



Lab Manual

**OBJECT ORIENTED PROGRAMMING**

**Semester : Fall 2023**

**Program : BS**

**Course Title and Name : CSC 213**

**Credits : 1**

**Faculty : Mehar Khan Niazi**

**Student Name :**

**Student ID :**

**Total Marks : 100**

**Obtained Marks :**

**Submitted Date :**

**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 |  |
| **5** | **Classes and Objects Continued**   * Working with Arrays of Objects * Destructors * Constant Member Functions & Objects |  |

| **6** | **Inheritance**   * Creating Derived and Base Classes * Derived Class Constructors * Understanding the Protected Specifier * Function Overriding * Scope Resolution with Overridden Functions |  |
| --- | --- | --- |
| **7** | **Level of Inheritance**   * Single Inheritance * Multilevel Inheritance * Multiple Inheritance |  |
| **8.** | **Mid-Term Break** |  |
| **9, 10,11** | **Version control**   * Version Control with Git and GitHub * Collaborative software development using version control.   **Polymorphism**   * Virtual Functions * Late and Early Binding * Friend Functions * Abstract Classes in Terms of Polymorphism * Function Overloading vs. Overriding * Type Casting * Handling Multiple Inheritance Issues |  |
| **12** | **Operator Overloading**   * Overloading Arithmetic Operators * Overloading Subscript Operators * Overloading the () Operator * Creating Functions that Perform Substring Operations * Working with Unary Operators |  |

**Week 12 Lab: Operator Overloading**

**Lab Objective:**

The aim of this lab session is to introduce students to the concept of operator overloading in Object-Oriented Programming (OOP) using C++. The objectives are:

* Understand the concept of operator overloading.
* Implement overloading for arithmetic operators, subscript operators, the () operator, and unary operators.
* Recognize real-world applications of operator overloading.
* Gain hands-on experience through practical examples demonstrating operator overloading.

**Tools/Software Requirement:**

* Dev-C++ / Online C++ Compiler

**Theory:**

**Operator Overloading:**

Operator overloading allows defining multiple behaviors for operators based on the types of their operands. In C++, you can overload various operators to work with user-defined data types.

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism. For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, etc.

**Code Examples:**

**Example 1: Overloading Arithmetic Operators**

**Code:**

#include <iostream>

class Money {

private:

    int dollars;

    int cents;

public:

    Money(int d = 0, int c = 0) : dollars(d), cents(c) {}

    void setDollars(int d) { dollars = d; }

    void setCents(int c) { cents = c; }

    int getDollars() const { return dollars; }

    int getCents() const { return cents; }

    void simplify();

    Money operator+(const Money& other) const;

    Money operator-(const Money& other) const;

};

int main() {

    Money money1(4, 75);

    Money money2(1, 25);

    Money money3;

    money3 = money1 + money2;

    std::cout << money3.getDollars() << " " << money3.getCents() << std::endl;

    return 0;

}

void Money::simplify() {

    dollars += cents / 100;

    cents = cents % 100;

}

Money Money::operator+(const Money& other) const {

    Money temp;

    temp.dollars = dollars + other.dollars;

    temp.cents = cents + other.cents;

    temp.simplify();

    return temp;

}

Money Money::operator-(const Money& other) const {

    Money temp;

    temp.dollars = dollars - other.dollars;

    temp.cents = cents - other.cents;

    temp.simplify();

    return temp;

}

**Output:**  


**Example 2: Overloading Arithmetic Operators**

**Complex Number Addition**

**Code:**

#include <iostream>

using namespace std;

class Complex {

private:

    double real;

    double imag;

public:

    // Constructor to initialize the real and imaginary parts

    Complex(double r, double i) : real(r), imag(i) {}

    // Overloaded + operator to add two Complex numbers

    Complex operator+(Complex& other) {

        return Complex(real + other.real, imag + other.imag);

    }

    // Function to display the Complex number

    void display() {

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

    }

};

int main() {

    // Create two Complex numbers

    Complex num1(2.5, 3.0);

    Complex num2(1.5, 2.5);

    // Use the overloaded + operator to add the two Complex numbers

    Complex result = num1 + num2;

    // Display the result

    result.display();

    return 0;

}

**Output:**

**Example 3: Overloading Subscript Operators:**

The Subscript or Array Index Operator is denoted by ‘[]’. This operator is generally used with arrays to retrieve and manipulate the array elements.

