UGC Recognized

PDEU PANDIT DEENDAYAL ENERGY UNIVERSITY

Formerly Pandit Deendayal Petroleum University [PDPU]

Laboratory Manual

24CS204P: Object Oriented Programming Lab

Computer Science and Engineering Department
School of Technology

Name of the Student:

Roll No.:

Branch:

Sem./Year:

Academic Year:

This is to certify that

Mr./Ms. _____ Roll no._____

Exam No. _____ of 3rd Semester Degree course in Computer Science and Engineering has satisfactorily completed his/her term work in Object Oriented Programming Lab (24CS204P) subject during the semester from_____ to _____ at School of Technology, PDEU.

Date of Submission:

Signature:

Faculty In-charge Head of Department

Index

Name:

Roll No:

Exam No:

Sr. No.	Experiment Title	Pages		Date of Completion	Marks (out of 10)	Sign.
		From	To			
1	Basic C++ and I/O					
2	Functions and Control Structures					
3	Classes and Objects					
4	Constructors and Destructors					
5	Operator Overloading					
6	Inheritance					
7	Polymorphism and Virtual Functions					
8	Friend Functions and Static Members					
9	File Handling					
10	Templates and Exception Handling					
11	STL Containers					
12	Unit Testing and Test-Driven Development (TDD)					

Course Outcome:

On completion of the course, student will be able to

CO1 - Apply object-oriented programming principles.

CO2 - Apply appropriate programming constructs for a problem.

CO3 - Illustrate the use of inheritance and polymorphism.

CO4 - Compose exception handling framework.

CO5 - Create a project based on OOP Concept.

CO6 - Demonstrate performance optimization, design patterns and associated best practices through a project.

Experiment 1: Introduction to C++ and I/O

Objective: Set up and get familiar with C++ programming environment. Understand basic input/output operations and program structure in C++. This lab focuses on getting familiar with the fundamental syntax of C++, including how to take input from the user and display output to the console. It also introduces the standard program structure with the main function.

To achieve this, we will primarily use **Code::Blocks**, a popular, free, open-source, and cross-platform IDE.

About Code::Blocks

Code::Blocks is a powerful and flexible IDE designed for C, C++, and Fortran programming. It's particularly well-suited for beginners due to its user-friendly interface and its ability to bundle a compiler (like MinGW GCC for Windows) directly within its installation package.

Key Features of Code::Blocks relevant to this lab:

1. **Integrated Compiler:** Code::Blocks often comes with a GCC compiler (MinGW on Windows, or it can be configured to use existing compilers on Linux/macOS). This means you don't have to install and configure the compiler separately, simplifying the setup process significantly.
2. **Text Editor:** It provides a robust text editor with syntax highlighting, auto-completion, and other features that make writing C++ code easier and more efficient.
3. **Project Management:** You can create and manage projects, which helps organize multiple source files, headers, and other resources for larger programs. For this lab, you'll typically create a simple console application project.
4. **Build System:** Code::Blocks integrates a build system that allows you to compile (build) your source code into an executable program with a single click.
5. **Debugger:** It includes a powerful debugger that helps you find and fix errors (bugs) in your code by allowing you to step through your program line by line, inspect variable values, and set breakpoints.
6. **Cross-Platform:** It runs on Windows, Linux, and macOS, ensuring a consistent development experience regardless of your operating system.

How Code::Blocks Helps in C++:

- **Basic Input/Output Operations:** Once you write your C++ code (e.g., using std::cout for output and std::cin for input), Code::Blocks allows you to compile and run it directly. The output will appear in a console window, and you can provide input there, making it easy to see your program's interaction.
- **Program Structure:** Code::Blocks helps you create new C++ source files (.cpp) within a project. When you start a new "Console Application" project, it often

generates a basic main function boilerplate, guiding you on the standard structure of a C++ program.
- **Fundamental Syntax:** The syntax highlighting in the editor helps you visually distinguish keywords, variables, and comments, aiding in understanding the C++ language structure. Error messages from the integrated compiler are displayed clearly, helping you learn from and correct syntax mistakes.

Problem 1: Write a C++ program to calculate and display the grade of a student based on marks in 5 subjects.

Steps to Perform:

1. Understand the problem statement.
2. Identify input/output requirements.
3. Determine suitable data types and structures.
4. Write pseudocode.
5. Implement using C++.
6. Compile and test the program.
7. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)
- fstream (for file-related tasks)

Sample Test Cases: Input: Marks for 5 subjects (e.g., 85 90 78 92 88)

Output: Student Grade: A (or appropriate grade based on calculation)

Problem 2: Write C++ program to find whether number is even or odd.

Steps to Perform:

1. Understand the problem statement.
2. Identify input/output requirements.
3. Determine suitable data types and structures.
4. Write pseudocode.
5. Implement using C++.
6. Compile and test the program.
7. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)
- fstream (for file-related tasks)

Test Cases: Input: 2

Output: even

Experiment 2: Functions and Control Structures

Objective: Implement functions and use control structures to solve problems. This lab focuses on modularizing code using functions and controlling program flow with conditional statements (if-else, switch) and loops (for, while, do-while). Functions promote reusability and readability, while control structures are essential for decision-making and repetitive tasks.

Problem 1: Write a program in C++ using recursive function to compute factorial and check for prime numbers.

Steps to Perform:

1. Understand the problem statement.

2. Identify input/output requirements.
3. Determine suitable data types and structures.
4. Write pseudocode.
5. Implement using C++.
6. Compile and test the program.
7. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)
- fstream (for file-related tasks)

Sample Test Cases: Input: Factorial of: 5 Check prime for: 7 **Output:** Factorial of 5 is: 120 7 is a prime number.

Problem 2: Simulate Collatz Conjecture for 1 to N and find the number with longest steps in C++.

Theory: The Collatz Conjecture is a famous unsolved problem in mathematics. It proposes that if you start with any positive integer n, and repeatedly apply the following rules, you will eventually reach the number 1:

- If n is even, divide it by 2 (n/2).
- If n is odd, multiply it by 3 and add 1 (3n+1).

For example, starting with n=6: 6→3→10→5→16→8→4→2→1. This sequence took 8 "steps" (or iterations) to reach 1.

Steps to Perform:

1. Understand the problem statement.
2. Identify input/output requirements.
3. Determine suitable data types and structures.
4. Write pseudocode.
5. Implement using C++.
6. Compile and test the program.
7. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)
- fstream (for file-related tasks)

Test Cases: Input: N= 10

Output:

Collatz Simulation up to 10:

Number 1: 0 steps

Number 2: 1 step

Number 3: 7 steps

Number 4: 2 steps

Number 5: 5 steps

Number 6: 8 steps

Number 7: 16 steps

Number 8: 3 steps

Number 9: 19 steps

Number 10: 6 steps

Problem 3: Write a C++ program to perform addition and multiplication of Two Matrices.

Steps to Perform:

8. Understand the problem statement.
9. Identify input/output requirements.
10. Determine suitable data types and structures.
11. Write pseudocode.
12. Implement using C++.
13. Compile and test the program.
14. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)
- fstream (for file-related tasks)

Test Cases

Input:

Enter dimensions for Matrix A (rows columns): 2 3

Enter elements for Matrix A (2x3):

1 2 3

4 5 6

Enter dimensions for Matrix B (rows columns): 2 3

Enter elements for Matrix B (2x3):

7 8 9

10 11 12

Enter dimensions for Matrix C (rows columns): 3 2

Enter elements for Matrix C (3x2):

1 2 3

4 5 6

Matrix Addition (A + B):

8 10 12

14 16 18

Matrix Multiplication (A * C):

22 28

49 64

Experiment 3: Introduction to Object-Oriented Programming and Encapsulation

Objective: Implement a C++ program using a class that represents a simple real-world entity (Student). This lab introduces foundational Object-Oriented Programming (OOP) concepts — classes and objects — and demonstrates **encapsulation** through access specifiers (private, public). Students will learn how to define and use member variables and member functions to encapsulate data and behavior securely.

