**SRM Institute of Science and Technology**

**Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204**

**Department of Computer Applications**

**Circular – 2024-25**

**BCA 2nd Sem**

**Object Oriented Programming (UCA24202J)**

**Lab Manual**

## Lab 1: I/O Operations and Operators

**Title:** Input/Output Operations and Operators

**Aim:** To understand and implement basic input and output operations in C++, and to utilize various operators.

### Procedure:

1. Write a program to take input from the user using cin.
2. Display the input to the user using cout.
3. Use arithmetic operators (+, -, *, /) to perform calculations.
4. Use relational operators (==, !=, >, <, >=, <=) to compare values.
5. Use logical operators (&&, ||, !) to combine conditions.

### Source Code:

```
#include <iostream>
using namespace std;

int main() {
    int num1, num2;

    // Input
    cout << "Enter the first number: ";
    cin >> num1;
    cout << "Enter the second number: ";
    cin >> num2;

    // Output
    cout << "Number 1: " << num1 << endl;
    cout << "Number 2: " << num2 << endl;

    // Arithmetic operations
    cout << "Sum: " << num1 + num2 << endl;
    cout << "Difference: " << num1 - num2 << endl;
    cout << "Product: " << num1 * num2 << endl;
    if (num2 != 0) {
        cout << "Division: " << (float)num1 / num2 << endl;
    } else {
        cout << "Cannot divide by zero." << endl;
    }

    // Relational operations
    cout << (num1 == num2) << endl;
    cout << (num1 != num2) << endl;
    cout << (num1 >  num2) << endl;
```

```
    // Logical operations
    cout << ((num1 > 0) && (num2 > 0)) << endl;

    return 0;
}
```

**Input:**

```
Enter the first number: 10
Enter the second number: 5
```

**Expected Output:**

```
Number 1: 10
Number 2: 5
Sum: 15
Difference: 5
Product: 50
Division: 2
0
1
1
1
```

## Lab 2: Control Structures and Functions

**Title:** Control Structures and Functions

**Aim:** To implement programs using different control structures (if-else, loops) and to define and use functions.

**Procedure:**

1. Write a program to find the largest of three numbers using if-else.
2. Write a program to print numbers from 1 to 10 using a for loop.
3. Write a function to calculate the factorial of a number.
4. Write a program to demonstrate function calls.

**Source Code:**

```
#include <iostream>
using namespace std;

// Function to calculate factorial
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int num1, num2, num3;

    // Largest of three numbers
    cout << "Enter three numbers: ";
    cin >> num1 >> num2 >> num3;
    if (num1 >= num2 && num1 >= num3) {
        cout << "Largest number: " << num1 << endl;
    } else if (num2 >= num1 && num2 >= num3) {
        cout << "Largest number: " << num2 << endl;
    } else {
        cout << "Largest number: " << num3 << endl;
    }

    // For loop
    cout << "Numbers from 1 to 10: ";
    for (int i = 1; i <= 10; i++) {
        cout << i << " ";
    }
    cout << endl;

    // Function call
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "Factorial of " << num << " is " << factorial(num) << endl;
    return 0;
}
```

**Input:**

```
Enter three numbers: 5 10 2
```

```
Enter a number: 4
```

## Expected Output:

```
Largest number: 10
Numbers from 1 to 10: 1 2 3 4 5 6 7 8 9 10
Factorial of 4 is 24
```

**Lab 3: Classes and Objects**

**Title:** Classes and Objects

**Aim:** To define classes and create objects in C++.

**Procedure:**

1. Define a class Rectangle with data members for length and width.
2. Implement member functions to set and get the values of length and width.
3. Implement a member function to calculate the area of the rectangle.
4. Create objects of the Rectangle class and use them to calculate the area.

