SRM Institute of Science and Technology

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

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 2nd Sem

OBJECT ORIENTED PROGRAMMING (USA23201J)- Lab Manual

This manual outlines the experiments for the Object Oriented Programming course. Each lab aims to provide hands-on experience with fundamental OOP concepts in C++.

Lab 1: I/O operations and operators

Title

Basic Input/Output Operations and Operators

Aim

To understand and implement basic input/output operations and various arithmetic, relational, logical, and bitwise operators in C++.

Procedure

1. **Include necessary headers:** Start by including `<iostream>` for input/output operations.
2. **Declare variables:** Declare variables of different data types (e.g., `int`, `float`, `char`).
3. **Input:** Use `cin` to get input from the user for at least two numeric variables.
4. **Perform operations:**
    - Apply arithmetic operators (`+`, `-`, `*`, `/`, `%`) and display results.
    - Apply relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) and display boolean results.
    - Apply logical operators (`&&`, `||`, `!`) and display boolean results.
    - (Optional) Apply bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`) and display results.
5. **Output:** Use `cout` to display prompts, input values, and the results of all operations with clear labels.

Source Code

```
// Example C++ code for Lab 1
#include <iostream>

int main() {
    int num1, num2;

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

    // Arithmetic Operations
    std::cout << "\n--- Arithmetic Operations ---" << std::endl;
    std::cout << "Sum: " << num1 + num2 << std::endl;
    // Add more arithmetic operations here

    // Relational Operations
    std::cout << "\n--- Relational Operations ---" << std::endl;
    std::cout << "num1 == num2: " << (num1 == num2) << std::endl;
    // Add more relational operations here

    // Logical Operations (example with boolean values or expressions)
    std::cout << "\n--- Logical Operations ---" << std::endl;
    bool condition1 = (num1 > 0);
    bool condition2 = (num2 < 100);
    std::cout << "num1 > 0 && num2 < 100: " << (condition1 && condition2) <<
std::endl;
    // Add more logical operations here

    return 0;
}
```

Input

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

Expected Output

```
--- Arithmetic Operations ---
Sum: 15
Difference: 5
Product: 50
Quotient: 2
Modulo: 0

--- Relational Operations ---
num1 == num2: 0
num1 != num2: 1
num1 > num2: 1
num1 < num2: 0
num1 >= num2: 1
num1 <= num2: 0

--- Logical Operations ---
num1 > 0 && num2 < 100: 1
num1 > 0 || num2 < 100: 1
!(num1 > 0): 0
```

Lab 2: Control structures and Functions

Title

Implementing Control Structures and User-Defined Functions

Aim

To implement and understand the usage of conditional statements (if-else, switch), looping constructs (for, while, do-while), and user-defined functions in C++.

Procedure

1. **Conditional Statements:**
   - Write a program that takes an integer input and uses an `if-else if-else` ladder to determine if it's positive, negative, or zero.
   - Write a program that takes a character input and uses a `switch` statement to check if it's a vowel or a consonant.
2. **Looping Constructs:**
   - Use a `for` loop to print numbers from 1 to 10.
   - Use a `while` loop to calculate the sum of digits of a given number.
   - Use a `do-while` loop to prompt the user for input until a specific condition is met (e.g., input is 'q').
3. **User-Defined Functions:**
   - Define a function `calculateFactorial(int n)` that takes an integer and returns its factorial.
   - Define a function `isPrime(int n)` that takes an integer and returns `true` if it's prime, `false` otherwise.
   - Call these functions from the `main` function and display their results.

Source Code

```
// Example C++ code for Lab 2
#include <iostream>

// Function to calculate factorial
long long calculateFactorial(int n) {
    long long fact = 1;
    for (int i = 1; i <= n; ++i) {
        fact *= i;
    }
    return fact;
}

// Function to check if a number is prime
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) return false;
    }
    return true;
}

int main() {
    // If-else example
    int num;
    std::cout << "Enter an integer for if-else check: ";
    std::cin >> num;
    if (num > 0) {
        std::cout << num << " is positive." << std::endl;
    } else if (num < 0) {
        std::cout << num << " is negative." << std::endl;
    } else {
        std::cout << num << " is zero." << std::endl;
    }

    // Switch example
    char ch;
    std::cout << "Enter a character for switch check: ";
    std::cin >> ch;
    switch (ch) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
        case 'A': case 'E': case 'I': case 'O': case 'U':
            std::cout << ch << " is a vowel." << std::endl;
            break;
        default:
            std::cout << ch << " is a consonant." << std::endl;
            break;
    }

    // For loop example
    std::cout << "\nNumbers from 1 to 10 using for loop: ";
    for (int i = 1; i <= 10; ++i) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // While loop example (sum of digits)
    int n_while, sum_digits = 0;
    std::cout << "Enter a number to find sum of its digits: ";
    std::cin >> n_while;
    int temp_n = n_while;
    while (temp_n > 0) {
        sum_digits += temp_n % 10;
        temp_n /= 10;
    }
    std::cout << "Sum of digits of " << n_while << " is: " << sum_digits <<
std::endl;

    // Do-while loop example
    char choice;
    do {
        std::cout << "Enter 'q' to quit: ";
        std::cin >> choice;
    } while (choice != 'q' && choice != 'Q');
    std::cout << "Exited do-while loop." << std::endl;

    // Function calls
    int fact_num = 5;
    std::cout << "Factorial of " << fact_num << " is: " <<
calculateFactorial(fact_num) << std::endl;

    int prime_num = 7;
    std::cout << prime_num << (isPrime(prime_num) ? " is prime." : " is not
prime.") << std::endl;

    return 0;
}
```

