

## Problem Set 1: Basic C++ Programming

**Note: Try your best to use the appropriate C++ features.**

### Pre-task

Please install vscode C++ environment before the lab:

- **Recommended:** setup vscode on your PC
  - Powerful functions/features
  - Setup vscode C++ on Win11.pdf (refer to L@G)
  - Setup vscode C++ on macOS.pdf (refer to L@G)
- Griffith ELF
  - Easy to setup, but no debugging function.
  - Setup vscode C++ on Griffith ELF.pdf (refer to L@G)

### Tasks

Q1, Define five **char** variables with uniform initialization, then prompt the user to enter five values with appropriate instructions and store them in the five variables, then print the five values.

Define five **int** variables with uniform initialization, then prompt the user to enter five values with appropriate instructions and store them in the five variables, then print the five values.

Then, implement **overloaded functions** to shift the stored values of five variables in a circular manner by a given number of steps (**default** is 1) for char and int.

For example,

c1 c2 c3 c4 c5, with step =1 -> c2 c3 c4 c5 c1

Q2, Implement the same functionality of the circular fashion using a **templated function**.

Q3, Rewrite the function in Q1 to work specifically with **std::array** for both char and int using **template function**.

Define a **std::array** of five **characters** with uniform initialization, then prompt the user to enter five values with appropriate instructions and store them in the array, then print the array.

Define a **std::array** of five **integers** with uniform initialization, then prompt the user to enter five values with appropriate instructions and store them in the array, then print the array.

Then, implement a **template function** to shift the stored values in a circular fashion for the nominated steps (default is 1) for the **std::array** for char and int.

Q4, Reimplement Q3 to work with **std::vector**.

Define a **std::vector** of five **characters** with uniform initialization, then prompt the user to enter five values with appropriate instructions and store them in the vector, then print the vector.

Define a **std::vector** of five **integers** with uniform initialization, then prompt the user to enter five values with appropriate instructions and store them in the vector, then print the vector.

Then, implement a **template function** to shift the stored values in a circular fashion for the nominated steps (default is 1) for the **std::vector** for char and int.

## Problem Set 2: STL Container

**Note: Try to make the best use of appropriate C++ features.**

### Task 1

Write a program that uses iterators to traverse a `vector<int>` and separate the odd and even numbers into two new `vector<int>` containers. Print these three vectors.

### Task 2

Given two `vector<int>`, use iterator and set to merge them and remove duplicates. Then, save it to a new vector and print the result in ascending order.

### Task 3: Investigate `std::vector` Capacity Growth & Shrink Rules

Write a program to analyze how `std::vector` dynamically changes its capacity when elements are added or removed. The goal is to determine the growth factor used by the C++ Standard Library when resizing the `std::vector`

1. Measure Capacity Growth on Insertion
  - 1) Create an empty `std::vector<int>`, and print its size and capacity.
  - 2) Continuously insert elements (e.g., from 1 to 100) using `.push_back()` and print the vector's size and capacity at each step.
  - 3) Identify the pattern of capacity increase (e.g., does it follow a fixed growth factor?).
2. Investigate Capacity Behavior on `erase()`
  - 1) Remove half of the elements using `vector.erase()`
  - 2) Print the size and capacity after the erase operation.
  - 3) Check whether `std::vector` automatically reduces its capacity when elements are removed.
3. Effect of `shrink_to_fit()` on Capacity
  - 1) Call `vector.shrink_to_fit()` after erasing elements.
  - 2) Print the size and capacity again.
  - 3) Does `shrink_to_fit()` always reduce the capacity to exactly match the size?