**Source Code:**

```
#include <iostream>
using namespace std;

class Rectangle {
private:
    double length;
    double width;

public:
    void setLength(double l) {
        length = l;
    }

    void setWidth(double w) {
        width = w;
    }

    double getLength() const{
        return length;
    }

    double getWidth() const{
        return width;
    }

    double calculateArea() const{
        return length * width;
    }
};

int main() {
    Rectangle rect1, rect2;
    rect1.setLength(10);
    rect1.setWidth(5);
    rect2.setLength(20);
    rect2.setWidth(10);

    cout << "Area of rectangle 1: " << rect1.calculateArea() << endl;
    cout << "Area of rectangle 2: " << rect2.calculateArea() << endl;
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Area of rectangle 1: 50
Area of rectangle 2: 200
```

## Lab 4: Parameterized Constructor and Constructor Overloading

**Title:** Parameterized Constructor and Constructor Overloading

**Aim:** To understand and implement constructors, including parameterized constructors and constructor overloading.

**Procedure:**

1. Define a class Complex to represent complex numbers.
2. Implement a default constructor.
3. Implement a parameterized constructor to initialize the real and imaginary parts.
4. Implement constructor overloading with different parameter lists.
5. Create objects of the Complex class using different constructors.

**Source Code:**

```
#include <iostream>
using namespace std;

class Complex {
private:
    double real;
    double imag;

public:
    // Default constructor
    Complex() : real(0), imag(0) {}

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

    // Constructor overloading
    Complex(double r) : real(r), imag(0) {}

    void display() const{
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1;         // Default constructor
    Complex c2(5, 3);  // Parameterized constructor
    Complex c3(10);    // Constructor overloading

    cout << "c1: ";
    c1.display();
    cout << "c2: ";
    c2.display();
    cout << "c3: ";
    c3.display();
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
c1: 0 + 0i
```

```
c2: 5 + 3i
c3: 10 + 0i
```

## Lab 5: Function Overloading

**Title:** Function Overloading

**Aim:** To understand and implement function overloading in C++.

**Procedure:**

1. Write a function add to add two integers.
2. Overload the add function to add two floating-point numbers.
3. Overload the add function to concatenate two strings.
4. Call the add function with different types of arguments.

**Source Code:**

```
#include <iostream>
#include <string>
using namespace std;

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Function to add two floating-point numbers
double add(double a, double b) {
    return a + b;
}

// Function to concatenate two strings
string add(const string& a, const string& b) {
    return a + b;
}

int main() {
    int sum1 = add(5, 10);
    double sum2 = add(2.5, 3.7);
    string strSum = add("Hello, ", "World!");

    cout << "Sum of integers: " << sum1 << endl;
    cout << "Sum of floats: " << sum2 << endl;
    cout << "Concatenated string: " << strSum << endl;
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Sum of integers: 15
Sum of floats: 6.2
Concatenated string: Hello, World!
```

## Lab 6: Operator Overloading

**Title:** Operator Overloading

**Aim:** To understand and implement operator overloading in C++.

**Procedure:**

1. Define a class Vector to represent a 2D vector.
2. Overload the + operator to add two Vector objects.
3. Overload the * operator to multiply a Vector object by a scalar.
4. Write a program to demonstrate the use of overloaded operators.

**Source Code:**

```
#include <iostream>
using namespace std;

class Vector {
private:
    double x;
    double y;

public:
    Vector() : x(0), y(0) {}
    Vector(double x_val, double y_val) : x(x_val), y(y_val) {}

    void display() const{
        cout << "(" << x << ", " << y << ")" << endl;
    }

    // Overload the + operator
    Vector operator+(const Vector& other) const{
        Vector result;
        result.x = x + other.x;
        result.y = y + other.y;
        return result;
    }

    // Overload the * operator
    Vector operator*(double scalar) const{
        Vector result;
        result.x = x * scalar;
        result.y = y * scalar;
        return result;
    }
};

int main() {
    Vector v1(1, 2);
    Vector v2(3, 4);
    Vector v3;

    cout << "v1: ";
    v1.display();
    cout << "v2: ";
    v2.display();

    v3 = v1 + v2; // Use overloaded +
    cout << "v1 + v2: ";
    v3.display();
```

```
    v3 = v1 * 2; // Use overloaded *
    cout << "v1 * 2: ";
    v3.display();
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
v1: (1, 2)
v2: (3, 4)
v1 + v2: (4, 6)
v1 * 2: (2, 4)
```

## Lab 7: Inheritance

**Title:** Inheritance

**Aim:** To understand and implement single inheritance in C++.

**Procedure:**

1. Define a base class Shape with a member function to set the color.
2. Define a derived class Circle that inherits from Shape and has a member to set the radius and calculate the area.
3. Create objects of the Circle class and use the inherited and derived class members.