Concept Explanation:

Object-Oriented Programming (OOP) is a programming paradigm that models software design around real-world entities. Key principles include:

- **Class**: A blueprint for creating objects, encapsulating data for the object and methods to manipulate that data.
- **Object**: An instance of a class.

- **Encapsulation**: Bundling of data and functions into a single unit (class) and restricting access to some of the object's components using access specifiers.

Access Specifiers:

- private: Members accessible only within the class.
- public: Members accessible from outside the class (used to control access).

Problem 1: Create a class Student with the following:

- Private members: name, rollNumber, marks
- Public methods:
  - setDetails() – to set values
  - displayDetails() – to show student information

Ensure that direct access to data is restricted using private, and interaction occurs only via public methods.

Steps to Perform:

1. Understand the problem statement.
2. Identify input/output requirements.
3. Determine suitable data types and structures.
4. Write pseudocode.
5. Implement using C++.
6. Compile and test the program.
7. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)
- fstream (for file-related tasks)

Sample Test Cases:

Input:

Enter Name: Alice
Enter Roll Number: 101
Enter Marks: 89.5

Output:

Student Details:
Name: Alice
Roll Number: 101
Marks: 89.5

Problem 2: Implement a Polynomial class with methods to add and evaluate polynomials.

Theory: This problem requires you to design and implement a Polynomial class in C++. A polynomial is a mathematical expression consisting of variables and coefficients, involving only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables (e.g., $3x^2+2x-5$).

Create a class Polynomial that includes:

- Data members to store terms (coefficient and exponent)
- A method to add two polynomials
- A method to evaluate the polynomial at a given x

Steps to Perform:

1. Understand the problem statement.
2. Identify input/output requirements.
3. Determine suitable data types and structures.
4. Write pseudocode.
5. Implement using C++.
6. Compile and test the program.
7. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)
- fstream (for file-related tasks)

Test Cases: Input: Polynomial 1: $3x^2 + 2x - 5$ Polynomial 2: $x^2 – x + 7$ Evaluate sum at $x=2$

Output:

Polynomial 1: 3x^2 + 2x^1 - 5x^0
Polynomial 2: 1x^2 - 1x^1 + 7x^0
Sum of Polynomials: 4x^2 + 1x^1 + 2x^0
Evaluation of Sum at x=2: 20

Experiment 4: Constructors and Destructors

Objective: Use constructors and destructors to initialize and destroy objects. This lab focuses on the special member functions in C++ classes: constructors, which are automatically called when an object is created to initialize its state, and destructors, which are called when an object is destroyed to perform cleanup operations like releasing dynamically allocated memory.

Problem 1: Write a C++ program to create a Rectangle class.

1. The class should have private data members: length and width (both of type double).
2. Implement a **parameterized constructor** Rectangle(double l, double w) that initializes the length and width of the rectangle.
3. Implement a public method calculateArea() that computes and returns the area of the rectangle (length * width).
4. Implement a public method displayDimensions() that prints the length and width of the rectangle.
5. In the main() function, create at least two Rectangle objects using the parameterized constructor with different dimensions, then display their dimensions and calculated areas.

Steps to Perform:

1. Understand the problem statement.
2. Identify input/output requirements.
3. Determine suitable data types and structures.
4. Write pseudocode.
5. Implement using C++.
6. Compile and test the program.
7. Verify the output.

Libraries/Functions Required:

- iostream
- cmath
- string
- vector (if needed)

- fstream (for file-related tasks)

Sample Test Cases: Input: Rectangle dimensions: 10 5 **Output:** Area of Rectangle: 50

Problem 2: **Create a DynamicArray class with dynamic allocation and a destructor.**

Objective: To understand the fundamental principles of dynamic memory allocation using new[] and delete[] for a simple one-dimensional array, and to correctly implement a destructor to release this allocated memory, thereby preventing memory leaks.

Explanation:

In C++, when you declare an array like int arr[10];, its size is fixed at compile time. However, often you need arrays whose size is determined during the program's execution (e.g., based on user input). This is where **dynamic memory allocation** comes in.

- **new[] operator:** This operator is used to allocate a block of memory for an array of a specified type and size on the heap (dynamic memory store). For example, int* arr = new int[size]; allocates memory for size integers and returns a pointer to the first element.
- **delete[] operator:** This operator is used to deallocate memory that was previously allocated with new[]. It is crucial to use delete[] (with the square brackets) when deallocating an array to ensure all elements are properly destructed and the entire block of memory is returned to the system. Failing to use delete[] for memory allocated with new[] (or delete for memory allocated with new) leads to **memory leaks**.

A **destructor** is a special member function (~ClassName()) that is automatically called when an object goes out of scope or is explicitly deleted. For classes that manage dynamically allocated resources (like our DynamicArray class, which holds a pointer to dynamically allocated memory), the destructor is vital. Its sole purpose is to perform cleanup operations, primarily releasing the memory allocated by new[] using delete[], thus preventing memory leaks.

This problem provides a straightforward scenario to practice allocating and deallocating memory for a 1D array within a class, highlighting the essential role of the destructor in responsible memory management.

DynamicArray class

1. The class should represent a one-dimensional array of integers.
2. Use **dynamic memory allocation** (new int[size]) within the constructor to allocate memory for the array elements based on a size provided during object creation.
3. Implement a **destructor** (~DynamicArray()) that correctly deallocates the dynamically allocated memory using delete[] to prevent memory leaks.

4. Include public methods to:
    - Set an element at a specific index (setElement(int index, int value)).
    - Get an element at a specific index (getElement(int index)).
    - Display all elements of the array (displayArray()).

Steps to Perform:

1. Define a DynamicArray class with private members: int* arr (the pointer to the dynamically allocated array) and int size.
2. Implement a parameterized constructor DynamicArray(int s):
    - Initialize size.
    - Dynamically allocate arr using new int[size].
    - (Optional) Initialize all elements to 0.
3. Implement the destructor ~DynamicArray():
    - Print a message indicating the destructor is called.
    - Use delete[] arr; to deallocate the memory.
    - Set arr = nullptr; to avoid dangling pointers (good practice).
4. Add public methods setElement, getElement, and displayArray. Include basic bounds checking for setElement and getElement.
5. In main(), create DynamicArray objects of different sizes, set some elements, display them, and observe the destructor calls when the objects go out of scope.

Libraries/Functions Required:

- iostream for input/output.

Sample Test Cases:

Input: (No direct user input for array elements; sizes are hardcoded in main())

C++

```
// In main():
DynamicArray arr1(5); // Create a dynamic array of size 5
arr1.setElement(0, 10);
arr1.setElement(1, 20);
arr1.setElement(4, 50);

DynamicArray arr2(3); // Create a dynamic array of size 3
arr2.setElement(0, 100);
arr2.setElement(2, 300);
```

Output:

```
DynamicArray constructor called for size 5.
DynamicArray constructor called for size 3.
```

Array 1:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 0 // Default initialized or not set
Element at index 3: 0 // Default initialized or not set
Element at index 4: 50

Array 2:
Element at index 0: 100
Element at index 1: 0 // Default initialized or not set
Element at index 2: 300

DynamicArray destructor called for size 3.
DynamicArray destructor called for size 5.

Problem 3: C++ Constructor Delegation with *this* Pointer

Objective: To understand constructor overloading and how to delegate construction from one constructor to another within the same class using the this pointer. This promotes code reusability within a class's constructors.

Explanation: In C++, a class can have multiple constructors, provided they have different parameter lists (this is known as **constructor overloading**). Sometimes, these constructors might share common initialization logic. C++11 introduced **constructor delegation** using the this pointer. This allows one constructor to call another constructor of the *same class* to perform part or all of its initialization. When you use : this(...) in a constructor's initializer list, it means "call this other constructor before executing my body." The this pointer, in this context, refers to the current object being constructed. This technique helps in writing cleaner and more maintainable code by avoiding redundant initialization logic.

