National University of Computer and Emerging Sciences

Assignment
For
Object-Oriented Programming

Course Instructor	Ch. Usman Ghous
Semester	Spring 2024
Open Date	04-May-2024
Submission Date	12-May-2024

FAST School of Computing

Submission deadline
12-May-2024

Instructions:

- Make a word document with the naming convention "SECTION_ LAB#_ROLLNO" and put all your source code and snapshots of its output in it. Make sure your word file is formatted properly.
- Zip the word file with all Source code.
- Plagiarism is strictly prohibited.
- Do not discuss solutions with one another.

Question#1 Multi-functional Template-Based Program

Design a C++ program that incorporates the following functionalities using templates:

1. Student Data Management:
   - Create a template class to store student data.
   - The student data should include their name, roll number, course list, and section.
   - Implement getter and setter methods to manage and display the student data.
2. Recursive Prime Number Printer:
   - Implement a recursive template function to print prime numbers within a given range.
   - The function should accept the range as template parameters.
3. Baseball Players Data Management:
   - Declare another template class to store data of baseball players.
   - Each player's data should include their name and the number of home runs.
   - Utilize a dynamic array to store multiple players' data.
   - Implement methods to add, display, and update player data.
4. Template-Based Calculator:
   - Design a calculator using templates.
   - Implement template functions for addition, subtraction, and multiplication operations.
   - Ensure that the calculator is capable of performing arithmetic operations on different data types.

Question#2 Exception Handling

Write a C++ program that prompts the user to enter a person's date of birth in numeric form such as 8-27-1980. The program should then output the date of birth in the form: August 27, 1980. Your program must contain at least two exception classes: `InvalidDay` and `InvalidMonth`. If the user enters an invalid value for the day or month, the program should throw and catch the appropriate exception. Additionally, ensure correct handling of leap years for February and consider boundary conditions for day, month, and year inputs.

Here are the steps your program should follow:

1. Prompt the user to input a date in the specified format (MM-DD-YYYY).

2. Parse the input to extract the month, day, and year.

3. Validate the extracted values:

   - Check if the month is in the range 1-12. If not, throw `InvalidMonth`.

   - Based on the month, determine the maximum number of days (considering leap years for February).

   - Check if the day is within the valid range for the given month. If not, throw `InvalidDay`.

4. If the input passes validation, convert the numeric month to its corresponding name (e.g., 1 to January, 2 to February, etc.).

5. Output the date of birth in the specified format: Month Day, Year.

Ensure your program handles exceptions gracefully by catching and displaying appropriate error messages. This will ensure a smooth user experience and help in troubleshooting any issues with the input data.

Question#3

Fraction Arithmetic Calculator

Objective

Develop a robust menu-driven fraction arithmetic calculator that gracefully handles potentialerrors and provides a user-friendly experience.

Functionality

The calculator will support the following core arithmetic operations on fractions represented inthe form a/b (where a and b are integers, and b is non-zero):

- Addition (+)
- **Subtraction (-)**
- Multiplication (*)
- **Division (/)**

Program Structure

- Function menu()
  - Displays a clear welcome message explaining the calculator's functionality.
  - Presents the menu of arithmetic operations (+, -, *, /).
  - Guides the user on how to enter fractions in the correct format (a/b).
  - Obtains valid user input for the selected operation.
- Functions addFractions(), subtractFractions(), multiplyFractions(), divideFractions()
  - Each function takes four integer inputs (numerators and denominators of two fractions).
  - Performs the specified arithmetic operation.
  - Returns the resulting fraction's numerator and denominator.

Exception Handling

- **Division by Zero:** Implement checks to prevent division by zero operations. The program should throw an appropriate exception and display a user-friendly error message.
- **Invalid Input:** Validate all user inputs to ensure fractions are entered in the correct format (a/b) and that values are integers. The program should handle incorrect input gracefully, provide guidance to the user, and prompt for re-entry.

User Experience

- **Clear Instructions:** Provide concise instructions throughout the program's execution.
- **Informative Error Messages:** In cases of exceptions, display clear error messages thathelp the user understand the problem and how to provide correct input

Question#4

Text-Based Adventure Engine

Objective

Design and develop a flexible text-based adventure game engine that provides a foundation forcrafting interactive, narrative-driven experiences. The engine should emphasize extensibility, allowing developers to easily create unique game worlds with diverse locations, items, and challenges.

Core Functionality

- **World Representation:** Establish a system for modeling interconnected rooms or locations. Each location should have:
    - A detailed description to immerse the player.
    - Clearly defined exits (north, south, east, west, etc.) that link to other locations.
    - Potential for containing interactive items or objects.
- Player Interaction:
    - Implement a command parser to interpret player input.
    - Support basic movement commands (north, south, east, west).
    - Enable actions like:
        - "look" or "examine" for detailed inspection of the surroundings.
        - "take" or "get" to acquire items.
        - "inventory" to view what the player is carrying.
- Game Logic:
    - Maintain the player's current location.
    - Process player commands, validating actions and updating the game state accordingly.
    - Provide descriptive feedback to the player based on their actions.

Engine Design

- **Object-Oriented Approach:**
    - Create a Location class to encapsulate room descriptions, exits, and item storage.
    - Use a Player class to track location, inventory, and potentially other attributes (health, etc.).
    - Consider an Item class for objects that can be interacted with or collected.

- **Extensibility:**

    - Design with future expansion in mind.
    - Emphasize clear separation of concerns to enable easy addition of:
        - New locations.
        - Unique items with special properties.
        - Complex puzzles or challenges.