**Source Code:**

```
#include <iostream>
#include <cmath>
using namespace std;

class Shape {
protected:
    string color;

public:
    void setColor(const string& c) {
        color = c;
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    void setRadius(double r) {
        radius = r;
    }

    double calculateArea() const{
        return M_PI * radius * radius;
    }

    void display() const{
         cout << "Color: " << color << endl;
         cout << "Radius: " << radius << endl;
         cout << "Area: " << calculateArea() << endl;
    }
};

int main() {
    Circle c1;
    c1.setColor("Red");
    c1.setRadius(5);

    c1.display();
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Color: Red
Radius: 5
Area: 78.5398
```

## Lab 8: Multiple and Multilevel Inheritance

**Title:** Multiple and Multilevel Inheritance

**Aim:** To understand and implement multiple and multilevel inheritance in C++.

**Procedure:**

1. **Multiple Inheritance:**

   Define two base classes, Base1 and Base2.

   Define a derived class Derived that inherits from both Base1 and Base2.

   Implement member functions in each class to display a message.

   Create an object of the Derived class and call the member functions of the base classes.

2. **Multilevel Inheritance:**

   Define a base class Grandparent.

   Define a derived class Parent that inherits from Grandparent.

   Define a derived class Child that inherits from Parent.

   Implement member functions in each class to display a message.

   Create an object of the Child class and call the member functions of the grandparent and parent classes.

**Source Code:**

```
#include <iostream>
using namespace std;

// Multiple Inheritance
class Base1 {
public:
    void displayBase1() const{
        cout << "Base1 class" << endl;
    }
};

class Base2 {
public:
    void displayBase2() const{
        cout << "Base2 class" << endl;
    }
};

class Derived : public Base1, public Base2 {
public:
    void displayDerived() const{
        cout << "Derived class" << endl;
    }
```

```
};

// Multilevel Inheritance
class Grandparent {
public:
    void displayGrandparent() const{
        cout << "Grandparent class" << endl;
    }
};

class Parent : public Grandparent {
public:
    void displayParent() const{
        cout << "Parent class" << endl;
    }
};

class Child : public Parent {
public:
    void displayChild() const{
        cout << "Child class" << endl;
    }
};

int main() {
    // Multiple Inheritance
    Derived d;
    d.displayBase1();
    d.displayBase2();
    d.displayDerived();

    // Multilevel Inheritance
    Child c;
    c.displayGrandparent();
    c.displayParent();
    c.displayChild();
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Base1 class
Base2 class
Derived class
Grandparent class
Parent class
Child class
```

## Lab 9: Abstract Classes and Virtual Functions

**Title:** Abstract Classes and Virtual Functions

**Aim:** To understand and implement abstract classes and virtual functions in C++.

**Procedure:**

1. Define an abstract class Shape with a pure virtual function calculateArea().
2. Define derived classes Rectangle and Circle that inherit from Shape and implement the calculateArea() function.
3. Create objects of the derived classes using pointers to the base class.
4. Call the calculateArea() function using the base class pointers.

**Source Code:**

```
#include <iostream>
#include <cmath>
using namespace std;

class Shape {
public:
    virtual double calculateArea() const = 0; // Pure virtual function
    virtual void display() const = 0;
protected:
    string color;
};

class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(const string& c, double l, double w): color(c), length(l),
width(w) {}
    double calculateArea() const override {
        return length * width;
    }
     void display() const override{
        cout << "Color: " << color << endl;
        cout << "Length: " << length << endl;
        cout << "Width: " << width << endl;
        cout << "Area: " << calculateArea() << endl;
    }
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(const string& c, double r) : color(c), radius(r) {}
    double calculateArea() const override {
        return M_PI * radius * radius;
    }
    void display() const override{
        cout << "Color: " << color << endl;
        cout << "Radius: " << radius << endl;
        cout << "Area: " << calculateArea() << endl;
    }
};
```

```
int main() {
    Shape* shape1 = new Rectangle("Red", 10, 5);
    Shape* shape2 = new Circle("Blue", 5);

    shape1->display();
    shape2->display();

    delete shape1;
    delete shape2;
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Color: Red
Length: 10
Width: 5
Area: 50
Color: Blue
Radius: 5
Area: 78.5398
```