Problem Statement: Write a C++ program to create a Product class. Implement two constructors:

1. A default constructor that initializes product id to 0 and name to "Unknown".
2. A parameterized constructor that takes id and name as arguments. Demonstrate how the default constructor can call the parameterized constructor using the this pointer to set initial values. Include a method to display product details.

Steps to Perform:

1. Define a Product class with id (int) and name (string) as private members.
2. Implement a parameterized constructor Product(int id, std::string name).

3. Implement a default constructor Product() that delegates to the parameterized constructor using this(0, "Unknown").
4. Add a public method displayProduct() to print the product's details.
5. In main(), create objects using both constructors and display their details.

Libraries/Functions Required:

- iostream for input/output.
- string for string manipulation.

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
```
// In main():
Product p1;             // Uses default constructor, delegates
Product p2(101, "Laptop"); // Uses parameterized constructor
```

Output:

```
Default constructor called, delegating to parameterized.
Parameterized constructor called: ID=0, Name=Unknown
Product ID: 0, Name: Unknown
Parameterized constructor called: ID=101, Name: Laptop
Product ID: 101, Name: Laptop
```

Experiment 5: Operator Overloading

Objective: Overload operators in C++ classes. This lab explores operator overloading, a feature in C++ that allows operators (like +, -, *, /, ==, etc.) to be redefined for user-defined types (classes). This enables more intuitive and natural syntax when working with objects.

Problem 1: Overload + operator in Complex class to add two complex numbers.

For a Complex number class, which typically has a real part and an imaginary part (e.g., a+bi), adding two complex numbers means adding their real parts together and their imaginary parts together: (a+bi)+(c+di)=(a+c)+(b+d)i.

By overloading the + operator, you can write Complex c3 = c1 + c2; instead of Complex c3 = c1.add(c2);, which is much closer to how you would write mathematical expressions.

When overloading a binary operator (like +), you typically implement it as a **member function** of the class or as a **friend function**. For arithmetic operators that return a new object (like addition), a member function is often a good choice. The operator function will take one Complex object as an argument (the right-hand operand) and operate on the current object (the left-hand operand, accessed via this). It then returns a new Complex object representing the sum.

Complex class.

1. The class should have private data members: real and imag (both of type double) to represent the real and imaginary parts of a complex number.
2. Implement a parameterized constructor Complex(double r, double i) to initialize the real and imaginary parts.
3. **Overload the + operator** as a member function to perform the addition of two Complex numbers. The overloaded operator should return a new Complex object representing the sum.
4. Implement a public method displayComplex() to print the complex number in the format a + bi.
5. In the main() function, create two Complex objects, add them using the overloaded + operator, and display the result.

Steps to Perform:

1. Define a Complex class with private double real and double imag.
2. Implement the parameterized constructor Complex(double r, double i).
3. Implement the operator+ member function:

```
Complex operator+(const Complex& other) const {
   // Calculate new real and imaginary parts
   // Create and return a new Complex object
}
```

4. Implement the displayComplex() method.
5. In main(), create Complex objects (e.g., Complex c1(3, 4); Complex c2(1, 2);), use Complex c3 = c1 + c2;, and then display c3.

Libraries/Functions Required:

- iostream for input/output.

Sample Test Cases: Input: Complex number 1: 3 4 (representing 3 + 4i) Complex number 2: 1 2 (representing 1 + 2i) **Output:** Sum: 4 + 6i

Problem 2: Write a C++ program to create a Fraction class.

1. The class should have private data members: numerator and denominator (both of type int).
2. Implement a parameterized constructor Fraction(int num, int den) that initializes the fraction. This constructor should also **normalize/simplify** the fraction and ensure the denominator is not zero (if zero, handle it appropriately, e.g., by setting to 1 and printing an error, or throwing an exception).
3. Implement a private helper method simplify() that reduces the fraction to its lowest terms by dividing both numerator and denominator by their GCD.
4. **Overload the following arithmetic operators** as member functions or friend functions (choose the most appropriate for each):
    - operator+ (for addition)
    - operator- (for subtraction)
    - operator* (for multiplication)
    - operator/ (for division) - Handle division by zero for the second operand. Each overloaded operator should return a new Fraction object that is already simplified.
5. Implement a public method displayFraction() to print the fraction (e.g., 3/4 or 5 if denominator is 1).
6. In the main() function, create several Fraction objects, perform various arithmetic operations using the overloaded operators, and display the results.

Steps to Perform:

1. Define a Fraction class with private int numerator and int denominator.
2. Implement a gcd(int a, int b) helper function (can be a static member or a standalone function).

3. Implement the simplify() private method.
4. Implement the parameterized constructor Fraction(int num, int den) that calls simplify().
5. Implement operator+, operator-, operator*, operator/.
6. Implement displayFraction().
7. In main(), test all operations with various fractions, including those that simplify and those that might lead to division by zero.

Libraries/Functions Required:

- iostream for input/output.
- numeric for std::gcd (C++17) or implement your own GCD function.
- stdexcept for exceptions (optional, for robust error handling).

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
// In main():
Fraction f1(1, 2); // 1/2
Fraction f2(3, 4); // 3/4
Fraction f3(2, 4); // 2/4 (should simplify to 1/2)
Fraction f4(5, 0); // Division by zero case

Output:

Fraction 1: 1/2
Fraction 2: 3/4
Fraction 3: 1/2 (simplified from 2/4)
Error: Denominator cannot be zero. Setting to 0/1. // Or throw exception
Fraction 4: 0/1

Addition (1/2 + 3/4): 5/4
Subtraction (3/4 - 1/2): 1/4
Multiplication (1/2 * 3/4): 3/8
Division (3/4 / 1/2): 3/2

Experiment 6: Inheritance

Objective: Implement different types of inheritance. This lab introduces inheritance, a fundamental OOP concept that allows a class to inherit properties and behaviours from another class. This promotes code reusability and establishes an "is-a" relationship between classes. Students will explore single, multiple, hierarchical, and multilevel inheritance.

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP). It allows a new class (the **derived class** or **subclass**) to inherit attributes and behaviors from an existing class (the **base class** or **superclass**). This promotes:

- **Code Reusability:** Common properties and methods can be defined once in the base class and reused by multiple derived classes.
- **"Is-A" Relationship:** Inheritance models a hierarchical relationship where the derived class is a specialized version of the base class. For example, a Student *is a* Person.
- **Extensibility:** New functionalities specific to the derived class can be added without modifying the base class.

Problem 1: Write a C++ program to demonstrate single inheritance using Person and Student classes.

1. Create a Person class (base class) with the following private data members: name (string) and age (int).
2. Implement a constructor for the Person class that takes name and age as arguments to initialize these members.
3. Implement a public method displayPerson() in the Person class to print the person's name and age.
4. Create a Student class (derived class) that **publicly inherits** from the Person class.
5. The Student class should have additional private data members: studentId (string) and major (string).
6. Implement a constructor for the Student class that takes name, age, studentId, and major as arguments. This constructor should properly initialize the inherited Person members by calling the Person class constructor in its initializer list.
7. Implement a public method displayStudent() in the Student class that first calls displayPerson() (to show inherited details) and then prints the studentId and major.
8. In the main() function, create a Student object, initialize it using its constructor, and then call its displayStudent() method to show all details.

Steps to Perform:

1. Define the Person class with private name, age, a parameterized constructor, and displayPerson().
2. Define the Student class inheriting from Person (class Student : public Person).
3. Add private studentId, major to Student.
4. Implement Student's constructor, ensuring it calls the Person constructor in its initializer list (e.g., Student(string name, int age, string id, string major) : Person(name, age), studentId(id), major(major) {}).
5. Implement displayStudent() in Student.
6. In main(), create a Student object and test.

Libraries/Functions Required:

- iostream for input/output.
- string for string manipulation.

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

```
C++
// In main():
Student student1("Alice Smith", 20, "S1001", "Computer Science");
student1.displayStudent();
```

Output:

```
Person constructor called.
Student constructor called.
Person Details:
  Name: Alice Smith
  Age: 20
Student Details:
  Student ID: S1001
  Major: Computer Science
```

Problem 2: Write a C++ program to illustrate the usage of this pointer and base class member access.

