Exercise session 06

The Standard Template Library, smart pointers and move semantics.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

11 Nov 2025

Exercise 1: Monte Carlo estimate of π

In this exercise, you will perform a Monte Carlo simulation to estimate π.

1. Consider the square $[0, 1]^2$ and the quarter-circle centered at $(0, 0)$ with radius 1.
2. Generate random points within the square.
3. Count how many of these random points fall within the quarter-circle ($x^2 + y^2 \leq 1$).
4. After generating a sufficient number of random points, you can estimate

$$\pi \approx \frac{4 \cdot \text{Number of points inside the quarter-circle}}{\text{Total number of generated points}}.$$

To improve estimation accuracy, try increasing the number of random points in your simulation.

Exercise 2: `std::pair`

Create a `MyPair` class template that mimics `std::pair`, holding two elements of any type. Implement member variables `first` and `second`, with constructors for default and custom initialization. Overload `operator==`, `operator!=`, and `operator<` for equality and lexicographical comparison. Test your implementation using various data types such as `int`, `double`, and `std::string`.

Note: Lexicographical comparison means comparing first elements first; if equal, then compare second elements.

Exercise 3: `std::set` (1/2)

In a building security system, door locks are opened by entering an access code into a keypad. The access code's validation process is handled through an `Access` object with the following interface:

```
class Access
{
public:
    void activate(unsigned int code);
    void deactivate(unsigned int code);
    bool is_active(unsigned int code) const;
};
```

Each employee is assigned a unique access code, which can be activated using the `activate()` function. When an employee leaves the company, their access code can be deactivated using the `deactivate()` function.

Exercise 3: `std::set` (2/2)

Your task is to implement the `Access` class as described above. Write a test program that accomplishes the following tasks:

1. Create an instance of the `Access` object.
2. Activate the access codes 1234, 9999, and 9876.
3. Prompt the user to enter an access code, and read the code from the console.
4. Inform the user whether the door opens successfully.
5. Repeat the last two steps until the door successfully opens.
6. Deactivate the code that worked (the one entered by the user that opened the door). Also, deactivate the code 9999 and activate the code 1111.
7. Repeat steps 3 and 4 until the door successfully opens.

Exercise 4: `std::map` (1/2)

In the previous exercise, the customer using the security system wants to associate an access level with each access code. Users with higher access levels should be able to open doors to more security-sensitive areas of the building compared to users with lower access levels. Start with your solution from the previous exercise and make the following modifications to the `Access` class:

```
class Access
{
public:
    void activate(unsigned int code, unsigned int level);
    void deactivate(unsigned int code);
    bool is_active(unsigned int code, unsigned int level) const;
};
```

The `is_active()` function should return `true` if the specified access code has an access level greater than or equal to the specified access level. If the access code is not active at all, it should return `false`.

Exercise 4: `std::map` (2/2)

Now, update the main program to perform the following tasks:

1. After creating an instance of the `Access` object, activate code 1234 with access level 1, code 9999 with access level 5, and code 9876 with access level 9.
2. Prompt the user to enter an access code, and read the code from the console.
3. Assuming a door requires access level 5 for entry, print whether it opened successfully.
4. Repeat the last two steps until the door opens.
5. Deactivate the code that worked. Then, change the access level of code 9999 to 8 (hint: reactivate it with the new level), and activate code 1111 with access level 7.
6. Prompt the user for an access code, read the code from the console.
7. Assuming a door requires access level 7 for entry, print whether it opened successfully.
8. Repeat the last two steps until the door opens.

Exercise 5: STL containers and algorithms

1. Create a vector named `random_numbers` that contains 100 random integers between 0 and 9.
2. Create a new vector named `sorted_numbers` by sorting the `random_numbers` vector in ascending order (keep duplicates).
3. Create a new vector named `sorted_unique_numbers` by sorting the `random_numbers` vector (remove duplicates).
4. Create a new vector named `unsorted_unique_numbers` by printing unique entries from the `random_numbers` in the same order they appear, without repetitions.

Hints: Use `std::sort()` for sorting. Use `std::unique()` combined with `erase()` for removing consecutive duplicates. For task 4, consider using `std::unordered_set` to track seen elements.

Exercise 6: Word frequency analysis

The file `input.txt` contains a list of random complete sentences in English. Develop a C++ program that reads the file, calculates the frequency of each word in the text, and outputs the word-frequency pairs to a new file in a dictionary format.

Write a C++ program to process the input text file by splitting it into words and counting the occurrences of each unique word. Spaces and punctuation should be discarded. Convert words to lowercase for case-insensitive counting.

The program should generate a new file (named `output.txt`) containing the word-frequency pairs in a dictionary format. Each line in the output file should consist of a list of entries of the form `word: frequency`.

Bonus: sort the output by frequency, in descending order. If two words have the same frequency, then sort them alphabetically.

Exercise 7: Move semantics for efficient data transfers

Define a class `Vector` that represents a one-dimensional vector of double values, stored as a raw pointer `double *data`, along with its `size`. Implement constructors, the copy constructor, the copy assignment operator, and the destructor.

1. Implement a **move constructor** that transfers ownership of the underlying data from the source vector to the destination vector. The move constructor should ensure that the source vector's data is no longer accessible after the transfer.
2. Define a **move assignment operator** that allows for the efficient transfer of ownership of the underlying data from one Vector object to another. Similarly, the move assignment operator should ensure that the source vector's data is no longer accessible after the transfer.
3. Compare the performance of copying and moving large vectors. Measure the time taken to copy and move vectors by increasing the input size from 2^{20} to 2^{30} elements. Analyze the performance gain achieved by using move over copy semantics.

Exercise 8: Smart pointers with polymorphism

Create a C++ program demonstrating smart pointers with polymorphism:

1. Create a base class `Base` with a method `virtual void display() const` and a virtual destructor.
2. Create two derived classes `Derived1` and `Derived2` that override `display()`.
3. Use smart pointers to manage instances of `Base` that actually point to `Derived` objects. Demonstrate:
    i. Creating `std::unique_ptr<Base>` pointing to `Derived` objects;
    ii. Creating `std::shared_ptr<Base>` pointing to `Derived` objects;
    iii. Transferring ownership of `std::unique_ptr` (using `std::move`);
    iv. Sharing ownership with `std::shared_ptr` (observe reference counting);
    v. Polymorphic behavior through virtual dispatch, i.e., by calling `display()` through the smart pointers.