## Lab 10: Simple File Programs

**Title:** Simple File Programs

**Aim:** To perform basic file operations in C++.

**Procedure:**

1. Write a program to create a file and write data to it.
2. Write a program to read data from an existing file.

**Source Code:**

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    // Write to a file
    ofstream outFile("myFile.txt");
    if (outFile.is_open()) {
        outFile << "Hello, world!" << endl;
        outFile << "This is a test." << endl;
        outFile.close();
        cout << "Data written to file." << endl;
    } else {
        cout << "Unable to open file for writing." << endl;
    }

    // Read from a file
    ifstream inFile("myFile.txt");
    string line;
    if (inFile.is_open()) {
        cout << "Reading from file:" << endl;
        while (getline(inFile, line)) {
            cout << line << endl;
        }
        inFile.close();
    } else {
        cout << "Unable to open file for reading." << endl;
    }
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Data written to file.
Reading from file:
Hello, world!
This is a test.
```

## Lab 11: Working with Files

**Title:** Working with Files

**Aim:** To perform more advanced file operations in C++.

**Procedure:**

1. Write a program to append data to an existing file.
2. Write a program to read and write data in binary mode.

**Source Code:**

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    // Append to a file
    ofstream outFile("myFile.txt", ios::app);
    if (outFile.is_open()) {
        outFile << "Appending more data." << endl;
        outFile.close();
        cout << "Data appended to file." << endl;
    } else {
        cout << "Unable to open file for appending." << endl;
    }

    // Read and write in binary mode
    struct Data {
        int id;
        char name[20];
        float salary;
    };

    Data myData = {123, "John Doe", 50000.0f};
    //Write to binary file
    ofstream outBinFile("myBinFile.dat", ios::binary);
     if (outBinFile.is_open()) {
        outBinFile.write(reinterpret_cast<char*>(&myData), sizeof(myData));
        outBinFile.close();
        cout << "Binary data written to file" << endl;
     }
     else{
        cout << "Unable to open binary file for writing" << endl;
     }

    //Read from binary file
    ifstream inBinFile("myBinFile.dat", ios::binary);
    Data readData;
    if(inBinFile.is_open()){
        inBinFile.read(reinterpret_cast<char*>(&readData), sizeof(readData));
        inBinFile.close();

        cout << "Reading binary data from file: " << endl;
        cout << "ID: " << readData.id << endl;
        cout << "Name: " << readData.name << endl;
        cout << "Salary: " << readData.salary << endl;
    }
    else{
        cout << "Unable to open binary file for reading" << endl;
```

```
    }
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Data appended to file.
Binary data written to file
Reading binary data from file:
ID: 123
Name: John Doe
Salary: 50000
```

## Lab 12: Command Line Arguments Program

**Title:** Command Line Arguments

**Aim:** To understand and use command-line arguments in C++.

**Procedure:**

1. Write a program that takes two numbers as command-line arguments.
2. Convert the command-line arguments to integers.
3. Calculate the sum of the two numbers.
4. Print the sum.

**Source Code:**

```
#include <iostream>
#include <cstdlib> // For atoi()
using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cout << "Usage: " << argv[0] << " num1 num2" << endl;
        return 1;
    }

    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);
    int sum = num1 + num2;

    cout << "Sum of " << num1 << " and " << num2 << " is " << sum << endl;
    return 0;
}
```

**Input:**

```
// Compile the program (e.g., g++ myProgram.cpp -o myProgram)
// Run the program from the command line:
// ./myProgram 10 20
```

**Expected Output:**

```
Sum of 10 and 20 is 30
```

## Lab 13: Templates

**Title:** Templates

**Aim:** To understand and implement generic programming using templates in C++.

**Procedure:**

1. Write a template function to find the maximum of two values.
2. Write a template class to represent a generic array.
3. Create instances of the template function and class with different data types.

