The Standard Template Library (STL) I

Raul P. Pelaez

March 19, 2026

Contents

1 The STL

Warning

For this lesson, I am expecting you to use the C++ documentation to look up the functions and classes you need. I will not be providing the exact syntax for each function, but I will give you the general idea of what you need to do.
Trust cppreference with your life. Avoid cplusplus, geeksforgeeks is cool, but easily outdated so always check the date of the article and take it into account.

The C++ standard provides a framework for dealing with data as sequences of elements, called the Standard Template Library (STL).
The STL offers a collection of ready-to-use classes and functions, which are generic and can work with any data type. It includes:

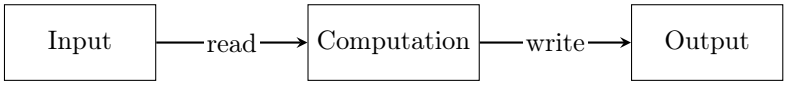
- Containers: Objects that store data (e.g., `std::vector`, `std::list` and `std::map`).
- Iterators: Objects that point to elements within containers.
- Algorithms: Functions that perform operations on containers (e.g., `std::sort`, `std::find` and `std::accumulate`).

There are two major parts to computing: the computation and the data. Sometimes we focus on the computation and talk about `if`-statements, loops, functions, etc. At other times we focus on the data and talk about arrays, vectors, strings, files, etc. However, to get useful work done we need both.
The STL provides abstractions that separate both. For instance, we do not care if we are sorting an `std::vector` or an `std::array` or a range given by a couple of pointers. The computation to be performed is exactly the same.

What would we like to do with "a lot of data"? Let me give you some examples:

- Sort the words in a dictionary in order
- Find the information of a person in a database given their name
- Find the highest temperature for each day in the last millennium
- Count the number of occurrences of a given word on the web.

Note that we can describe these tasks without actually mentioning how the data is stored. Clearly we must be dealing with something like lists, vectors, files, input streams, etc. for these tasks to make sense, but we do not need to know the details about how the data is stored to talk about what to do with it. What is important is the type of the values or objects (the element type), how we access those values or objects and what we want to do with them.



These kinds of tasks are very common, and we would like to write code performing such tasks simply and efficiently. On the other hand, the problems for us as programmers are:

- There are infinite data types
- There are many ways to store data
- There are many tasks we might want to perform on collections of data

The STL focus on the commonalities of these tasks and gives us ways to "sort a range of elements" (without depending on how the elements are stored) or "store a collection of unique elements (a set)" (without depending on what task we want to perform on the data).

1.1 Key new concepts

Info

The STL
The Standard Template Library is a collection of tools in the standard library.

- Provides a collection of classes and functions for working with data.
- Consists of containers (storage), iterators (access, traversal), and algorithms (tasks/operations).

2 Containers

2.1 Sequential Containers

Sequential containers maintain the ordering of elements and allow sequential access. The most common sequential container is `std::vector`, which is a dynamic array that can change size. Other sequential containers include `std::list` (doubly-linked list) and `std::deque` (double-ended queue).

Info

Sequential Containers

- `std::vector`: A variable-size container.
- `std::list`, `std::forward_list`: A doubly-linked and single list respectively.
- `std::deque`: A double-ended queue.

There are also more specialized containers, like `std::array` which is a fixed-size array, or `std::string` which is a container of characters.

2.1.1 Common operations

The good thing about the STL is that it provides a common interface for all containers. For instance, all containers have a `begin` and `end` function that returns iterators (funny pointers) to the beginning and end of the container. This means that we can write a function that somehow operates on a container without knowing if it is a `std::vector` or a `std::list` or a `std::deque`.

Info

Common operations (partial)

- `begin`, `end`: Return iterators to the beginning and end of the container.
- `size`: Returns the number of elements in the container.
- `empty`: Returns true if the container is empty.
- `reserve`: Increases the capacity of the container.
- `insert`: Inserts elements into the container.
- `erase`: Removes elements from the container.
- `at`: Accesses the element at a given position, with bounds checking.
- `find`: Searches for an element in the container.

The capacity of a container is the number of elements it can hold without needing to allocate more memory. When the capacity is reached, the container may need to reallocate memory, which can be expensive. By using `reserve`, you can preallocate memory to avoid reallocations.

Advice

Each container has its own properties that make it suitable for different use cases. For example, `std::vector` is good for random access and appending elements, while `std::list` is good for frequent insertions and deletions in the middle of the sequence.
The continuity of vector makes it so that if one deletes or inserts an element in the middle of the vector, all the elements after it must be moved. This is not the case for list, where the elements are stored in a linked list and only the pointers need to be updated.

Info

Most containers will store more elements than they currently have to avoid reallocations. This is called the "capacity" of the container. When we call push_back on a vector, it will check if it has enough capacity to store the new element. If it does not, it will allocate more memory and copy the elements to the new memory. Normally, it will double the capacity of the vector instead of just adding one element, making future insertions faster for a while.

2.2 Associative Containers

After `std::vector`, the most useful container is probably `std::map`. It is an ordered sequence of (key, value) pairs in which you can look up a value based on a key (like a dictionary in Python), for instance `age["Alice"]` could store the age of Alice. Closely following we have `std::unordered_map`, which is simply a map optimized for keys that are strings. Data structures similar to these are known under many names, such as hash tables, red-black trees, associative arrays, etc. In the STL they are called associative containers.
The STL provides eight associative containers:

Info

Associative Containers

- `map`: An ordered container of key-value pairs.
- `set`: An ordered container of unique keys.
- `unordered_map`: An unordered container of key-value pairs.
- `unordered_set`: An unordered container of unique keys.
- `multimap`: A `map` where a key can occur multiple times.
- `multiset`: A `set` where a key can occur multiple times.
- `unordered_multimap`: An `unordered_map` where a key can occur multiple times.
- `unordered_multiset`: An `unordered_set` where a key can occur multiple times.

Associative containers offer many of the same operations as sequential containers, but they are optimized for fast lookups based on keys. For example, `find` in a `std::map` is much faster than searching through a `std::vector`. The trade-off is that some other operations become slower, like inserting elements. Additionally, each associative container has a unique set of properties and methods that make it suitable for different use cases.
All of them, however, provide a begin and end function, which makes them compatible with the STL algorithms, as well as range-based for loops.

2.3 Container Adapters

The STL also provides some adaptors, which are containers that provide a specific interface using an underlying container. Look them up in the documentation if you need them.

- `std::stack`: A Last-In-First-Out (LIFO) data structure.

- `std::queue`, `std::priority_queue`: First-In-First-Out (FIFO) and priority queue data structures.

3 Exercises

3.1 Zipf's law

Goal

For a not very well known reason, when a list of measured values is sorted in decreasing order, the value of the nth entry is often approximately inversely proportional to n. This is known as Zipf's law. The best known instance of Zipf's law applies to the frequency table of words in a text or corpus of natural language

$$\text{word frequency} \propto \frac{1}{\text{word rank}}$$

In other words, it is usually found that the most common word occurs approximately twice as often as the next common one, three times as often as the third most common, and so on. So, if we compute, for a given text, the frequency of each word and sort them in decreasing order, we should see a straight descending line in a log-log plot when plotting against the rank. The rank is just the position of the word in the sorted list of words
To showcase this law, we will be investigating the book Moby Dick; Or, The Whale by Herman Melville. Take the txt containing the book from BB and put it in your current folder.

Milestone

Reading the text
Write a function that takes the filename as argument and returns an `std::vector<char>` containing only the alphabetic characters in the file. Replace any non alphabetic characters with a space.
You can use this as a skeleton.

Source code

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>
#include <map>
#include <set>
std::vector<char> readBook(std::string fileName);
```

You are not allowed to include anything other than these headers (I do not want you to use STL algorithms, like sort, for today's lesson).

hint

- Use the `std::isalpha` and `std::isspace` functions.
- Files streams have a get function that will give you the next character in the file.

Milestone

Counting the words
Write a new function that takes the characters in the book and returns a map, containing the number of times each word appears in the book. The function should have the following syntax:

Source code

```
std::map<std::string, int> computeWordFrequency(const std::vector<char>&
↪  book);
```

hint

- Use a `std::stringstream` to read words from the `std::vector<char>`.

Milestone

Unique words
Count the number of unique words in the book, you can use the map from the previous milestone, but instead, take the vector of chars and write a function that uses an `std::set`.

Source code

```
int countUniqueWords(const std::vector<char>& book);
```

Milestone

Sorting the frequencies
Use a `std::multimap<int, std::string>` to store the frequencies and words. The key will be the frequency and the value the word. We will be using the fact that a multimap is sorted by the key, and that it allows duplicate keys (many words can have the same frequency). Write a function that takes the map from the second milestone and returns the sorted multimap.

Source code

```
std::multimap<int, std::string> sortFrequencies(const
↪  std::map<std::string, int>& frequencies);
```

hint

- You will need to use a custom comparator to sort the map in decreasing order, look for `std::greater<>`. See also the multimap documentation.

Milestone

Plotting the results
Write a function that outputs the contents of the multimap to a file in the format "rank freq word". Use some external tool to plot this file in log-log scale.

hint

- Use a range-based for loop to iterate over the multimap.
- The rank is just the position of the word in the sorted list of words, you can use a counter for this.
- Remember the most used word should have rank 1, not 0, for it to be visible in a log-log plot.

Advanced milestone

Use structured-binding to iterate over the multimap, avoiding temporary variables.
Structured-binding is a C++17 feature that allows you to unpack tuples and other structured types, for instance:

Source code

```
std::pair<int, int> p = {1, 2};
auto [a, b] = p;
```

Advanced milestone

Plot directly from C++ using some external library.

hint

- Do not underestimate this one. It will give you immense experience, but it will come with a price.
- Check a couple of options: Matplot++, Sciplot. You will not believe how those are installed and integrated into your project ;P.

Milestone

The Hápax legómenon
In a corpus of text, a hapax legomenon is a word that occurs only once. Use the multimap from a previous milestone and print to terminal the number of words that occur only once and some of them.

Milestone

Look for another book or large text-corpus in your own language (if its english look for something exotic, like old english or some author that has a very particular style, like Lovecraft). Non ASCII alphabets (like Arabic or Cyrillic) will probably require you to move away from "char" into some UTF-8 aware type, like `wchar_t`.
Plot it on top of the Moby Dick curve and marvel.

hint

- Resort back to english if you start getting dizzy with non-ASCII. String parsing gets real crazy real fast.
- Arabic has a more involved definition of what a "space" is that is not fully covered by `std::isspace`, you will need to add some extra checks for that.
- For non-ASCII you will probably need to use `ostream::imbue` and `std::locale` to set the locale to the correct one. I am but an ASCII peasant, so this is kind of uncharted territory for me.