## Problem Set 3: STL Container

**Note: Try to make the best use of appropriate C++ features.**

The given program generates a vector **V1** containing 10 random **even numbers** in the range **2 to 10**. Using **STL (Standard Template Library)** efficiently, complete the following tasks:

1. **Perform a binary search** in **V1** to check if the number **8** is present.
2. **Modify each element** in **V1**: change the sign to **negative** if the number is **greater than 5**.
3. **Count the number of negative elements** in **V1**.
4. **Remove all negative numbers** from **V1**.
5. Generate two vectors, **V2** and **V3**, each containing 10 random numbers in the range **[1, 20]**. Then, **merge** V2 and V3 into a new vector, **V4**.
6. **Find the minimum and maximum values** in **V4**.
7. **Create a set: S1** from **V4**
8. **Create a map M1** from **S1**, where the key follows the format "**id-index**" (e.g., "id-0"), and the value is the corresponding element from the set.

# 2800ICT Object Oriented Programming

## Problem Set 4 : Classes

**Note: Try to make the best use of appropriate C++ features.**

Design and implement a class `Box<T>` that models a 3D box with dimensions **length**, **width**, and **height**, and supports size comparison, arithmetic operations, and formatted output. This task will reinforce your understanding of constructors, destructors, access control, operator overloading, this pointer, const methods, and class.

Your implementation must demonstrate the following C++ features:

1. Member Variables

At least three **private** member variables: length, width, and height.

2. Constructors

- A **default** constructor that initializes the content to a default value.
- A **parameterized** constructor that accepts a value of type T and uses it to initialize the box.

3. Destructor

Define a destructor that prints a message when the object is destroyed.

4. Access Control

- The member variables must be private.
- Member functions be public.

5. Other Member Functions and const Member Functions

- Design a set of appropriate member functions for your class, and
- carefully consider which functions should be declared as **const** to ensure they do not modify the state of the object.

6. Use of this Pointer

Appropriately use the **this** pointer.

7. Operator Overloading

- `==` : Return true if two boxes have the same volume
- `<` and `>` : Compare boxes based on volume
- `+` and `-` : return sum/difference of the two boxes' volume.
- `<<` : Print in the format, for example, `Box(length=3, width=4, height=5, volume=60)`

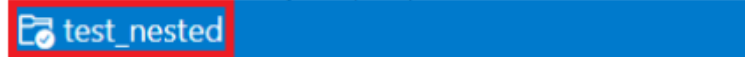
8. Organize your program using three separate files

- `Box.hpp` – Header file: contains the class definition and all function implementations.
- `Box.cpp` – Implementation file (only used if you choose to instantiate the with specific types like `int` or `double`, otherwise leave empty or unused).
- `run_box.cpp` – Main file: contains your `main()` function to test the Box class.

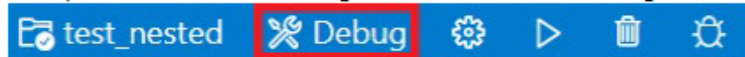
- Compile all files in one folder

1 Select the folder that contains the C/C++ files you want to compile.

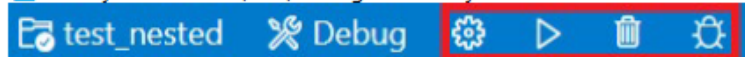
You can select the folder by the quick pick menu from the status bar.







2 Optional: Select either debug or release mode for building the binary (debug is the default case).



3 Now you can build/run/debug the binary.



-  Build: This task will compile all C/C++ files in the selected folder and will link them into a binary.
-  Run\*: This task will execute the built binary.
-  Clean\*: This helper task will delete all files in the build dir.
-  Debug\*: This task will start a debugging session for the binary.

## Week 5 - Problem Set

**Note: Try to make the best use of appropriate C++ features.**

Implement the classes Rectangle for the given UML.

Requirement: Object should maintain its own independent space and must not share data with other objects. The operator<< should output the object's information, including its top-left coordinates (i.e., x and y), width, and length.

Then, write a main function to test the copy constructor, move constructor, copy assignment, and move assignment.

The operator<< function checks whether rect.x and rect.y are valid before dereferencing them. If either pointer has been moved and is nullptr, it outputs "null, null" instead of attempting to dereference them.