Input

```
Enter an integer for if-else check: -5
Enter a character for switch check: B
Enter a number to find sum of its digits: 123
Enter 'q' to quit: a
Enter 'q' to quit: q
```

Expected Output

```
-5 is negative.
B is a consonant.

Numbers from 1 to 10 using for loop: 1 2 3 4 5 6 7 8 9 10
Sum of digits of 123 is: 6
Enter 'q' to quit: Enter 'q' to quit: Exited do-while loop.
Factorial of 5 is: 120
7 is prime.
```

Lab 3: Classes and Objects

Title

Introduction to Classes and Objects

Aim

To understand and implement the fundamental concepts of classes and objects in C++, including data members, member functions, and access specifiers.

Procedure

1. **Define a Class:** Create a class named `Student` with the following private data members: `name` (string), `rollNumber` (int), and `grade` (char).
2. **Member Functions:**
   - Declare public member functions:
     - `void setDetails(std::string n, int rn, char g)`: To set the student's details.
     - `void displayDetails()`: To display the student's details.
3. **Create Objects:** In the `main` function, create at least two objects of the `Student` class.
4. **Access Members:**
   - Use the `setDetails` function to assign values to the data members of each object.
   - Use the `displayDetails` function to print the details of each student object.

Source Code

```
// Example C++ code for Lab 3
#include <iostream>
#include <string>

class Student {
private:
    std::string name;
    int rollNumber;
    char grade;

public:
    // Function to set student details
    void setDetails(std::string n, int rn, char g) {
        name = n;
        rollNumber = rn;
        grade = g;
    }

    // Function to display student details
    void displayDetails() {
        std::cout << "Name: " << name << ", Roll No: " << rollNumber << ",
Grade: " << grade << std::endl;
    }
};

int main() {
    // Create objects of Student class
    Student student1;
    Student student2;

    // Set details for student1
    student1.setDetails("Alice", 101, 'A');
    // Set details for student2
    student2.setDetails("Bob", 102, 'B');

    // Display details for student1
    std::cout << "Student 1 Details:" << std::endl;
    student1.displayDetails();

    // Display details for student2
    std::cout << "Student 2 Details:" << std::endl;
    student2.displayDetails();

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
Student 1 Details:
Name: Alice, Roll No: 101, Grade: A
Student 2 Details:
Name: Bob, Roll No: 102, Grade: B
```

Lab 4: Parameterized Constructor and Constructor Overloading

Title

Parameterized Constructors and Constructor Overloading

Aim

To understand and implement parameterized constructors and demonstrate constructor overloading in C++ classes.

Procedure

1. **Define a Class:** Create a class named `Rectangle` with private data members `length` (double) and `width` (double).
2. **Default Constructor:** Define a default constructor that initializes `length` and `width` to 0.0.
3. **Parameterized Constructor:** Define a parameterized constructor `Rectangle(double l, double w)` that initializes `length` and `width` with provided values.
4. **Constructor Overloading:** Define another parameterized constructor `Rectangle(double side)` that initializes both `length` and `width` to `side` (for a square).
5. **Member Function:** Add a public member function `double calculateArea()` that returns the area of the rectangle.
6. **Create Objects:** In `main`, create objects using each of the defined constructors.
7. **Display Results:** Call `calculateArea()` for each object and display their areas, indicating which constructor was used.

Source Code

```
// Example C++ code for Lab 4
#include <iostream>

class Rectangle {
private:
    double length;
    double width;

public:
    // Default constructor
    Rectangle() : length(0.0), width(0.0) {
        std::cout << "Default constructor called." << std::endl;
    }

    // Parameterized constructor
    Rectangle(double l, double w) : length(l), width(w) {
        std::cout << "Parameterized constructor (length, width) called." <<
std::endl;
    }

    // Constructor overloading (for a square)
    Rectangle(double side) : length(side), width(side) {
        std::cout << "Parameterized constructor (side) called for a square."
<< std::endl;
    }

    // Member function to calculate area
    double calculateArea() {
        return length * width;
    }
};

int main() {
    // Create object using default constructor
    Rectangle rect1;
    std::cout << "Area of rect1: " << rect1.calculateArea() << std::endl <<
std::endl;

    // Create object using parameterized constructor (length, width)
    Rectangle rect2(5.0, 3.0);
    std::cout << "Area of rect2: " << rect2.calculateArea() << std::endl <<
std::endl;

    // Create object using overloaded constructor (side)
    Rectangle rect3(4.0); // This creates a square
    std::cout << "Area of rect3: " << rect3.calculateArea() << std::endl <<
std::endl;

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
Default constructor called.
Area of rect1: 0

Parameterized constructor (length, width) called.
Area of rect2: 15

Parameterized constructor (side) called for a square.
Area of rect3: 16
```

Lab 5: Function Overloading

Title

Demonstrating Function Overloading

Aim

To understand and implement function overloading, where multiple functions share the same name but differ in their parameter lists (number, type, or order of parameters).

