Lab 2: Term Project Discussion and Basic Programming in C++

Course Code: CSE 4302
Week: 2
Topics: Project Discussion, Enumeration, Structure, Reference Argument, Overload Function, Inline Function, Default Argument, Returning by Reference, `const` Function Argument
Instructors:
Faisal Hussain, Assistant Professor, CSE
Farzana Tabassum, Lecturer, CSE

Term Project

This section describes the term project requirements, timeline, deliverables, expectations, templates, and instructor guidance for the course. Each lab section forms groups internally. The aim is for the students to design and implement a small but complete C++ program that demonstrates the core object-oriented concepts and correct use of exception handling, readable code, and version control.

Key Principles/Scope

- **Focus:** Core OOP concepts in C++: classes and objects, encapsulation (data and module hiding), inheritance, polymorphism, constructors/destructors, basic file and console I/O, namespaces, and exception handling.
- **Keep it simple:** Students are NOT required to use advanced features (smart pointers, heavy templates, external libraries, CI, unit-test frameworks, design patterns beyond basic class design, or complex build systems).
- **Language level:** C++20 or below is accepted. Solutions should compile with a common compiler (`g++`/`clang++`/`msvc`) using standard flags.
- **Collaboration:** Each group uses a single GitHub repository. Minimal but clear version control usage is expected.

Group Formation

Member Count: Groups are formed only within a section. Each project group will have 4 members. If a section size is not divisible by 4, up to three groups may have 5 members (to accommodate remainders). The CRs will coordinate with the students to ensure this.
Group Registration: Once you have decided on the group formation, fill out the following Google Form: https://forms.gle/rxmSzyKvNcTQspuJ9 by providing your group name, remote Git repository link, and member information.

Deliverables and Submission Format

1. **Project Proposal (Due Week 4):** Use the provided proposal template (will be provided in Google Classroom). Submit a single PDF via Google Classroom. Faculty feedback provided by Week 6.
2. **Final Project Report (Week 15 - Tentative):** A short single-file final report PDF. The template will be provided later.
3. **Presentation (Week 15 - Tentative):** Each group will present and demonstrate their working program. The guideline will be provided later.
4. **Peer Evaluation:** Each student will complete a peer evaluation form after presentations. The form will ask each student to evaluate their teammates.

Documentation Requirements

In the remote Git repository (in the README.md file), include:

- **Project title:** Name of the project
- **Short description:** What the project is about.
- **How to compile and run:** Exact commands
- **Sample input files:** Conventions and where to put them
- **Class descriptions:** Each class name, purpose, main fields, and methods.
- **Relationships:** Which classes use/own others, where inheritance is used, and why.
- How polymorphism is applied.
- **Exception Handling:** What exceptions are expected and how they are handled.
- Known issues.
- Authors and short contributions.

Coding Standards and Conventions

Adopt the following minimal conventions to ensure readability and maintainability:

- **File structure:** `.h` files for declarations, `.cpp` files for definitions. One class per pair of `.h`/`.cpp` where practical.
- Header guards or `#pragma once` in each header.
- **Naming:** Use meaningful names. Prefer camelCase or lower_case_with_underscores consistently for variables/functions; class names in PascalCase. State your chosen convention in README.
- **Indentation:** Consistent (2 or 4 spaces), avoid tabs. Keep lines reasonably short (<80 characters).
- **`const` correctness:** Mark methods as `const` when they do not modify object state. Use `const` references for non-modified large parameters.
- **No globals:** Pass needed data via function parameters or class fields.
- **Comments:** Brief function headers and comments, where non-obvious; rely on clear names more than comments.

- **Resource management:** Avoid raw dynamic memory where possible. If dynamic resources are used, document ownership and ensure proper cleanup (destructors). Do not require smart pointers.
- **Basic error handling:** Use exceptions for error conditions (e.g., file open fails, wrong input type) and handle them at a suitable boundary; avoid swallowing exceptions silently.
- Use basic STL containers (`std::vector`, `std::string`, `std::map`, etc.) as needed; complex template metaprogramming is not required.

Version Control

- One remote Git repository (GitHub, GitLab, or any other suitable provider) per group. Create the repository before group registration and add an initial README and .gitignore.
- **Commits:** Make regular meaningful commits with short messages (e.g., ``Added class Player and basic I/O''). Demonstrate individual contribution via commit history (authors will be visible). No complex branching model required; working on the main branch is acceptable for this course.
- Include a brief note in README describing who did what and where the main executable or demo script is, as discussed before.