1. Create a Vehicle class (base class) with a protected std::string color and a constructor that initializes color. Include a displayColor() method.
2. Create a Car class (derived class) that publicly inherits from Vehicle.
3. The Car class should have a private std::string model and an int year.
4. Implement two constructors in the Car class:
    - A parameterized constructor Car(std::string color, std::string model, int year):
        - Use the **initializer list** (: Vehicle(color)) to call the Vehicle class constructor.
        - Use this->model = model; and this->year = year; to explicitly use the this pointer for instance variable assignment (even when not strictly necessary for ambiguity).
    - A default constructor Car():
        - Use **constructor delegation** (: this("White", "Unknown Model", 2023)) to call the parameterized constructor of the Car class.
5. Implement a displayCarDetails() method in the Car class that:

    - Calls Vehicle::displayColor() to explicitly display the inherited color via the base class scope.
    - Displays the model and year using this->model and this->year.
6. In the main method, create objects of Car using both constructors and display their details.

Steps to Perform:

1. Define Vehicle class with protected color, constructor, and displayColor().
2. Define Car class inheriting public Vehicle, with model, year.
3. Implement both Car constructors as specified, using initializer list for Vehicle constructor and this(...) for delegation.
4. Implement displayCarDetails() using Vehicle::displayColor() and this->model, this->year.
5. In main(), create Car objects and test.

Libraries/Functions Required:

- iostream for input/output.
- string for strings.

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
// In main():
Car car1("Red", "Sedan", 2024);
Car car2; // Uses default constructor

Output:

Vehicle constructor called.
Car parameterized constructor called.
Car Details:
  Color: Red
  Model: Sedan
Year: 2024
Vehicle constructor called.
Car parameterized constructor called.
Car default constructor called (delegating).
Car Details:
  Color: White
  Model: Unknown Model
  Year: 2023

The final keyword in C++ (introduced in C++11) is a context-sensitive keyword used to restrict the user. It can be applied to:

- **final methods:**
  - When applied to a virtual method in a base class, it prevents any derived class from overriding that specific method. This is useful when you want to ensure that a class's specific implementation of a virtual function remains consistent across the inheritance hierarchy.
- **final classes:**
  - When applied to a class, it prevents any other class from inheriting from it. This is useful when you want to ensure that a class's design and implementation are not extended or modified through inheritance.

Note that C++ does not have a direct equivalent for final variables in the same way Java does for constants. For constants in C++, you typically use const or constexpr.

This problem will demonstrate the uses of final for methods and classes.

Problem 3: Write a C++ program to illustrate the usage of the final keyword.

1. **final method:**
   - Create a base class Animal with a public virtual void makeSound() method that prints a generic sound.
   - Mark the makeSound() method as final (virtual void makeSound() final).
   - Create a subclass Dog that publicly inherits from Animal.
   - Attempt to override the makeSound() method in Dog (this attempt should result in a compile-time error, which you will comment out to allow compilation).
   - Add a non-final virtual method eat() in Animal and override it in Dog to show the difference.
2. **final class:**
   - Create a final class ImmutablePoint with private int x and int y, and a constructor to initialize them.
   - Attempt to create another class ColoredPoint that tries to extend ImmutablePoint (this attempt should result in a compile-time error, which you will comment out).
   - Demonstrate creating an ImmutablePoint object and accessing its values.

Steps to Perform:

1. Define Animal class with virtual void makeSound() final and virtual void eat().

2. Define Dog class inheriting public Animal. Attempt to define void makeSound() override (comment out). Define void eat() override.
3. Define ImmutablePoint as a final class with private members and a constructor.
4. Attempt to define ColoredPoint inheriting public ImmutablePoint (comment out).
5. In main(), create objects and demonstrate all concepts.

Libraries/Functions Required:

- iostream for input/output.
- string (optional, but good for general use).

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
```
// In main():
Animal* myAnimal = new Animal();
myAnimal->makeSound();
myAnimal->eat();
delete myAnimal;

Dog* myDog = new Dog();
myDog->makeSound(); // Calls Animal's final method
myDog->eat(); // Calls Dog's overridden method
delete myDog;

ImmutablePoint p(10, 20);
std::cout << "Point: (" << p.getX() << ", " << p.getY() << ")" << std::endl;
```

Output:

```
--- Demonstrating final methods ---
Animal makes a generic sound.
Animal is eating.
Animal makes a generic sound.
Dog is eating kibble.
(Note: Attempting to override makeSound() in Dog would cause a compilation error.)
```

Experiment 7: Polymorphism and Virtual Functions

Objective: To understand runtime polymorphism in C++ and the crucial role of virtual functions in achieving it. This lab demonstrates how objects of different derived classes can be treated uniformly through a base class pointer or reference, with the correct method implementation being invoked at runtime.

Polymorphism (meaning "many forms") is a fundamental concept in Object-Oriented Programming that allows objects of different classes to be treated as objects of a common base class. In C++, runtime polymorphism is primarily achieved through virtual functions.

- **Virtual Functions:** A member function declared with the virtual keyword in a base class indicates that it can be overridden by derived classes. When a virtual function is called through a pointer or reference to the base class, the decision of which version of the function to execute (base or derived) is made at runtime, based on the *actual type* of the object, not the type of the pointer/reference. This is known as **dynamic dispatch**.
- **Pure Virtual Functions and Abstract Classes:** If a virtual function in a base class has no implementation and is declared with = 0 (e.g., virtual double calculateArea() = 0;), it becomes a **pure virtual function**. A class containing at least one pure virtual function is an **abstract class**. You cannot create direct objects of an abstract class; it serves as an interface for its derived classes, forcing them to provide implementations for the pure virtual functions. This is ideal when a base class concept (like Shape) has a method (like calculateArea) that makes sense conceptually but cannot have a meaningful default implementation in the base class itself. In this problem, Shape will be an abstract base class with a pure virtual calculateArea() method. Circle and Rectangle will be concrete derived classes that provide their specific implementations for calculateArea().

Problem 1: Write a C++ program to demonstrate runtime polymorphism using a Shape class hierarchy.

1. Create an **abstract base class** named Shape.
2. The Shape class should have a public virtual void displayInfo() const method that prints a generic message like "This is a shape."
3. The Shape class must include a **pure virtual function** public virtual double calculateArea() const = 0;.
4. Create two derived classes: Circle and Rectangle, both publicly inheriting from Shape.
5. Circle class:
    - Private data member: radius (double).
    - Parameterized constructor to initialize radius.
    - **Override** displayInfo() to print "This is a circle with radius X."

    - **Override** calculateArea() to compute the area of a circle (pitimesradius2).
6. Rectangle class:
    - Private data members: length and width (double).
    - Parameterized constructor to initialize length and width.
    - **Override** displayInfo() to print "This is a rectangle with length X and width Y."
    - **Override** calculateArea() to compute the area of a rectangle (lengthtimeswidth).
7. In the main() function:
    - Create objects of Circle and Rectangle dynamically using new.
    - Store pointers to these objects in a std::vector<Shape*> (demonstrating polymorphism).
    - Iterate through the vector, calling displayInfo() and calculateArea() for each shape via the base class pointers. Observe how the correct overridden method is invoked for each object.
    - Remember to delete the dynamically allocated objects to prevent memory leaks.

Steps to Perform:

1. Define the Shape abstract class with virtual void displayInfo() const and virtual double calculateArea() const = 0;.
2. Define Circle class inheriting public Shape, with radius, constructor, and overridden displayInfo() and calculateArea(). Use M_PI from <cmath> for pi.
3. Define Rectangle class inheriting public Shape, with length, width, constructor, and overridden displayInfo() and calculateArea().
4. In main(), dynamically create Circle and Rectangle objects.
5. Store their pointers in std::vector<Shape*>.
6. Loop through the vector, call displayInfo() and calculateArea() via base class pointers, and then delete each object.