Procedure

1. **Define a Class (Optional but good practice):** Create a class named `Calculator` (or simply use global functions).
2. **Overload a Function:** Define multiple functions named `add` (or any other meaningful name) within the class or globally, each performing addition but accepting different types or numbers of arguments:
   - `int add(int a, int b)`: Adds two integers.
   - `double add(double a, double b)`: Adds two doubles.
   - `int add(int a, int b, int c)`: Adds three integers.
3. **Call Overloaded Functions:** In `main`, call each of the overloaded `add` functions with appropriate arguments.
4. **Display Results:** Print the results of each function call, clearly indicating which version of the function was invoked.

Source Code

```
// Example C++ code for Lab 5
#include <iostream>

class Calculator {
public:
    // Overloaded function: add two integers
    int add(int a, int b) {
        std::cout << "add(int, int) called." << std::endl;
        return a + b;
    }

    // Overloaded function: add two doubles
    double add(double a, double b) {
        std::cout << "add(double, double) called." << std::endl;
        return a + b;
    }

    // Overloaded function: add three integers
    int add(int a, int b, int c) {
        std::cout << "add(int, int, int) called." << std::endl;
        return a + b + c;
    }
};

int main() {
    Calculator calc;

    // Call the int version of add
    std::cout << "Sum of 5 and 10: " << calc.add(5, 10) << std::endl <<
std::endl;
```

```
    // Call the double version of add
    std::cout << "Sum of 5.5 and 10.3: " << calc.add(5.5, 10.3) << std::endl
<< std::endl;

    // Call the three-int version of add
    std::cout << "Sum of 1, 2, and 3: " << calc.add(1, 2, 3) << std::endl <<
std::endl;

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
add(int, int) called.
Sum of 5 and 10: 15

add(double, double) called.
Sum of 5.5 and 10.3: 15.8

add(int, int, int) called.
Sum of 1, 2, and 3: 6
```

Lab 6: Operator Overloading

Title

Implementing Operator Overloading

Aim

To understand and implement operator overloading in C++ to enable operators to work with user-defined data types (objects).

Procedure

1. **Define a Class:** Create a class named `Complex` to represent complex numbers, with private data members `real` (double) and `imag` (double).
2. **Constructor:** Add a constructor `Complex(double r = 0.0, double i = 0.0)` to initialize complex numbers.
3. **Overload an Operator:** Overload the `+` operator to add two `Complex` objects. This can be done as a member function or a friend function.
   - **Member function approach:** `Complex operator+(const Complex& other)`
   - **Friend function approach:** `friend Complex operator+(const Complex& c1, const Complex& c2)`
4. **Display Function:** Add a public member function `void display()` to print the complex number in the format `real + imag i`.
5. **Create Objects:** In `main`, create at least two `Complex` objects.
6. **Use Overloaded Operator:** Use the overloaded `+` operator to add the objects and store the result in a new `Complex` object.
7. **Display Result:** Display the original complex numbers and the resultant complex number using the `display()` function.

Source Code

```
// Example C++ code for Lab 6
#include <iostream>

class Complex {
private:
    double real;
    double imag;

public:
    // Constructor
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    // Overload the + operator as a member function
    Complex operator+(const Complex& other) {
        Complex temp;
        temp.real = real + other.real;
        temp.imag = imag + other.imag;
        return temp;
    }

    // Function to display the complex number
    void display() {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};
```

```
int main() {
    Complex c1(3.0, 4.0); // 3 + 4i
    Complex c2(1.5, 2.5); // 1.5 + 2.5i
    Complex c3;          // To store the result

    std::cout << "Complex Number 1: ";
    c1.display();

    std::cout << "Complex Number 2: ";
    c2.display();

    // Use the overloaded + operator
    c3 = c1 + c2; // This calls c1.operator+(c2)

    std::cout << "Sum of Complex Numbers: ";
    c3.display();

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
Complex Number 1: 3 + 4i
Complex Number 2: 1.5 + 2.5i
Sum of Complex Numbers: 4.5 + 6.5i
```

Lab 7: Inheritance

Title

Implementing Single Inheritance

Aim

To understand and implement single inheritance, demonstrating how a derived class can inherit properties and behaviors from a base class.

