Explain when and why a copy constructor is used in simple terms.

**Code:**

#include <iostream>

#include <string>

using namespace std;

class Book {

private:

string title;

int pages;

public:

Book(string t, int p) {

title = t;

pages = p;

}

Book(const Book& other) {

title = other.title;

pages = other.pages;

}

void displayInfo() {

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

cout << "Pages: " << pages << endl;

}

};

int main() {

Book book1("Around the world in 80 days ", 160);

Book book2 = book1;

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

book1.displayInfo();

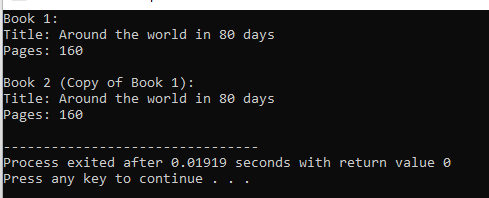
cout << "\nBook 2 (Copy of Book 1):" << endl;

book2.displayInfo();

return 0;

}

**Output:**

****

**Reason:**

In this code I define a Book class with private attributes title and pages. It illustrates the use of a copy constructor, which allows you to create a new Book object (book2) by copying the values from an existing one (book1). This is valuable for maintaining data integrity and creating duplicates of objects while preserving the original data.