Libraries/Functions Required:

- iostream for input/output.
- vector for dynamic array of pointers.
- cmath for M_PI (or define PI as a constant).
- iomanip for std::fixed and std::setprecision (for formatting output).

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
```
// In main():
Circle* circle1 = new Circle(5.0);
Rectangle* rect1 = new Rectangle(10.0, 4.0);
```

```
std::vector<Shape*> shapes;
shapes.push_back(circle1);
shapes.push_back(rect1);
```

Output:

```
This is a circle with radius 5.
Area: 78.54

This is a rectangle with length 10 and width 4.
Area: 40.00

Destructor for Circle called.
Destructor for Rectangle called.
```

Problem 2: Manage smart devices using base class pointers and virtual destructors.

1. Create an **abstract base class** named SmartDevice.
    - It should have a protected std::string deviceId and a constructor to initialize it.
    - It must have a **pure virtual function** public virtual void connect() = 0; to simulate connecting to the device.
    - It must have a **pure virtual function** public virtual void displayStatus() const = 0; to show device-specific status.
    - **Crucially, declare a virtual ~SmartDevice(); destructor.** Provide a simple implementation (e.g., print a message) to observe its call.
2. Create two concrete derived classes: SmartLight and SmartSpeaker, both publicly inheriting from SmartDevice.
3. SmartLight class:
    - Private data member: bool isOn and a **dynamically allocated** std::string* colorMode.
    - Parameterized constructor to initialize deviceId, isOn, and dynamically allocate/initialize colorMode.
    - **Override** connect() to print a light-specific connection message.
    - **Override** displayStatus() to show light status (on/off, color mode).
    - Implement its own **destructor ~SmartLight()** that prints a message and **deallocates colorMode**.
4. SmartSpeaker class:
    - Private data member: int volume and std::string currentSong.
    - Parameterized constructor to initialize deviceId, volume, and currentSong.
    - **Override** connect() to print a speaker-specific connection message.
    - **Override** displayStatus() to show speaker status (volume, current song).

    - Implement its own **destructor ~SmartSpeaker()** that prints a message. (No dynamic allocation here, but still good practice to have a destructor in a polymorphic class).
5. In the main() function:
    - Create objects of SmartLight and SmartSpeaker dynamically using new.
    - Store pointers to these objects in a std::vector<SmartDevice*> (demonstrating polymorphism).
    - Iterate through the vector:
        - Call connect() for each device.
        - Call displayStatus() for each device.
    - After the loop, iterate through the vector again and delete each SmartDevice* pointer. **Observe the order of destructor calls in the output** to confirm that the derived class destructors are called before the base class destructor, correctly cleaning up resources.

Steps to Perform:

1. Define the SmartDevice abstract class with protected deviceId, constructor, virtual ~SmartDevice(), virtual void connect() = 0;, and virtual void displayStatus() const = 0;.
2. Define SmartLight class inheriting public SmartDevice, with isOn, std::string* colorMode, constructor (allocating colorMode), overridden virtual methods, and its own destructor (~SmartLight()) that deletes colorMode.
3. Define SmartSpeaker class inheriting public SmartDevice, with volume, currentSong, constructor, overridden virtual methods, and its own destructor (~SmartSpeaker()).
4. In main(), dynamically create SmartLight and SmartSpeaker objects.
5. Store their pointers in std::vector<SmartDevice*>.
6. Loop to call connect() and displayStatus().
7. Loop again to delete each SmartDevice* pointer.

Libraries/Functions Required:

- iostream for input/output.
- string for string manipulation.
- vector for dynamic array of pointers.

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
```
// In main():
SmartLight* light1 = new SmartLight("Light001", true, "Warm White");
SmartSpeaker* speaker1 = new SmartSpeaker("Speaker001", 75, "Bohemian Rhapsody");
```

```
std::vector<SmartDevice*> devices;
devices.push_back(light1);
devices.push_back(speaker1);
```

Output:

```
SmartDevice constructor called: Light001
SmartLight constructor called.
SmartDevice constructor called: Speaker001
SmartSpeaker constructor called.

--- Connecting and Displaying Status ---
Smart Light Light001 connecting...
Smart Light Light001 is ON. Color Mode: Warm White.

Smart Speaker Speaker001 connecting...
Smart Speaker Speaker001 is at volume 75, playing: Bohemian Rhapsody.

--- Deleting Devices ---
SmartLight destructor called for Light001. Deallocating colorMode.
SmartDevice destructor called for Light001.
SmartSpeaker destructor called for Speaker001.
SmartDevice destructor called for Speaker001.
```

Experiment 8: Friend Functions and Static Members

Objective: Use friend functions and static members. This lab introduces two specialized concepts in C++: friend functions and static members. Friend functions are non-member functions that are granted access to the private and protected members of a class. Static members (variables and functions) belong to the class itself rather than to individual objects, providing shared data or utility functions across all instances of a class.

Problem 1: Use static variable to count number of objects.

In C++, **static members** (both data members and member functions) belong to the class itself, rather than to any specific object of that class. This means:

- **Static Data Member:**
  - There is only **one copy** of a static data member, shared by all objects of the class.

- It exists even if no objects of the class are created.
- It must be **defined outside the class** in the source file, usually in the .cpp file, to allocate storage for it.
- It can be accessed directly using the class name and the scope resolution operator (e.g., ClassName::staticVariable).

A common application of a static data member is to keep track of the number of objects currently existing for a particular class. You can increment this static counter in the class's constructor (each time a new object is created) and decrement it in the destructor (each time an object is destroyed).

Write a C++ program to create a Widget class.

1. The Widget class should have a private instance data member (e.g., int id).
2. It must have a **private static data member** (e.g., static int objectCount) to keep track of the total number of Widget objects created.
3. Implement a constructor for the Widget class that increments objectCount each time a new Widget object is created.
4. Implement a destructor for the Widget class that decrements objectCount each time a Widget object is destroyed.
5. Implement a **public static member function** (e.g., static int getObjectCount()) that returns the current value of objectCount. This demonstrates how static member functions can access static data members without needing an object instance.
6. In the main() function:
   - Display the initial objectCount (before any objects are created).
   - Create several Widget objects (some on the stack, some dynamically on the heap).
   - Display the objectCount after each object creation.
   - Explicitly delete any dynamically allocated objects.
   - Observe how objectCount changes as objects are created and destroyed.

Steps to Perform:

1. Define the Widget class with private int id and private static int objectCount.
2. Declare the static member objectCount outside the class definition in the .cpp file (e.g., int Widget::objectCount = 0;).
3. Implement the Widget constructor to increment objectCount and assign a unique id.
4. Implement the Widget destructor to decrement objectCount.
5. Implement the static int getObjectCount() method.
6. In main(), create Widget objects on the stack and heap, and print Widget::getObjectCount() at various stages. Remember to delete heap objects.

Libraries/Functions Required:

- iostream for input/output.
- string (optional).

Sample Test Cases:

Input: (No direct user input, operations are hardcoded in main())

```
C++
// In main():
std::cout << "Current Widget count: " << Widget::getObjectCount() << std::endl;

Widget w1; // Stack object
std::cout << "Current Widget count: " << Widget::getObjectCount() << std::endl;

Widget* w2 = new Widget(); // Heap object
std::cout << "Current Widget count: " << Widget::getObjectCount() << std::endl;

{ // New scope to demonstrate object destruction
   Widget w3; // Another stack object
   std::cout << "Current Widget count: " << Widget::getObjectCount() << std::endl;
} // w3 goes out of scope and is destroyed
std::cout << "Current Widget count after w3 destroyed: " << Widget::getObjectCount()
<< std::endl;

delete w2; // Delete heap object
std::cout << "Current Widget count after w2 deleted: " << Widget::getObjectCount() <<
std::endl;
```

Output:

```
Current Widget count: 0
Widget constructor called. ID: 1
Current Widget count: 1
Widget constructor called. ID: 2
Current Widget count: 2
Widget constructor called. ID: 3
Current Widget count: 3
Widget destructor called. ID: 3
Current Widget count after w3 destroyed: 2
Widget destructor called. ID: 2
Current Widget count after w2 deleted: 1
Widget destructor called. ID: 1
```

Problem 2: In C++, **encapsulation** is a core principle where data members are typically kept private to protect them from unauthorized external access. Member functions are the only ones allowed to directly manipulate this private data. However, there are situations

where a non-member function (or a function from another class) needs direct access to the private or protected members of a class. This is where **friend functions** come into play.

- A friend function is a non-member function that is declared as a friend within a class definition.
- By declaring a function as a friend, you grant it special permission to access all private and protected members of that class, even though it is not a member of that class.
- A single friend function can be a friend to multiple classes, allowing it to access private data from all those classes. This is particularly useful when an operation logically involves combining data from different, otherwise unrelated, objects.

For this problem, we will create two distinct classes, each holding some private data. A friend function will be designed to take objects of both these classes and perform an operation (like addition) using their private data, which would otherwise be inaccessible.

Write a C++ program to demonstrate the use of a friend function that operates on data from two different classes.

1. Create a class named Wallet with a private data member int cashAmount.
2. Implement a parameterized constructor for Wallet to initialize cashAmount.
3. Create another class named BankAccount with a private data member int savings.
4. Implement a parameterized constructor for BankAccount to initialize savings.
5. Declare a **non-member function** (e.g., int getTotalFunds(const Wallet& w, const BankAccount& ba)) as a **friend** in *both* the Wallet and BankAccount classes.
6. Implement the getTotalFunds friend function to access the private cashAmount from Wallet and savings from BankAccount, and return their sum.
7. Include display() methods in both Wallet and BankAccount to show their respective private data.
8. In the main() function, create objects of Wallet and BankAccount, display their individual funds, and then use the getTotalFunds friend function to calculate and display the combined total.

Steps to Perform:

1. Define Wallet class with private int cashAmount and a constructor.
2. Inside Wallet class, declare friend int getTotalFunds(const Wallet& w, const BankAccount& ba);.
3. Define BankAccount class with private int savings and a constructor.
4. Inside BankAccount class, declare friend int getTotalFunds(const Wallet& w, const BankAccount& ba);.
5. Implement the getTotalFunds function outside both classes.
6. Implement display() methods for both Wallet and BankAccount.
7. In main(), create objects, display individual amounts, and then call getTotalFunds to show the combined amount.

Libraries/Functions Required:

- iostream for input/output.
- string (optional).

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
```
// In main():
Wallet myWallet(500);
BankAccount myAccount(1500);
```

Output:

```
My Wallet: Cash Amount = $500
My Bank Account: Savings = $1500

Total funds (Wallet + BankAccount): $2000
```

Experiment 9: File Handling

Objective: Implement file input/output operations. This lab focuses on how C++ programs interact with files, enabling persistent storage and retrieval of data. Students will learn to open, read from, write to, and close text and binary files, including techniques for seeking within files.

Problem 1: Write a C++ program that reads content from a specified text file and calculates:

1. The total number of characters.

2. The total number of words.
3. The total number of sentences. The program should then display these counts to the console.

Steps to Perform:

1. Prompt the user to enter the name of the input text file.
2. Open the file using std::ifstream.
3. Check if the file was opened successfully. If not, print an error message and exit.
4. Initialize counters for characters, words, and sentences to zero.
5. Read the file content:
    - You can read character by character using file.get(char_variable) or file.read(&char_variable, 1).
    - Alternatively, read word by word using file >> word_string and count characters in each word.
    - For sentences, look for '.', '!', '?' characters.
6. Increment the respective counters based on the analysis.
7. Close the file.
8. Display the final counts.

Libraries/Functions Required:

- iostream for console I/O.
- fstream for file I/O (std::ifstream).
- string for string manipulation (if reading words or lines).
- cctype for character classification (e.g., isspace, isalpha).

Sample Test Cases:

Input File (sample.txt):

Hello, world! This is a test file.
It has multiple lines. Isn't it great?

Output:

Enter the name of the input file: sample.txt
File opened successfully.

File Analysis:
Total Characters: 67
Total Words: 13
Total Sentences: 3

Problem 2: Write a C++ program that reads all content from a source text file, converts all alphabetic characters to uppercase, and then writes the modified content into a new destination text file.

Steps to Perform:

1. Prompt the user for the name of the input file and the output file.
2. Open the input file using std::ifstream.
3. Open the output file using std::ofstream.
4. Check if both files were opened successfully. Handle errors if not.
5. Read the input file character by character (e.g., using file.get()).
6. For each character read:
    - Use std::toupper() (from <cctype>) to convert it to uppercase. Note that toupper() works on int and returns int, so cast the character to unsigned char before passing to toupper() to avoid issues with negative char values, then cast back to char.
    - Write the converted character to the output file using outputFile.put().
7. Close both files.
8. Print a success message.

Libraries/Functions Required:

- iostream for console I/O.
- fstream for file I/O (std::ifstream, std::ofstream).
- cctype for std::toupper().

Sample Test Cases:

Input File (input.txt):

This is a mixed case sentence.
Hello World!
123 numbers.

**Output File (output.txt): (Content of output.txt after program execution)

THIS IS A MIXED CASE SENTENCE.
HELLO WORLD!
123 NUMBERS.

Problem 3: Write a C++ program that reads a text file, identifies and removes duplicate lines, and then writes the unique lines to a new output file. The order of the unique lines in the output file does not strictly need to be preserved from the original file if using std::set (as std::set stores elements in sorted order). If order preservation is critical, a

std::unordered_set or a std::vector with manual checking would be needed. For this problem, std::set is acceptable.

Steps to Perform:

1. Prompt the user for the name of the input file and the output file.
2. Open the input file using std::ifstream.
3. Open the output file using std::ofstream.
4. Check if both files were opened successfully. Handle errors if not.
5. Declare a std::set<std::string> to store unique lines.
6. Read the input file line by line using std::getline(inputFile, line_string).
7. For each line read, insert it into the std::set. The set will automatically handle duplicates, ensuring only unique lines are stored.
8. After reading all lines from the input file, iterate through the std::set.
9. Write each unique line from the set to the output file, followed by a newline character.
10. Close both files.
11. Print a success message.

Libraries/Functions Required:

- iostream for console I/O.
- fstream for file I/O (std::ifstream, std::ofstream).
- string for reading lines.
- set for storing unique lines (std::set).

Sample Test Cases:

Input File (data.txt):

apple
banana
orange
apple
grape
banana
kiwi

**Output File (unique_data.txt): (Content of unique_data.txt after program execution)

apple
banana
grape
kiwi
orange

(Note: The order might be sorted alphabetically if std::set is used, as shown above. If original order is required, a different approach would be needed, but for "removing duplicates," this is standard.)

Problem 4: Create a C++ class called Student with private data members: int id, std::string name, and double gpa. Write a student manager program that allows a user to:

1. **Add a new student:** Prompt for id, name, gpa, create a Student object, and add it to an in-memory collection (e.g., std::vector<Student>).
2. **List all students:** Display details of all students currently in memory.
3. **Save students to file:** Write all student objects from the in-memory collection to a binary file (e.g., students.dat).
4. **Load students from file:** Read student objects from the binary file into the in-memory collection, replacing any existing data.
5. **Exit:** Terminate the program.

Steps to Perform:

1. Define the Student class with private id, name, gpa.
2. Implement a parameterized constructor for Student.
3. Implement a displayStudent() method in Student.
4. In the main() function:
    - Create a std::vector<Student> to hold student objects in memory.
    - Implement a menu-driven interface for the user.
    - **For "Add Student":** Create a Student object and push_back to the vector.
    - **For "List Students":** Iterate through the vector and call displayStudent() for each.
    - **For "Save Students":**
        - Open ofstream in binary mode (std::ios::binary).
        - Iterate through the std::vector<Student>.
        - For each Student object:
            - Write id and gpa directly using outputFile.write(reinterpret_cast<const char*>(&student.id), sizeof(student.id)).
            - For name (string): first write its length, then its characters.
    - **For "Load Students":**
        - Clear the current std::vector<Student>.
        - Open ifstream in binary mode (std::ios::binary).
        - Loop while the file is not at end-of-file:
            - Read id and gpa directly.
            - Read string length, then read characters into a char array, then construct std::string.
            - Create a new Student object and add to the vector.
    - **For "Exit":** Break the loop.