Procedure

1. **Define a Base Class:** Create a base class named `Animal` with:
    - A protected data member `name` (string).
    - A public member function `void eat()` that prints "Animal is eating."
    - A public member function `void setName(std::string n)` to set the animal's name.
2. **Define a Derived Class:** Create a derived class named `Dog` that publicly inherits from `Animal`.
    - Add a public member function `void bark()` that prints "Dog is barking."
3. **Create Objects:** In `main`, create an object of the `Dog` class.
4. **Access Members:**
    - Use the `setName()` function (inherited from `Animal`) to set the dog's name.
    - Call the `eat()` function (inherited from `Animal`).
    - Call the `bark()` function (specific to `Dog`).
5. **Display Output:** Observe how the `Dog` object can access both its own members and the inherited members from `Animal`.

Source Code

```
// Example C++ code for Lab 7
#include <iostream>
#include <string>

// Base class
class Animal {
protected:
    std::string name;

public:
    void setName(std::string n) {
        name = n;
    }

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }
};

// Derived class
class Dog : public Animal {
public:
    void bark() {
        std::cout << name << " is barking: Woof! Woof!" << std::endl;
    }
};
```

```
int main() {
    // Create an object of the derived class
    Dog myDog;

    // Access inherited member function to set name
    myDog.setName("Buddy");

    // Access inherited member function
    myDog.eat();

    // Access derived class's own member function
    myDog.bark();

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
Buddy is eating.
Buddy is barking: Woof! Woof!
```

Lab 8: Multiple, Multilevel Inheritance

Title

Implementing Multiple and Multilevel Inheritance

Aim

To understand and implement multiple inheritance (a class inheriting from multiple base classes) and multilevel inheritance (a class inheriting from another derived class) in C++.

Procedure

1. **Multilevel Inheritance:**
   - Create a base class `Vehicle` with a method `startEngine()`.
   - Create a derived class `Car` that inherits from `Vehicle` and adds a method `drive()`.
   - Create another derived class `SportsCar` that inherits from `Car` and adds a method `activateTurbo()`.
   - In `main`, create an object of `SportsCar` and call methods from all three levels.
2. **Multiple Inheritance:**
   - Create two base classes `Swimmer` (with `swim()`) and `Walker` (with `walk()`).
   - Create a derived class `Amphibian` that publicly inherits from both `Swimmer` and `Walker`.
   - In `main`, create an object of `Amphibian` and call methods from both base classes.

Source Code

```
// Example C++ code for Lab 8
#include <iostream>
#include <string>

// --- Multilevel Inheritance ---

// Base class for Multilevel
class Vehicle {
public:
    void startEngine() {
        std::cout << "Vehicle engine started." << std::endl;
    }
};

// Derived class 1 (inherits from Vehicle)
class Car : public Vehicle {
public:
    void drive() {
        std::cout << "Car is driving." << std::endl;
    }
};

// Derived class 2 (inherits from Car)
class SportsCar : public Car {
public:
    void activateTurbo() {
        std::cout << "Sports car turbo activated!" << std::endl;
    }
};

// --- Multiple Inheritance ---
// Base class 1 for Multiple
class Swimmer {
public:
    void swim() {
        std::cout << "Creature is swimming." << std::endl;
    }
};

// Base class 2 for Multiple
class Walker {
public:
    void walk() {
        std::cout << "Creature is walking." << std::endl;
    }
};

// Derived class (inherits from Swimmer and Walker)
class Amphibian : public Swimmer, public Walker {
public:
    void displayNature() {
        std::cout << "I am an amphibian, I can do both!" << std::endl;
    }
};


int main() {
    std::cout << "--- Multilevel Inheritance Example ---" << std::endl;
    SportsCar mySportsCar;
    mySportsCar.startEngine();    // From Vehicle
    mySportsCar.drive();         // From Car
    mySportsCar.activateTurbo(); // From SportsCar
    std::cout << std::endl;

    std::cout << "--- Multiple Inheritance Example ---" << std::endl;
    Amphibian frog;
    frog.displayNature();
    frog.swim(); // From Swimmer
    frog.walk(); // From Walker

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
--- Multilevel Inheritance Example ---
Vehicle engine started.
Car is driving.
Sports car turbo activated!

--- Multiple Inheritance Example ---
I am an amphibian, I can do both!
Creature is swimming.
Creature is walking.
```

Lab 9: Abstract classes and Virtual Functions

Title

Abstract Classes and Virtual Functions for Polymorphism

Aim

To understand and implement abstract classes and virtual functions to achieve runtime polymorphism and define an interface for derived classes.

Procedure

1. **Define an Abstract Base Class:** Create an abstract base class named `Shape` with:
   - A pure virtual function `virtual double calculateArea() = 0;`
   - A regular virtual function `virtual void display() { std::cout << "This is a shape." << std::endl; }`
2. **Define Derived Classes:** Create at least two concrete derived classes (e.g., `Circle`, `Rectangle`) that publicly inherit from `Shape`.
   - Each derived class must provide its own implementation for `calculateArea()`.
   - Optionally, override the `display()` function in derived classes to show specific shape details.
3. **Use Pointers/References to Base Class:** In `main`:
   - Create pointers of type `Shape*`.
   - Dynamically allocate objects of `Circle` and `Rectangle` and assign their addresses to the `Shape*` pointers.
4. **Demonstrate Polymorphism:** Call `calculateArea()` and `display()` using the `Shape*` pointers. Observe that the correct derived class's function is called at runtime.
5. **Clean up:** Remember to `delete` dynamically allocated memory.

