Lab 6: Operator Overloading and Friend Function

Course Code: CSE 4302
Week: 6
Topics: Operator Overloading, Friend Functions, and Type Conversion Operators
Instructors:
Md. Bakhtiar Hasan, Assistant Professor, CSE
Ishmam Tashdeed, Lecturer, CSE
Submission Deadline: 12:00 P.M., December 17, 2025

Upload and Turn In your `.cpp` files to **Lab 6** on **Google Classroom**. If you are unable to finish all the tasks, **submit the tasks you have completed within the given time** (You have the chance to complete the remaining ones individually later on and demonstrate them during the next week's lab evaluation). No late submissions are allowed.
DO NOT forget to rename the files like `XXX_T1.cpp`, `XXX_T2.cpp`, ... (Replace `XXX` with the last three digits of your student ID).
After the submission deadline, you will undergo a one-on-one evaluation by one of the course instructors. The instructor will ask you to show your submissions on Google Classroom. They may ask you to run the code locally and ask you questions about it.

Lab Tasks

1. Gigabyte Gone Wild

As a university student living in a dormitory, managing your limited internet data pack is a daily struggle. You constantly purchase small add-ons, consume data by streaming lectures (or Netflix), and need to know exactly how many megabytes you have left to finish your assignment. You need a system to track this volatility to avoid the dreaded "No Internet" dinosaur game.
Class Specification: Create a class named `DataPack`. The class should have private member variables to store the available data in Gigabytes (floating-point) and the provider name (String). It should also track a boolean status indicating if the data is exhausted.

Implement public getter and setter functions for all member variables, along with sensible error handling. Use `const` member function, Use pass by reference while passing object, make the parameter `const` where it is appropriate.

Implement the following functionalities:

- Constructors to initialize the provider and data amount (defaulting to `0.0`). A destructor to print a "`Connection Terminated`" message.
- A binary '`+`' operator to allow adding to data packs together or adding a float value (extra GB) to a `DataPack` instance.
- A binary '`-`' operator to simulate data consumption. If consumption exceeds the current balance, the balance becomes `0` and the exhausted status is set to `true`.
- A type conversion operator '`int()`' that converts the data pack object into an integer representing the total Megabytes (1 GB = 1024 MB).
- A function to print the current available data of the pack.

Example Usage: The code should allow you to create a `DataPack d1`, add `2.5` GB to it using '`d1 = d1 + 2.5;`', and subtract usage like '`d1 = d1 - 1.2;`'. Furthermore, you should be able to do '`int mb = d1;`' to get the remaining size in Megabytes for precise logging.

2. Game On, Power Up

In a role-playing game (RPG), characters gather experience points (XP) and collect gold. You are writing the logic to handle player progression. You need to compare players to see who is the leader and handle the accumulation of loot after a dungeon raid.

Class Specification: Create a class named `PlayerStats`. Private members must store the username, XP (`long`), and Gold (`int`). Negative values for XP or Gold are not permitted and should be handled during initialization.

Implement public getter and setter functions for all member variables, along with sensible error handling. Use `const` member function, Use pass by reference while passing object, make the parameter `const` where it is appropriate.

Implement the following functionalities:

- Necessary constructors and a destructor with sensible defaults.
- Overload the '`+=`' operator. It should accept another '`PlayerStats`' object (a party merge). It adds the XP and Gold of the argument object to the calling object. The argument object remains unchanged.
- Overload the '`+=`' operator again (overloading function logic) to accept a single integer, which adds to the Gold count only.
- Overload '`==`' to check if two players are at the exact same progression level (requires both XP and Gold to be equal).
- Overload '`!=`' to return the opposite of '`==`'.
- Overload '`>=`' to compare players based solely on XP.

Example Usage: Player '`p1`' defeats a boss. '`p1 += 500;`' adds 500 gold. '`p1`' joins forces with '`p2`', but only '`p1`' keeps the loot: '`p1 += p2;`'. The leaderboard checks: '`if(p1 >= p2)`' to rank them.

3. Going Up?

You are designing the control logic for a smart elevator system in a skyscraper. The elevator tracks its current floor. It needs to move up and down one floor at a time, but it must respect the building's limits (Ground floor 0 to Top floor 100). You also need a way to quickly configure the elevator's start position and print its status to the security console.
Class Specification: Create a class named `Elevator`. Private members include the current floor and a constant integer representing the maximum floor (set via constructor or default).

Implement public getter and setter functions for all member variables, along with sensible error handling. Use `const` member function, Use pass by reference while passing object, make the parameter `const` where it is appropriate.

Implement the following functionalities:

- Necessary constructors and a destructor with sensible defaults.
- Overload the prefix '`--`' operator to move the elevator down one floor. If at floor `0`, it should not move and should print a "`Ground Floor Reached`" warning.
- Overload the postfix '`--`' operator to move down but return the state before the move.
- Overload the '`>>`' operator to input the current floor. If the user inputs a floor > max or < `0`, clamp the value to the nearest valid limit.
- Overload the '`<<`' operator to print "`Elevator is currently at Floor: X`".

Example Usage: '`Elevator e1;`' Security sets pos: '`cin >> e1;`'. It moves down: '`--e1;`'. The system logs: '`cout << e1;`'. The postfix is used for logging previous state: '`Elevator prev = e1--;`'.

4. Fading into the Background (Optional, will not be evaluated)

You are working on an image processing library like Photoshop. You are dealing with pixels, specifically their opacity (alpha channel). Furthermore, you need to be able to dim a pixel by multiplying its opacity by a factor (e.g., 0.5 for 50% transparency). This operation must be commutative.
Class Specification: Create a class named `PixelAlpha`. Private members include three integers for red, green, and blue values (0 to 255). It should also have an integer alpha (0 to 255), where 0 is fully transparent, 255 is opaque.

Implement public getter and setter functions for all member variables, along with sensible error handling. Use `const` member function, Use pass by reference while passing object, make the parameter `const` where it is appropriate.

Implement the following functionalities:

- Necessary constructors and a destructor with sensible defaults.

- A binary '*' operator (member function) that accepts a float factor. It returns a new '`PixelAlpha`' object with the alpha value scaled. (e.g., `200 * 0.5 = 100`). Do not change the colors.
- A binary '`+`' operator to blend two pixels (add their alpha values), capping the result at `255`. Do the same for colors.
- A binary '*' operator (non-member) that accepts a float as the left operand and a '`PixelAlpha`' object as the right operand. This allows the syntax '`0.5 * pixel`'. Do not change the colors.
- Overload '`<<`' to display the red, green, blue, and alpha value.

Example Usage: '`PixelAlpha p1(200);`' Dimming: '`PixelAlpha p2 = p1 * 0.5;`'. Reverse dimming logic: '`PixelAlpha p3 = 0.25 * p1;`'. Blending: '`PixelAlpha p4 = p2 + p3;`'. Output: '`cout << p4;`'.