Rectangle
<pre>- x: shared_ptr&lt;double&gt; - y: shared_ptr&lt;double&gt; - width: double - length: double</pre>
<pre>+ Rectangle(): + Rectangle(x: double, y: double, w: double, l:double): + Rectangle(other: const Rectangle&amp;): + Rectangle(other: Rectangle&amp;&amp;): + operator=(other: const Rectangle&amp;): Rectangle&amp; + operator=(other: Rectangle&amp;&amp;): Rectangle&amp; + setWidth(w: double): void + setLegnth(l: double): void + &lt;&lt;friend&gt;&gt; operator&lt;&lt;(out: ostream&amp;, other: const Rectangle&amp;): ostream&amp;</pre>

## Week 6 - Problem Set

**Note: Try to make the best use of appropriate C++ features.**

Implement the classes for the given UML.

The print() function is responsible for printing the details and the salary. StaffMember's << should call its print().

Details:

Volunteer::pay() — Returns 0.0 since volunteers are unpaid.

Volunteer::print() — Prints "Volunteer -", basic personal info, and salary.

Employee::pay() — Returns the fixed salary (pay\_rate).

Employee::print() — Prints personal info and social security number (SSN).

Hourly::pay() — Returns pay\_rate \* hoursWorked as salary.

Hourly::addHours() — Adds additional hours worked.

Hourly::print() — Prints "Hourly -", employee info, hourly rate, hours worked, and calculated salary.

Executive::pay() — Returns pay\_rate + bonus as total salary.

Executive::awardBonus() — Increases the executive's bonus.

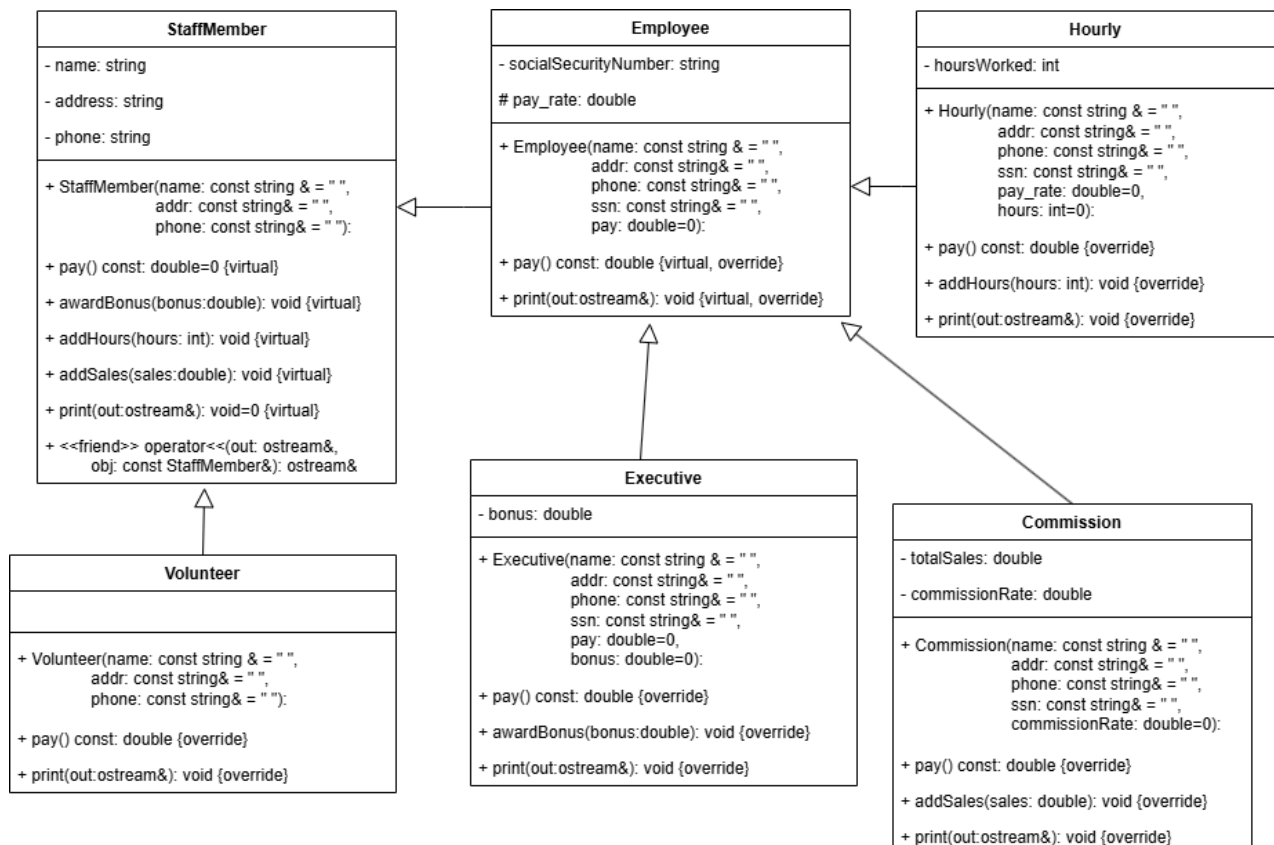
Executive::print() — Prints "Executive -", employee info, pay rate, bonus, and calculated salary.