Source Code

```
// Example C++ code for Lab 9
#include <iostream>
#include <cmath> // For M_PI

// Abstract Base Class
class Shape {
public:
    // Pure virtual function
    virtual double calculateArea() = 0;

    // Virtual function
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    // Virtual destructor is good practice for polymorphic classes
    virtual ~Shape() {}
};

// Derived Class: Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double calculateArea() override {
        return M_PI * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << "." <<
std::endl;
    }
};

// Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

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

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << "." << std::endl;
    }
};

int main() {
    Shape* shapePtr1 = new Circle(5.0);
    Shape* shapePtr2 = new Rectangle(4.0, 6.0);

    std::cout << "--- Using Circle Object ---" << std::endl;
    shapePtr1->display();
    std::cout << "Area: " << shapePtr1->calculateArea() << std::endl <<
std::endl;

    std::cout << "--- Using Rectangle Object ---" << std::endl;
    shapePtr2->display();
    std::cout << "Area: " << shapePtr2->calculateArea() << std::endl <<
std::endl;

    // Clean up dynamically allocated memory
    delete shapePtr1;
    delete shapePtr2;

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
--- Using Circle Object ---
This is a Circle with radius 5.
Area: 78.5398

--- Using Rectangle Object ---
This is a Rectangle with length 4 and width 6.
Area: 24
```

Lab 10: Simple file programs

Title

Basic File Input/Output Operations

Aim

To understand and implement basic file operations in C++, including opening a file for writing, writing data to it, opening a file for reading, and reading data from it.

Procedure

1. **Include Headers:** Include `<fstream>` for file stream operations.
2. **Writing to a File:**
   - Create an `ofstream` object (output file stream).
   - Open a file (e.g., "example.txt") in output mode. Check if the file opened successfully.
   - Write a few lines of text or some data to the file using the << operator.
   - Close the file.
3. **Reading from a File:**
   - Create an `ifstream` object (input file stream).
   - Open the same file ("example.txt") in input mode. Check if the file opened successfully.
   - Read data from the file line by line using `getline()` or word by word using >>.
   - Print the read content to the console.
   - Close the file.

Source Code

```
// Example C++ code for Lab 10
#include <iostream>
#include <fstream> // Required for file operations
#include <string>

int main() {
    // --- Writing to a file ---
    std::ofstream outFile("example.txt"); // Create an output file stream
object

    // Check if the file was opened successfully
    if (outFile.is_open()) {
        outFile << "Hello, this is the first line.\n";
        outFile << "This is the second line with a number: " << 123 <<
std::endl;
        outFile << "End of file writing." << std::endl;
        outFile.close(); // Close the file
        std::cout << "Data written to example.txt successfully." <<
std::endl;
    } else {
        std::cerr << "Error: Unable to open file for writing." << std::endl;
        return 1; // Indicate an error
    }

    // --- Reading from a file ---
    std::ifstream inFile("example.txt"); // Create an input file stream
object
    std::string line;
    // Check if the file was opened successfully
    if (inFile.is_open()) {
        std::cout << "\n--- Content of example.txt ---" << std::endl;
        while (getline(inFile, line)) { // Read line by line until end of
file
            std::cout << line << std::endl;
        }
        inFile.close(); // Close the file
    } else {
        std::cerr << "Error: Unable to open file for reading." << std::endl;
        return 1; // Indicate an error
    }

    return 0;
}
```

Input

(No direct user input for this program, file content is generated by the program)

Expected Output

```
Data written to example.txt successfully.

--- Content of example.txt ---
Hello, this is the first line.
This is the second line with a number: 123
End of file writing.
```

Lab 11: Working with files

Title

Advanced File Operations

Aim

To perform more advanced file operations such as appending data, seeking within a file, and handling file errors in C++.

Procedure

1. **Append to a File:**
   - Open "example.txt" (from Lab 10) in append mode (`std::ios::app`).
   - Write new lines of text to the end of the file.
   - Close the file.
2. **Read and Seek:**
   - Open "example.txt" in input mode (`std::ios::in`).
   - Read some initial content.
   - Use `seekg()` to move the file pointer to a specific position (e.g., 5th character from the beginning, or 10th character from the end).
   - Read content from the new position and display it.
3. **Error Handling:**
   - Attempt to open a non-existent file for reading and demonstrate error handling using `is_open()` or `fail()`.
   - Clear error flags using `clear()` if needed.