Libraries/Functions Required:

- iostream for console I/O.
- fstream for file I/O (std::ifstream, std::ofstream).
- string for student names.
- vector for managing students in memory.
- limits for std::numeric_limits (useful for clearing input buffer).

Sample Test Cases:

Initial State: students.dat does not exist or is empty.

Input (User Interaction):

Student Manager Menu:
1. Add Student
2. List Students
3. Save Students
4. Load Students
5. Exit
Enter your choice: 1
Enter Student ID: 101
Enter Student Name: Alice
Enter Student GPA: 3.8
Student added.

Enter your choice: 1
Enter Student ID: 102
Enter Student Name: Bob
Enter Student GPA: 3.5
Student added.

Enter your choice: 2
Listing Students:
ID: 101, Name: Alice, GPA: 3.8
ID: 102, Name: Bob, GPA: 3.5

Enter your choice: 3
Students saved to students.dat.

Enter your choice: 4
Students loaded from students.dat. (In-memory data replaced)

Enter your choice: 2
Listing Students:
ID: 101, Name: Alice, GPA: 3.8

ID: 102, Name: Bob, GPA: 3.5

Enter your choice: 5
Exiting program.

Binary File Content (students.dat): (This file will not be human-readable, but its internal structure would correspond to the serialized Student objects.) For example, it might contain raw bytes for 101, 3.8, then length of "Alice", then characters 'A','l','i','c','e', etc.

Experiment 10: Templates and Exception Handling

Objective: Use function and class templates with exception handling. This lab introduces two powerful features in C++: templates and exception handling. Templates enable writing generic code that works with different data types, promoting reusability. Exception handling provides a robust mechanism to deal with runtime errors, ensuring graceful program termination or recovery.

Problem 1: Write a C++ program that implements a **template function** to find the maximum of three values.

1. Define a function template named findMax that takes three arguments of a generic type T.
2. The findMax function should compare the three values and return the largest among them.
3. In the main() function, demonstrate the use of the findMax template function with at least two different data types:
    - Find the maximum of three int values.
    - Find the maximum of three double values.
    - (Optional) Find the maximum of three char values or std::string values to further illustrate generality.

Steps to Perform:

1. Define the template function template <typename T> T findMax(T val1, T val2, T val3):
    - Implement the logic to compare val1, val2, and val3 and return the maximum.
2. In main():
    - Declare three int variables, assign values, and call findMax.

- Declare three double variables, assign values, and call findMax.
- Print the results clearly.

Libraries/Functions Required:

- iostream for input/output.
- string (if demonstrating with strings).

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
```
// In main():
int i1 = 10, i2 = 25, i3 = 15;
double d1 = 3.14, d2 = 2.71, d3 = 5.0;
char c1 = 'X', c2 = 'A', c3 = 'Z';
```

Output:

```
Maximum of (10, 25, 15) is: 25
Maximum of (3.14, 2.71, 5) is: 5
Maximum of (X, A, Z) is: Z
```

Problem 2: Write a C++ program to create a **template class** named MyArray.

1. The MyArray class should be able to store elements of any data type T.
2. It must use **dynamic memory allocation** (new T[size]) in its constructor to create an array of a specified size.
3. Implement a **destructor** (~MyArray()) to correctly deallocate the dynamically allocated memory (delete[]).
4. Implement a public method void setElement(int index, const T& value) to set an element at a given index.
5. Implement a public method T getElement(int index) const to retrieve an element at a given index.
6. For both setElement and getElement, implement **exception handling**:

    - If the index provided is out of the valid range (i.e., index < 0 or index >= size), the method should **throw an std::out_of_range exception** (or a custom exception class you define).
7. In the main() function:
    - Create instances of MyArray for at least two different data types (e.g., MyArray<int>, MyArray<double>).
    - Demonstrate setting and getting elements within valid bounds.
    - Demonstrate attempting to set/get elements outside valid bounds within a try-catch block to show how the exception is caught and handled gracefully.

Steps to Perform:

1. Define the MyArray class template: template <typename T> class MyArray { ... };.
2. Private members: T* arr and int size.
3. Implement constructor MyArray(int s): allocate arr.
4. Implement destructor ~MyArray(): delete[] arr;.
5. Implement setElement(int index, const T& value):
    - Check index bounds. If invalid, throw std::out_of_range("Index out of bounds for setElement.");.
    - Otherwise, arr[index] = value;.
6. Implement getElement(int index) const:
    - Check index bounds. If invalid, throw std::out_of_range("Index out of bounds for getElement.");.
    - Otherwise, return arr[index];.
7. In main():
    - Create MyArray<int> and MyArray<double> objects.
    - Use try-catch blocks around calls that might throw exceptions.
    - Print appropriate messages for successful operations and caught exceptions.

Libraries/Functions Required:

- iostream for input/output.
- stdexcept for std::out_of_range exception.
- string (if using MyArray<std::string>).

Sample Test Cases:

Input: (No direct user input, operations are hardcoded in main())

C++

```
// In main():
MyArray<int> intArray(5);
```

```
intArray.setElement(0, 10);
intArray.setElement(4, 50);

MyArray<double> doubleArray(3);
doubleArray.setElement(1, 3.14);
```

Output:

```
MyArray<int> created with size 5.
Element at index 0: 10
Element at index 4: 50

Attempting to access out of bounds for intArray:
Caught exception: Index out of bounds for getElement.

MyArray<double> created with size 3.
Element at index 1: 3.14

Attempting to set out of bounds for doubleArray:
Caught exception: Index out of bounds for setElement.

MyArray<double> destructor called.
MyArray<int> destructor called.
```

Problem 3: Write a C++ program that performs division and handles the "division by zero" error using exception handling.

1. Define a **custom exception class** named DivideByZeroException that publicly inherits from std::runtime_error. Its constructor should take a const char* message and pass it to the base class constructor.
2. Implement a function double divideNumbers(double numerator, double denominator):
    - This function should perform the division numerator / denominator.
    - If denominator is 0.0, it must **throw an instance of DivideByZeroException** with an appropriate error message (e.g., "Error: Division by zero is not allowed!").
    - Otherwise, it should return the result of the division.
3. In the main() function:
    - Use a try-catch block to call divideNumbers with valid inputs (non-zero denominator) and print the result.
    - Use another try-catch block to call divideNumbers with invalid inputs (zero denominator). In the catch block, catch the DivideByZeroException and print its error message using the what() method.

- (Optional) Add a generic catch(...) block to catch any other unexpected exceptions.

Steps to Perform:

1. Include <stdexcept> for std::runtime_error.
2. Define class DivideByZeroException : public std::runtime_error { ... }; with its constructor.
3. Implement double divideNumbers(double numerator, double denominator) with the if (denominator == 0.0) throw DivideByZeroException(...) logic.
4. In main():
   - Set up a try block for a valid division.
   - Set up a try block for an invalid division.
   - For each try block, follow it with a catch (const DivideByZeroException& e) block to handle the specific exception.
   - (Optional) Add catch (const std::exception& e) for other standard exceptions and catch (...) for any unknown exceptions.

Libraries/Functions Required:

- iostream for console I/O.
- stdexcept for std::runtime_error.

Sample Test Cases:

Input: (No direct user input, values are hardcoded in main())

C++
// In main():
double num1 = 10.0, den1 = 2.0;
double num2 = 5.0, den2 = 0.0;

Output:

--- Test Case 1: Valid Division ---
Attempting to divide 10.0 by 2.0
Result of division: 5

--- Test Case 2: Division by Zero ---
Attempting to divide 5.0 by 0.0
Caught exception: Error: Division by zero is not allowed!