Build and Execute

- Provide clear compile instructions in README. Example:
  `g++ -std=c++20 -Wall -Wextra -o myprog src/*.cpp`
- Avoid complicated build systems. A simple Makefile is acceptable but not required.
- If external data files are required, include them and explain how to place them relative to the executable.

Testing

- Formal unit testing is not required. Provide a set of manual test cases and/or sample inputs and expected outputs in a `tests/` or `data/` folder. Describe how to run each test in the README.
- Demonstrate that basic edge/error cases are handled (file not found, invalid input, etc.)

Academic Integrity

Students must write their own code for the project. If any code is reused (e.g., from class demos, textbook snippets, or online tutorials), it must be clearly cited in the code by providing a link to the source and what was reused. If you adapt external code, state what you changed and why.

Common Pitfalls and Advice

- **Scope Creep:** Pick a small MVP and deliver that first. Add nice extras only if MVP is stable.

- Keep class responsibilities narrow; avoid ‘’God” classes that do everything.
- Commit early and often. A single final massive commit makes it hard to show contribution.
- Test simple edge cases early. Handling invalid input early avoids last-minute bugs.
- Choose clear names and write a README as if someone new must run your program in 5 minutes.

Example Project Ideas

Library Book Checkout

A console program to manage a small book library with patrons. Support add/remove books, checkout/return, search by title/author, and simple overdue check.
Core OOP: Classes for book, patron, library; encapsulation of book state; basic inheritance for different patron subtypes (e.g., student/faculty). Exception handling for invalid checkout/return.

Simple Banking System

Manage accounts (checking/savings), deposit/withdraw, transfer funds, and view transaction history.
Core OOP: Base account class with derived checking and savings that require inheritance, virtual functions for interest/fees. Transactions class for history, encapsulation of balance, and exception handling for insufficient funds or invalid operations.

TODO Manager

A console task/todo manager supporting tasks with priorities, due dates, categories, and mark-as-done. Search and list by various filters.
Core OOP: Classes for task, category/project, manager; demonstrate composition and encapsulation; use of inheritance for different task types. Exception handling for invalid dates or file I/O.

Lab Tasks

1. Sum-Thing About Fractions

If you have two fractions, a/b and c/d, their sum can be obtained from the formula:

$$\frac{a}{b} + \frac{c}{d} = \frac{a \times d + b \times c}{b \times d}$$

Write a program that asks the user to enter two fractions and then displays their sum in fractional form. You do not need to reduce it to the lowest terms.
You can take advantage of the fact that the extraction operator (>>) can be chained to read in more than one quantity at once:

```
cin >> num1 >> dummychar >> denom1;
```

[Sample Program]

```
Enter first fraction: 1/2
Enter second fraction: 2/5
Sum = 9/10
```

2. Enumerate My Job Title

C++ I/O statements do not automatically understand the data types of enumerations. Instead, the (>>) and (<<) operators think of such variables simply as integers. You can overcome this limitation by using `switch` statements to translate between the user's way of expressing an enumerated variable and the actual values of the enumerated variable. For example, imagine an enumerated type with values that indicate an employee type within an organization:

```
enum etype
{
  laborer, secretary, manager, accountant, executive, researcher
};
```

Write a program that asks the user to specify a type by entering its first letter (‘`l`’, ‘`s`’, ‘`m`’, etc.) and stores the type chosen as a value of a variable of type enum etype. Then it should display the complete word for the type.
Your program should be able to handle invalid input.

[Sample Program]

```
Enter employee type (first letter only): a
Employee type is accountant.
```

3. Restructuring the Fractions

Copy your program written in Task 1 and modify it so that all fractions are stored in a variable of type `struct fraction`, whose two members are the fraction's numerator and denominator (both type int). All fraction-related data should be stored in structures of this type.

4. A Functioning Member of Society

Go through all the example code in Chapter 5. The source code is available on the FTP (ftp://10.220.20.26/ACADEMIC/CSE/Faisal%20Hussain/CSE%204301/Books/student/Progs/) and CSE 4301 Google Classroom. Check and run the following files from Ch05.

- Ref.cpp
- Reforder.cpp
- Referst.cpp
- Overload.cpp
- Overeng1.cpp
- Inliner.cpp
- Missarg.cpp
- Retref.cpp
- Constarg.cpp