Source Code

```
// Example C++ code for Lab 11
#include <iostream>
#include <fstream>
#include <string>

int main() {
    // --- Appending to a file ---
    std::ofstream appendFile("example.txt", std::ios::app); // Open in append
mode

    if (appendFile.is_open()) {
        appendFile << "\n--- Appended Content ---" << std::endl;
        appendFile << "This line was appended later." << std::endl;
        appendFile.close();
        std::cout << "Data appended to example.txt successfully." <<
std::endl;
    } else {
        std::cerr << "Error: Unable to open file for appending." <<
std::endl;
        return 1;
    }

    // --- Reading and Seeking within a file ---
    std::ifstream readFile("example.txt");
    std::string line;

    if (readFile.is_open()) {
        std::cout << "\n--- Reading from example.txt with seekg ---" <<
std::endl;
```

```
        // Read first line normally
        getline(readFile, line);
        std::cout << "First line: " << line << std::endl;

        // Seek to a specific position (e.g., 20 bytes from beginning)
        readFile.seekg(20, std::ios::beg);
        std::cout << "Content after seeking 20 bytes from beginning: ";
        getline(readFile, line);
        std::cout << line << std::endl;

        // Clear EOF flag and seek to the beginning to read again
        readFile.clear();
        readFile.seekg(0, std::ios::beg);
        std::cout << "\nFull content after append:" << std::endl;
        while (getline(readFile, line)) {
            std::cout << line << std::endl;
        }
        readFile.close();
    } else {
        std::cerr << "Error: Unable to open file for reading and seeking." <<
std::endl;
        return 1;
    }

    // --- Error Handling Example ---
    std::ifstream nonExistentFile("non_existent.txt");
    if (!nonExistentFile.is_open()) {
        std::cerr << "\nError: 'non_existent.txt' could not be opened (as
expected)." << std::endl;
        // The failbit is set. You can clear it if you want to reuse the
stream object.
        nonExistentFile.clear();
    }

    return 0;
}
```

Input

(No direct user input for this program, file content is manipulated by the program)

Expected Output

```
Data appended to example.txt successfully.

--- Reading from example.txt with seekg ---
First line: Hello, this is the first line.
Content after seeking 20 bytes from beginning: is the second line with a
number: 123

Full content after append:
Hello, this is the first line.
This is the second line with a number: 123
End of file writing.

--- Appended Content ---
This line was appended later.

Error: 'non_existent.txt' could not be opened (as expected).
```

Lab 12: Command line arguments program

Title

Processing Command Line Arguments

Aim

To understand how to pass and process command line arguments to a C++ program using the `argc` and `argv` parameters of the `main` function.

Procedure

1. **Modify `main` function signature:** Change the `main` function signature to `int main(int argc, char* argv[])`.
    - `argc`: Represents the total number of command line arguments.
    - `argv`: An array of character pointers, where each pointer points to a command line argument string. `argv[0]` is always the program's name.
2. **Display Arguments:** Iterate through the `argv` array from index 0 to `argc - 1` and print each argument.
3. **Perform an operation:** Based on the arguments, perform a simple operation (e.g., if the arguments are numbers, sum them up; if they are strings, concatenate them).
    - For numerical operations, remember to convert string arguments to integers or doubles using `std::stoi`, `std::stod`, or `atoi`/`atof`.
4. **Error Handling:** Check `argc` to ensure the correct number of arguments are provided. If not, print a usage message.

Source Code

```
// Example C++ code for Lab 12
#include <iostream>
#include <string>   // For std::string and std::stoi
#include <cstdlib>  // For atoi (alternative to stoi)

int main(int argc, char* argv[]) {
    std::cout << "--- Command Line Arguments ---" << std::endl;
    std::cout << "Number of arguments: " << argc << std::endl;

    // Print all arguments
    for (int i = 0; i < argc; ++i) {
        std::cout << "Argument " << i << ": " << argv[i] << std::endl;
    }

    // Example: Summing numbers passed as arguments
    if (argc > 2) { // Expecting program name + at least two numbers
        int sum = 0;
        std::cout << "\n--- Summing provided numbers ---" << std::endl;
        for (int i = 1; i < argc; ++i) { // Start from index 1 to skip
program name
            try {
                int num = std::stoi(argv[i]); // Convert string to integer
                sum += num;
                std::cout << "Added " << num << std::endl;
            } catch (const std::invalid_argument& e) {
                std::cerr << "Warning: Argument '" << argv[i] << "' is not a
valid number. Skipping." << std::endl;
            } catch (const std::out_of_range& e) {
                std::cerr << "Warning: Argument '" << argv[i] << "' is out of
integer range. Skipping." << std::endl;
```

```
            }
        }
        std::cout << "Total sum: " << sum << std::endl;
    } else {
        std::cout << "\nUsage: " << argv[0] << " <number1> <number2> [more
numbers...]" << std::endl;
        std::cout << "Please provide at least two numbers as command line
arguments to see their sum." << std::endl;
    }

    return 0;
}
```

Input

To run this program, you would compile it (e.g., `g++ lab12.cpp -o lab12`) and then execute it from the terminal with arguments:

```
./lab12 10 20 30
```

or

```
./lab12 hello world
```

or

```
./lab12
```

Expected Output

For input: `./lab12 10 20 30`

```
--- Command Line Arguments ---
Number of arguments: 4
Argument 0: ./lab12
Argument 1: 10
Argument 2: 20
Argument 3: 30

--- Summing provided numbers ---
Added 10
Added 20
Added 30
Total sum: 60
```

For input: `./lab12 hello world`

```
--- Command Line Arguments ---
Number of arguments: 3
Argument 0: ./lab12
Argument 1: hello
Argument 2: world

--- Summing provided numbers ---
Warning: Argument 'hello' is not a valid number. Skipping.
Warning: Argument 'world' is not a valid number. Skipping.
Total sum: 0
```

For input: `./lab12`

```
--- Command Line Arguments ---
Number of arguments: 1
Argument 0: ./lab12

Usage: ./lab12 <number1> <number2> [more numbers...]
Please provide at least two numbers as command line arguments to see their
sum.
```

Lab 13: Templates

Title

Implementing Function and Class Templates

Aim

To understand and implement function templates and class templates in C++ to write generic code that can work with different data types without code duplication.