Experiment 11: STL Containers

Objective: Explore STL containers like vector, set, map. This lab introduces the Standard Template Library (STL) containers, powerful and efficient data structures that simplify common programming tasks. Students will learn to use vector for dynamic arrays, set for unique sorted collections, and map for key-value pairs.

The C++ **Standard Template Library (STL)** provides a collection of powerful, generic components that can be used with any data type. Among these, **containers** are classes that store collections of objects.

- **std::vector:** This is one of the most commonly used STL containers. It's a dynamic array, meaning its size can grow or shrink automatically as elements are added or removed during runtime. This is a significant advantage over traditional C-style arrays, whose size must be fixed at compile time.
    - **Dynamic Sizing:** You don't need to specify the size of a std::vector when you declare it (though you can). You can add elements using push_back(), and the vector will manage its memory.
    - **Contiguous Memory:** Elements in a std::vector are stored in contiguous memory locations, which allows for efficient random access (accessing an element by its index, like myVector[i]).
    - **Iterators:** std::vector supports iterators, which are like pointers that allow you to traverse through the elements.

Problem 1: Write a C++ program that uses std::vector to store a list of student marks.

1. Declare a std::vector to hold int (or double) type marks.
2. Prompt the user to enter the number of students.
3. Use a loop to prompt the user to enter marks for each student. Add each entered mark to the std::vector using push_back().
4. After all marks are entered, display all the stored marks to the console, clearly labeled.
5. (Optional) Calculate and display the average mark.

Steps to Perform:

1. Include <iostream> and <vector>.
2. Declare std::vector<int> studentMarks; (or double).
3. Prompt for the number of students.
4. Use a for loop to iterate n times:
    - Prompt for a mark.
    - Read the mark into a temporary variable.
    - Use studentMarks.push_back(mark); to add it to the vector.
5. Display a header like "Student Marks:".
6. Use a range-based for loop (for (int mark : studentMarks)) or a traditional for loop with an index to iterate through the vector and print each mark.
7. (Optional) Loop again to sum marks and divide by studentMarks.size() for the average.

Libraries/Functions Required:

- iostream for console I/O.
- vector for std::vector.
- numeric for std::accumulate (optional, for sum).

Sample Test Cases:

Input:

Enter the number of students: 3
Enter mark for student 1: 85
Enter mark for student 2: 90
Enter mark for student 3: 78

Output:

Student Marks:
85 90 78
Average Mark: 84.33

Problem 2: Write a C++ program to maintain student records using std::map and std::set.

1. Create a Student class with private members: int id, std::string name, and double gpa.
2. Implement a parameterized constructor for Student and a displayStudent() method.
3. Implement operator< for the Student class based on id if you plan to store Student objects directly in a std::set or std::map where Student itself is the key (though int ID as key is simpler for map).
4. In the main() function:
    - Declare a std::map<int, Student> to store student records, where the int key is the studentId.
    - Declare a std::set<std::string> to store unique student names for quick name lookup.
    - Implement a menu-driven program with the following functionalities:
        - **Add Student:** Prompt for id, name, gpa. Create a Student object. Insert it into the std::map using id as the key. Also, insert the name into the std::set. Handle cases where id is already present.
        - **Search by ID:** Prompt for a studentId. Use std::map::find() to search for the student. If found, display their details; otherwise, print "Student not found."
        - **Check Name Existence:** Prompt for a studentName. Use std::set::count() or std::set::find() to check if the name exists in the set. Print whether the name exists or not.
        - **Display All Students (Sorted by ID):** Iterate through the std::map (which is naturally sorted by key) and display all student records.
        - **Exit:** Terminate the program.

Steps to Perform:

1. Define Student class with id, name, gpa, constructor, and displayStudent().
2. In main():
    - Declare std::map<int, Student> studentRecords;.
    - Declare std::set<std::string> uniqueNames;.
    - Implement a do-while loop for the menu.
    - For "Add Student": Read input, create Student object, use studentRecords[id] = studentObject; (or studentRecords.insert({id, studentObject});), and uniqueNames.insert(name);.
    - For "Search by ID": Read ID, use studentRecords.find(id). Compare with studentRecords.end().
    - For "Check Name Existence": Read name, use uniqueNames.count(name).
    - For "Display All Students": Use a range-based for loop for (const auto& pair : studentRecords) to iterate and display pair.second.displayStudent().

Libraries/Functions Required:

- iostream for console I/O.
- string for names.
- map for std::map.
- set for std::set.
- limits for std::numeric_limits (for clearing input buffer after cin).

Sample Test Cases:

Input (User Interaction):

Student Record Manager
1. Add Student
2. Search by ID
3. Check Name Existence
4. Display All Students
5. Exit
Enter your choice: 1
Enter Student ID: 101
Enter Student Name: Alice
Enter Student GPA: 3.8
Student added.

Enter your choice: 1
Enter Student ID: 102
Enter Student Name: Bob
Enter Student GPA: 3.5
Student added.

Enter your choice: 1
Enter Student ID: 101
Enter Student Name: Alice Smith
Enter Student GPA: 3.9
Student with ID 101 already exists. Update not performed.

Enter your choice: 4
--- All Students (Sorted by ID) ---
ID: 101, Name: Alice, GPA: 3.8
ID: 102, Name: Bob, GPA: 3.5

Enter your choice: 2
Enter Student ID to search: 101
Student Found: ID: 101, Name: Alice, GPA: 3.8

Enter your choice: 2
Enter Student ID to search: 103
Student not found.

```
Enter your choice: 3
Enter Student Name to check: Alice
Name 'Alice' exists.

Enter your choice: 3
Enter Student Name to check: Charlie
Name 'Charlie' does not exist.

Enter your choice: 5
Exiting program.
```

Experiment 12: Unit Testing and Test-Driven Development (TDD)

Objective: To understand what unit testing is, its importance in software development, and how to write basic unit tests for C++ functions.

Unit Testing is a software testing method where individual units or components of a software are tested. A "unit" is the smallest testable part of an application, often a single

function, method, or class. The goal is to validate that each unit of the software performs as designed.

Why is Unit Testing Important?

- **Early Bug Detection:** Catch bugs early in the development cycle, making them cheaper and easier to fix.
- **Facilitates Change:** Allows developers to refactor code or introduce new features with confidence, knowing that existing functionality is still working.
- **Improves Design:** Writing tests often forces developers to think about the design of their code, leading to more modular, testable, and maintainable units.
- **Documentation:** Tests serve as living documentation of how a unit is supposed to behave.
- **Regression Prevention:** Prevents regressions (new bugs introduced into previously working code).

Characteristics of Good Unit Tests:

- **Isolated:** Each test should run independently and not rely on the state of other tests.
- **Fast:** Tests should execute quickly to encourage frequent running.
- **Repeatable:** Running the same test multiple times should yield the same result.
- **Automated:** Tests should be runnable automatically, ideally as part of a continuous integration pipeline.

In C++, popular unit testing frameworks like **Google Test** (GTest) provide rich features for writing and running tests. For this basic demonstration, we'll use a simplified assert-like approach to illustrate the core idea without the overhead of setting up a full framework. In a real-world project, you would always use a robust framework.

Problem 1: Implement a simple C++ function isEven(int number) that returns true if the number is even and false otherwise. Then, write basic unit tests to verify its correctness for various inputs.

Steps to Perform:

1. Implement the isEven function.
2. Create a simple testing mechanism (e.g., a TEST macro or function) to compare expected outputs with actual outputs.
3. Write several test cases covering positive, negative, and zero inputs.

Libraries/Functions Required:

- iostream for output.

- string for test messages.
- cassert (optional, for basic assertions).

Sample Input/Output:

(No direct user input for the program, the output is generated by the tests.)

--- Running isEven() Unit Tests ---

[PASS] isEven(4)

[PASS] isEven(7)

[PASS] isEven(0)

[PASS] isEven(-6)

[PASS] isEven(-3)

[PASS] isEven(1000000)

[PASS] isEven(999999)

--- isEven() Test Summary ---

Passed: 7

Failed: 0

Problem 2: TDD approach