**Code:**

#include <iostream>

using namespace std;

class FixedArray {

private:

    static const int SIZE = 5;  // Size of the array

    int data[SIZE];              // Array to store data

public:

    // Overloading subscript operator for read and write access

    int& operator[](int index) {

        // Check if the index is out of bounds

        if (index < 0 || index >= SIZE) {

            // Throw an exception for an out-of-bounds index

            throw out\_of\_range("Index out of bounds");

        }

        // Return a reference to the element at the specified index

        return data[index];

    }

    // Display the elements of the array

    void display() const {

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

            cout << data[i] << " ";

        }

        cout << endl;

    }

};

int main() {

    FixedArray myArray;

    // Using the subscript operator for assignment

    for (int i = 0; i < 5; ++i) {

        myArray[i] = i \* 2;

    }

    // Using the subscript operator for reading

    cout << "Array elements: ";

    myArray.display();

    // Accessing an out-of-bounds index (will throw an exception)

    cout << "Trying to access an out-of-bounds index:" << endl;

    int value = myArray[10];

    cout << "Value at index 10: " << value << endl;

    return 0;

}

**Output:**

**Example 4:**

**Code:**

#include <iostream>

using namespace std;

int main()

{

    // Character array representing a name

    char name[] = "We are in IT LAB 3";

    // Accessing the character at index 5 using the subscript operator

    // Both of the statements below print the same thing

    cout << name[5] << endl;    // Output: a

    cout << 5 [name] << endl;   // Output: a

    return 0;

}

**Output:**

**Output:**

cout << name[5] << endl; // Output: e

cout << 5 [name] << endl; // Output: e

Both of these statements achieve the same result, and they print the character at the 5th index of the character array name. Let's break it down:

cout << name[5] << endl;: This is the standard way of using the subscript operator. It accesses the element at index 5 in the array name and prints it. In C++, array indices start from 0, so name[5] refers to the 6th character in the array.

cout << 5 [name] << endl;: This might look a bit unusual, but it's a valid and equivalent expression in C++. This is possible because the subscript operator ([]) is commutative. So, 5 [name] is essentially the same as name[5]. It accesses the element at index 5 in the array name and prints it.

**Example 5: Overloading the () Operator**

The function call operator () can be overloaded for objects of class type. When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

**Code:**

#include <iostream>

using namespace std;

class Distance {

   private:

      int feet;             // 0 to infinite

      int inches;           // 0 to 12

   public:

      // required constructors

      Distance() {

         feet = 0;

         inches = 0;

      }

      Distance(int f, int i) {

         feet = f;

         inches = i;

      }

      // overload function call

      Distance operator()(int a, int b, int c) {

         Distance D;

         // just put random calculation

         D.feet = a + c + 10;

         D.inches = b + c + 100 ;

         return D;

      }

      // method to display distance

      void displayDistance() {

         cout << "F: " << feet << " I:" << inches << endl;

      }

};

int main() {

   Distance D1(11, 10), D2;

   cout << "First Distance : ";

   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()

   cout << "Second Distance :";

   D2.displayDistance();

   return 0;

}

**Output:**

**Explanation:**

**Example 6: Creating Functions that Perform Substring Operations:**

**Code:**

#include <iostream>

#include <string>

using namespace std;

class MyString {

private:

    string data;

public:

    MyString(const string& str) : data(str) {}

    MyString operator()(int start, int length) const {

        return MyString(data.substr(start, length));

    }

    void display() const {

        cout << data << endl;

    }

};

int main() {

    MyString original("Hello, Operator Overloading!");

    MyString substring = original(7, 9);

    substring.display(); // Outputs: Operator

    return 0;

}

**Output:**

**Explanation:**

**Example 7: Working with Unary Operators**

Unary operators are the operators that operate on a single operand to produce a specific value. Unary operators in C++ include logical NOT, dereferencing operator, bitwise NOT, increment operator, decrement operator, address of operator, size of the operator, and the unary plus and minus operators.

**Code:**

#include <iostream>

using namespace std;

class Complex {

private:

    double real;

    double imag;

public:

    Complex(double r, double i) : real(r), imag(i) {}

    Complex operator-() const {

        return Complex(-real, -imag);

    }

    void display() const {

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

    }

};

int main() {

    Complex num(2.5, 3.0);

    Complex negation = -num;

    negation.display(); // Outputs: -2.5 + -3i

    return 0;

}

**Output:**

**Lab 12 Tasks:**

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

**Task 1:** **Overloading Arithmetic Operators**

Design a class Money to represent currency. Overload the + operator to add two money objects.

**Task 2:** **Overloading Subscript Operators**

Create a class Matrix and overload the () operator to access and modify elements.

**Task 3:** **Overloading the () Operator**

Design a class Timer to measure time. Overload the () operator to start and stop the timer.

**Task 4:** **Creating Functions that Perform Substring Operations**

Implement a class TextManipulator that works with strings. Overload the () operator to extract substrings.

**Task 5: Working with Unary Operators**

Create a class Temperature to represent temperature values. Overload the - operator to get the negative temperature.