Procedure

1. **Function Template:**
   - Create a function template `T findMax(T a, T b)` that takes two arguments of any type `T` and returns the larger of the two.
   - In `main`, call `findMax` with `int`, `double`, and `char` arguments to demonstrate its versatility.
2. **Class Template:**
   - Create a class template `MyPair<T1, T2>` that can store two values of potentially different types `T1` and `T2`.
   - Include a constructor to initialize the pair and a member function `void display()` to print the stored values.
   - In `main`, create objects of `MyPair` with different type combinations (e.g., `MyPair<int, double>`, `MyPair<std::string, char>`).
   - Call the `display()` function for each `MyPair` object.

Source Code

```
// Example C++ code for Lab 13
#include <iostream>
#include <string>

// --- Function Template ---
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

// --- Class Template ---
template <typename T1, typename T2>
class MyPair {
private:
    T1 first;
    T2 second;
public:
    // Constructor
    MyPair(T1 f, T2 s) : first(f), second(s) {}

    // Member function to display the pair
    void display() {
        std::cout << "First value: " << first << ", Second value: " << second
<< std::endl;
    }
};

int main() {
    std::cout << "--- Function Template Examples ---" << std::endl;
    // Using findMax with integers
    std::cout << "Max of 10 and 20: " << findMax(10, 20) << std::endl;

    // Using findMax with doubles
    std::cout << "Max of 10.5 and 8.2: " << findMax(10.5, 8.2) << std::endl;

    // Using findMax with characters
    std::cout << "Max of 'a' and 'z': " << findMax('a', 'z') << std::endl;
    std::cout << std::endl;

    std::cout << "--- Class Template Examples ---" << std::endl;
    // MyPair with int and double
    MyPair<int, double> p1(100, 25.5);
    std::cout << "Pair 1: ";
    p1.display();

    // MyPair with string and char
    MyPair<std::string, char> p2("Hello", 'W');
    std::cout << "Pair 2: ";
    p2.display();

    // MyPair with two integers
    MyPair<int, int> p3(50, 75);
    std::cout << "Pair 3: ";
    p3.display();

    return 0;
}
```

Input

(No direct user input for this program, data is initialized within the code)

Expected Output

```
--- Function Template Examples ---
Max of 10 and 20: 20
Max of 10.5 and 8.2: 10.5
Max of 'a' and 'z': z

--- Class Template Examples ---
Pair 1: First value: 100, Second value: 25.5
Pair 2: First value: Hello, Second value: W
Pair 3: First value: 50, Second value: 75
```

Lab 14: Multilevel exceptional programs

Title

Multilevel Exception Handling

Aim

To understand and implement multilevel exception handling, where exceptions thrown in one function are caught by a `try-catch` block in a calling function (or higher up the call stack).

Procedure

1. **Define Functions:** Create three functions, `funcA`, `funcB`, and `funcC`, such that `funcC` is called by `funcB`, and `funcB` is called by `funcA`.
2. **Throw Exception:** In `funcC`, implement a condition that throws an exception (e.g., `throw "Error: Division by zero!"` if a denominator is 0).
3. **Catch at Different Levels:**
   - Option 1: Add a `try-catch` block in `funcB` to catch the exception from `funcC`.
   - Option 2: Allow `funcB` to propagate the exception and add a `try-catch` block in `funcA` to catch the exception originating from `funcC`.
   - Demonstrate both scenarios or choose one to illustrate the concept clearly. For this lab, let's demonstrate propagation.
4. **Main Function:** In `main`, call `funcA` within a `try-catch` block to catch any exceptions that propagate up.
5. **Display Messages:** Print messages at each function entry/exit and within `catch` blocks to trace the exception flow.

Source Code

```
// Example C++ code for Lab 14
#include <iostream>
#include <string>

// Function that might throw an exception
void funcC(int divisor) {
    std::cout << "  Inside funcC..." << std::endl;
    if (divisor == 0) {
        throw std::string("Exception: Division by zero attempted in funcC!");
    }
    std::cout << "  Result of 100 / " << divisor << " is " << (100 / divisor)
<< std::endl;
    std::cout << "  Exiting funcC normally." << std::endl;
}

// Function that calls funcC and might propagate an exception
void funcB(int val) {
    std::cout << " Inside funcB..." << std::endl;
    // No try-catch here, allowing exception to propagate
    funcC(val);
    std::cout << " Exiting funcB normally." << std::endl;
}

// Function that calls funcB and has a try-catch block
void funcA(int input) {
    std::cout << "Inside funcA..." << std::endl;
    try {
        funcB(input);
    } catch (const std::string& e) {
        std::cerr << "Caught exception in funcA: " << e << std::endl;
    } catch (...) { // Catch-all for any other unexpected exceptions
        std::cerr << "Caught an unknown exception in funcA." << std::endl;
    }
    std::cout << "Exiting funcA." << std::endl;
}

int main() {
    std::cout << "--- Scenario 1: No Exception (divisor is not zero) ---" <<
std::endl;
    funcA(5); // Call funcA with a valid divisor
    std::cout << "\n--- Scenario 2: Exception Occurs (divisor is zero) ---"
<< std::endl;
    funcA(0); // Call funcA with a divisor that causes an exception

    return 0;
}
```

Input

(No direct user input for this program, conditions are hardcoded to trigger exceptions)

Expected Output

```
--- Scenario 1: No Exception (divisor is not zero) ---
Inside funcA...
 Inside funcB...
  Inside funcC...
  Result of 100 / 5 is 20
  Exiting funcC normally.
 Exiting funcB normally.
Exiting funcA.

--- Scenario 2: Exception Occurs (divisor is zero) ---
Inside funcA...
 Inside funcB...
  Inside funcC...
Caught exception in funcA: Exception: Division by zero attempted in funcC!
Exiting funcA.
```

Lab 15: User defined Exceptions and simple CPP application.

Title

User-Defined Exceptions and a Simple C++ Application

Aim

To create and use user-defined exception classes and integrate them into a simple C++ application to handle specific error conditions gracefully.

Procedure

1. **Define a User-Defined Exception Class:**
   - Create a class `InvalidInputException` that inherits from `std::exception`.
   - Override the `what()` method to return a custom error message.
2. **Simple C++ Application:**
   - Create a simple application, e.g., a basic "Age Validator" or "Password Checker".
   - In a function (e.g., `validateAge(int age)`), implement logic to throw your custom `InvalidInputException` if the input is invalid (e.g., age < 0 or age > 120).
3. **Throw and Catch:**
   - In `main`, use a `try-catch` block to call the function that might throw the custom exception.
   - Catch `InvalidInputException` specifically and print its `what()` message.
   - Also, include a generic `catch (const std::exception& e)` for other standard exceptions and a `catch (...)` for unknown exceptions.
4. **User Interaction:** Prompt the user for input and provide feedback based on validation.

Source Code

```
// Example C++ code for Lab 15
#include <iostream>
#include <string>
#include <stdexcept> // Required for std::exception

// --- User-Defined Exception Class ---
class InvalidInputException : public std::exception {
private:
    std::string message;
public:
    // Constructor
    InvalidInputException(const std::string& msg) : message(msg) {}

    // Override the what() method
    const char* what() const noexcept override {
        return message.c_str();
    }
};

// --- Simple C++ Application Function (Age Validator) ---
void validateAge(int age) {
    if (age < 0) {
        throw InvalidInputException("Age cannot be negative!");
    } else if (age > 120) {
        throw InvalidInputException("Age seems too high (max 120 assumed).");
    } else {
        std::cout << "Age " << age << " is valid." << std::endl;
    }
}

int main() {
    int userAge;

    std::cout << "--- Age Validator Application ---" << std::endl;

    // Test Case 1: Valid Age
    std::cout << "Enter your age (e.g., 30): ";
    std::cin >> userAge;
    try {
        validateAge(userAge);
    } catch (const InvalidInputException& e) {
        std::cerr << "Custom Exception Caught: " << e.what() << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Standard Exception Caught: " << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Unknown Exception Caught." << std::endl;
    }
    std::cout << std::endl;

    // Test Case 2: Negative Age (Invalid)
    std::cout << "Enter your age (e.g., -5): ";
    std::cin >> userAge;
    try {
        validateAge(userAge);
    } catch (const InvalidInputException& e) {
        std::cerr << "Custom Exception Caught: " << e.what() << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Standard Exception Caught: " << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Unknown Exception Caught." << std::endl;
    }
    std::cout << std::endl;

    // Test Case 3: Very High Age (Invalid)
    std::cout << "Enter your age (e.g., 150): ";
    std::cin >> userAge;
    try {
        validateAge(userAge);
    } catch (const InvalidInputException& e) {
        std::cerr << "Custom Exception Caught: " << e.what() << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Standard Exception Caught: " << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Unknown Exception Caught." << std::endl;
    }
    std::cout << std::endl;

    return 0;
}
```

Input

```
Enter your age (e.g., 30): 30
Enter your age (e.g., -5): -5
Enter your age (e.g., 150): 150
```

Expected Output

```
--- Age Validator Application ---
Enter your age (e.g., 30): Age 30 is valid.

Enter your age (e.g., -5): Custom Exception Caught: Age cannot be negative!
Enter your age (e.g., 150): Custom Exception Caught: Age seems too high (max
120 assumed).
```