**Source Code:**

```
#include <iostream>
using namespace std;

// Template function to find the maximum of two values
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a :b;
}

// Template class to represent a generic array
template <typename T, int size>
class GenericArray {
private:
    T arr[size];
public:
    GenericArray() {
        for (int i = 0; i < size; i++) {
            arr[i] = T(); // Initialize with default value
        }
    }

    void setElement(int index, T value) {
        if (index >= 0 && index < size) {
            arr[index] = value;
        }
    }

    T getElement(int index) const{
       if (index >= 0 && index < size) {
            return arr[index];
        }
        return T();
    }

    void display() const{
        cout << "Array: ";
        for (int i = 0; i < size; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    // Template function
    int maxInt = maximum(5, 10);
    double maxDouble = maximum(2.5, 3.7);
```

```
    string maxString = maximum(string("apple"), string("banana"));

    cout << "Max of integers: " << maxInt << endl;
    cout << "Max of doubles: " << maxDouble << endl;
    cout << "Max of strings: " << maxString << endl;

    // Template class
    GenericArray<int, 5> intArray;
    intArray.setElement(0, 1);
    intArray.setElement(1, 2);
    intArray.setElement(2, 3);
    intArray.display();

    GenericArray<string, 3> strArray;
    strArray.setElement(0, "one");
    strArray.setElement(1, "two");
    strArray.setElement(2, "three");
    strArray.display();
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Max of integers: 10
Max of doubles: 3.7
Max of strings: banana
Array: 1 2 3 0 0
Array: one two three
```

## Lab 14: Multilevel Exception Programs

**Title:** Multilevel Exception Handling

**Aim:** To understand and implement multilevel exception handling in C++.

**Procedure:**

1. Write a program with nested try blocks.
2. Throw an exception in the innermost try block.
3. Catch the exception in an intermediate or the outermost catch block.
4. Demonstrate how exceptions are propagated in nested try-catch structures.

**Source Code:**

```
#include <iostream>
using namespace std;

int main() {
    try {
        cout << "Outer try block" << endl;
        try {
            cout << "Inner try block" << endl;
            // Throw an exception
            throw 10;
            cout << "This line will not be executed" << endl; //won't execute
        }
        catch (double) {
            cout << "Caught double exception" << endl;
        }
        catch (int n) {
            cout << "Caught int exception: " << n << endl;
        }
        cout << "This line will be executed after inner catch" << endl;
    }
    catch (char c) {
        cout << "Caught char exception: " << c << endl;
    }
    cout << "End of program" << endl;
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Outer try block
Inner try block
Caught int exception: 10
This line will be executed after inner catch
End of program
```

## Lab 15: User-Defined Exceptions and Simple CPP Application

**Title:** User-Defined Exceptions and Simple CPP Application

**Aim:** To define and use user-defined exceptions, and to create a simple C++ application.

**Procedure:**

1. Define a user-defined exception class.
2. Write a program that uses the user-defined exception.
3. Create a simple C++ application that demonstrates the use of classes, objects, and exception handling. For example, a simple banking application.

**Source Code:**

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

// User-defined exception class
class InsufficientFundsException : public runtime_error {
public:
    InsufficientFundsException(const string& message) :
runtime_error(message) {}
};

class BankAccount {
private:
    string accountNumber;
    double balance;

public:
    BankAccount(const string& accNum, double bal) : accountNumber(accNum),
balance(bal) {}

    string getAccountNumber() const{
        return accountNumber;
    }

    double getBalance() const{
        return balance;
    }

    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            cout << "Deposited " << amount << ". New balance: " << balance <<
endl;
        } else {
            cout << "Invalid deposit amount." << endl;
        }
    }

    void withdraw(double amount) {
        if (amount > 0) {
            if (balance >= amount) {
                balance -= amount;
                cout << "Withdrew " << amount << ". New balance: " << balance
<< endl;
            } else {
```

```
                throw InsufficientFundsException("Insufficient funds to
withdraw " + to_string(amount));
            }
        } else {
            cout << "Invalid withdrawal amount." << endl;
        }
    }
};

int main() {
    BankAccount account("1234567890", 1000);

    cout << "Account Number: " << account.getAccountNumber() << endl;
    cout << "Initial Balance: " << account.getBalance() << endl;

    try {
        account.deposit(500);
        account.withdraw(200);
        account.withdraw(1500); // This will throw an exception
    }
    catch (const InsufficientFundsException& e) {
        cerr << "Exception: " << e.what() << endl;
    }

    cout << "Final Balance: " << account.getBalance() << endl;
    return 0;
}
```

**Input:** (None)

**Expected Output:**

```
Account Number: 1234567890
Initial Balance: 1000
Deposited 500. New balance: 1500
Withdrew 200. New balance: 1300
Exception: Insufficient funds to withdraw 1500
      Final Balance: 1300
```