Exception Handling

1. Invalid Movement Commands

- **Problem:** The player enters a direction that doesn't correspond to a valid exit fromthe current location.
- **Exception Type:** You could create a custom exception like InvalidMovementException.
- Handling:
    - Catch the exception in your game logic loop.

- Display a friendly message to the player: "You can't go that way."

2. Attempting Actions on Non-Existent Objects

- **Problem:** The player tries to "take" or "examine" an item or object that isn't presentin the room.
- **Exception Type:** A custom ObjectNotFoundException could be suitable.
- Handling:
  - Catch the exception.
  - Inform the player: "There's no [object name] here."

3. Unexpected Input

- **Problem:** The player enters a command that isn't recognized by the parser (not a movement command, "look", "take", etc.).
- **Exception Type:** A custom InvalidCommandException might be appropriate.
- Handling:
  - Catch the exception.
  - Display a helpful message: "I don't understand that command."
  - Consider providing a list of valid commands.

4. Technical Issues

- **Problem:** Issues like file loading errors (if you store map data externally) or other unforeseen runtime problems.
- **Exception Type:** Use more general exception types (e.g., in C++, std::runtime_erroror a derivative).
- Handling:
  - Catch these exceptions at a top-level in your game loop.
  - Log the error for debugging.
  - Display an apologetic message to the player indicating an unexpected issuehas occurred.

Question#5

Student Grade Management System

Objective

Develop a comprehensive command-line application for streamlined student grade management and academic performance tracking. Emphasize flexibility through templates androbust error handling.

Core Functionality

- **Student Records:**

- Utilize a templated Student class to store information:
  - template <typename T> class Student { ... } // T for ID data type
  - Full name
  - Unique student identification number (type T)
  - Enrollment data across multiple courses
- Calculate and maintain each student's cumulative grade point average (GPA).

- Course Management:
  - Templated Course class to represent distinct courses:
    - template <typename T> class Course { ... } // T for grade data type
    - Course title or code
    - Roster of enrolled students (using Student objects).
    - Individual student grades for various assessments (type T).
- System Operations:
  - **Add Student:** Enable the creation of new student records.
  - **Enroll Student:** Facilitate enrolling students into specific courses.
  - **Add/Update Grades:** Allow input and modification of student grades.
  - **View Student Record:** Display a complete transcript.
  - **Generate Reports:** Provide options to generate class-wide reports.

Data Persistence:

- **Simple Text Files:** For basic storage

Templates: Employ templates to accommodate potential variations in data types for studentIDs (integer, string, etc.) and grade representations (numeric, letter grades).

Exception Handling

- **Invalid Input:** Validate all user input (IDs, grades, etc.). Throw custom exceptions (e.g., InvalidGradeException, StudentNotFoundException) for improper data.
- **File I/O Errors:** Handle potential errors during file operations. Throw FileAccessException or similar.
- **Robustness:** Implement comprehensive error handling throughout the application logic.

Additional Considerations

- **Security:** If handling sensitive data, consider input validation and potential data encryption.
- **User Interface:** Design a clear command-line interface with intuitive commands and helpful prompts.

Question#6

Templated Mathematical Operations Library

Objective

Develop a suite of templated mathematical functions in C++ designed to provide flexibility across various numeric data types while carefully addressing potential issues arising from mixed-type operations.

Core Functionality

- **Basic Arithmetic:**
    - template <typename T> T findMax(T a, T b): Determines the larger of two values.
    - template <typename T> T findMin(T a, T b): Determines the smaller of two values.
    - template <typename T> T add(T a, T b)
    - template <typename T> T subtract(T a, T b)
    - (Potentially add more operations: multiply, divide, etc.)
- Statistical Calculations:
    - template <typename T> T calculateAverage(T arr[], int size): Computes the meanof an array of numeric values.
    - template <typename T> T calculateMedian(T arr[], int size): Finds the median value.
- Geometric Calculations:
    - Create templated classes to represent shapes (e.g., Circle<T>, Rectangle<T>, Triangle<T>)
    - Implement calculateArea() and calculatePerimeter() methods within these shape classes, leveraging templates for diverse coordinate types.

Implementation

- **Use suitable concepts of OOP for this question.**

Question#7

Templated Safe Array Container

Objective

Develop a robust SafeArray class template that encapsulates the functionality of traditional arrays while augmenting it with essential bounds checking mechanisms. This design prioritizessafety by preventing out-of-range access errors, employing exception handling for graceful error management.

Core Functionality

- **Templatization:**

    - Implement the SafeArray class as a template (template <typename T>) to accommodate the storage of elements of any data type. This ensures broad reusability.

- Array-like Interface:
    - Overload the [] operator to enable intuitive syntax for element retrieval and modification (e.g., myArray[index]).
    - Thoroughly integrate bounds checking within the overloaded [] operator. If an attempted index falls outside the valid array range, immediately signal an error.
- Exception Handling:
    - Define a custom exception class, ArrayIndexOutOfBoundsException, specifically tailored to this scenario.
    - Construct the exception object to potentially include informative details, such asthe attempted invalid index, aiding in debugging.
    - When an out-of-bounds access is detected, throw an ArrayIndexOutOfBoundsException.

Design Considerations

- **User Experience:** Within the exception handling mechanism, provide clear and informative error messages to guide the user in resolving the issue.
- Performance vs. Safety:
    - Acknowledge the inherent overhead associated with bounds checking.
    - Investigate optimization techniques that mitigate performance impact without compromising the core safety guarantees.
    - Profile the code to measure any potential performance differences compared to standard arrays.