Commission::pay() — Returns totalSales \* commissionRate as salary (note: fixed pay\_rate is not added here).

Commission::addSales() — Adds additional sales to the employee's total.

Commission::print() — Prints "Commission -", employee info, total sales, commission rate, and calculated salary.

The attached main function is used to test your implementation.



The output should be like below:

```
Name: Paulie, Address: 456 Off Line, Phone: 555-0101, SSN: 987-65-4321
Volunteer - Name: Adrianna, Address: 987 Babe Blvd., Phone: 555-8374, Pay: 0
Hourly - Name: Michael, Address: 678 Fifth Ave., Phone: 555-0690, SSN: 958-47-3625, Pay Rate: 10.55, Hours Worked: 25, Pay: 263.75
Commission - Name: Michael, Address: 678 Fifth Ave., Phone: 555-0690, SSN: 958-47-3625, Total Sales: 225, Commission Rate: 0.1, Pay: 22.5
Executive - Name: Tony, Address: 123 Main Line, Phone: 555-0469, SSN: 123-45-6789, Pay Rate: 1023.07, Bonus: 345, Pay: 1368.07
```



## Problem Set

**Note: Try to make the best use of appropriate C++ features.**

In this lab, you will develop a simple Book Inventory Management System using C++. During the implementation process, you will practice several modern C++ features, especially:

- Exception handling (standard and custom)
- Stack unwinding
- Function objects (functors)
- Anonymous function objects
- Lambda expressions

The system should manage multiple book objects, support operations such as selling books, restocking, and inventory querying, and handle illegal operations using appropriate exceptions. Then, use the provided main function to test your implementation and verify that all features work as expected!

### Tasks:

#### 1. Implement a Book Class

Create a class named Book with:

- Private members: title, author, and stock
- A constructor that initializes these attributes
- A sell(int quantity) method:
  - Throws std::invalid\_argument if quantity is non-positive
  - Throws a custom OutOfStockException if requested quantity exceeds available stock
- A restock(int quantity) method:
  - Increases the stock
  - Throws std::invalid\_argument if the quantity is non-positive
- Getter functions: getStock() and getTitle()
- A destructor that prints a message (for observing stack unwinding)

#### 2. Define a OutOfStockException Class

Create a custom exception class that:

- Inherits from std::exception
- Accepts the book title in the constructor and builds an informative error message
- Stores the message as a private std::string
- Overrides what() to return the message as a const char\*

#### 3. Implement a LowStockChecker Functor

Define a function object class that:

- Takes a stock threshold as a constructor parameter
- Implements operator() to return true if a book's stock is below the threshold

#### 4. Implement an Inventory Class

Create an inventory manager class that:

- Stores a list of books (`std::vector<Book>`)
- Provides methods to:
  - `addBook(Book)` to add new books
  - `sellBook(title, quantity)` to sell books by title with error handling if not found
  - `restockBook(title, quantity)` to restock books
  - `findBook(title, Book&)` to safely search for a book by title
    - Returns true if found, otherwise false
    - Fills the provided reference parameter with a copy of the found book
  - `applyToAllBooks(function)` to apply a function to every book (using functors or lambdas)
  - `listLowStockBooks(threshold)` to print all books below a stock threshold

## Problem Set 8

**Note: Try to make the best use of appropriate C++ features.**

### Single-thread vs Multithreaded Performance Comparison

This lab aims to demonstrate how multithreading can divide a computation task into multiple subtasks executed in parallel to significantly improve efficiency.

In this lab, you will process a large array of integers to **count the number of even numbers**, first with a **single-threaded** approach, then using a **multithreaded** approach, and finally compare the performance results.

#### Objectives:

- Learn the basics of multithreaded programming (std::thread, std::ref, join, etc.).
- Understand data partitioning and independent thread computation.
- Learn how to aggregate partial results from multiple threads.
- compare single-threaded and multithreaded performance and compute speedup.

#### Tasks:

Given a **600,000,000-element** (600 million) 1D array:

1. **Single-threaded Version:** size\_t countEvenSingleThread(data)
  - Traverse the entire array using a single thread to count the number of even numbers.
2. **Multithreaded Version:** size\_t countEvenMultiThread(data, numThreads)
  - Launch **n threads**.
  - Evenly divide the array into **n** segments, with each thread independently processing one segment.
  - Each thread stores its partial count in a local variable.
  - After all threads complete, aggregate the partial counts to obtain the total number of even numbers.
3. **Performance Comparison:**
  - Use the provided main function to test both implementations and evaluate the performance differences by different dataSize and numThreads.

## Problem Set 9

**Note: Try to make the best use of appropriate C++ features.**

You are provided with a working implementation of a SkipList that stores integer keys. Your task is to extend this implementation to support generic types using C++ templates and then use it to develop a simple LogSystem that manages log entries with integer timestamps and string messages.

### Objective:

- Modify the given **SkipList** and **Node** classes to support **generic types using C++ templates**
- It should support **SkipList<Key, Value>**, for example **SkipList<Int, String>**
- Using your SkipList<int, std::string> implementation, create a LogSystem class to efficiently store and manage logs.
- The LogSystem should at least support the following operations:
  - void insert(int timestamp, const std::string& message);  
Inserts a new log entry into the system. The entry is indexed by the given timestamp and stores the associated message. If the timestamp already exists, you may choose to overwrite or ignore it (design-dependent).
  - std::string search(int timestamp) const;  
Searches for a log entry with the specified timestamp.
    - Returns the corresponding message if found.
    - Returns "NOT FOUND" if the timestamp does not exist in the log system.
  - bool remove(int timestamp);  
Removes the log entry with the given timestamp from the system.
    - Returns true if the entry existed and was successfully deleted.
    - Returns false if the timestamp was not found.
  - void display() const;  
Displays all log entries in ascending order of timestamps.
    - Each entry should include the timestamp and its corresponding message.
- Use the provided main function to test your implementation.

## Problem Set 10

**Note: Try to make the best use of appropriate C++ features.**

To become familiar with the structure and properties of a Treap by implementing a function that visually represents its hierarchical form. This exercise aims to reinforce your understanding of the dual properties of Treaps:

- The **binary search tree (BST) property**: left children are less than the parent, right children are greater.
- The **heap property**: the priority of each node is higher than that of its children.

### Objective:

In this task, you are required to implement a function named `print_treap_2D` based on the provided Treap class.

The goal of this function is to **visually print the structure of the Treap** in a human-readable, tree-like 2D format. The printed output should clearly represent the **hierarchical relationships between nodes**, showing how nodes are connected as parent and children in the binary structure.

The output format should resemble the following example:

