The Standard Template Library (STL) II

Raul P. Pelaez

October 31, 2024

Contents

1 Introduction

In this lesson, we'll explore two fundamental components of the C++ Standard Template Library (STL): iterators and algorithms. Building upon our understanding of containers, we'll see how iterators provide a unified way to access elements, and how algorithms perform operations on these elements efficiently.

2 Iterators

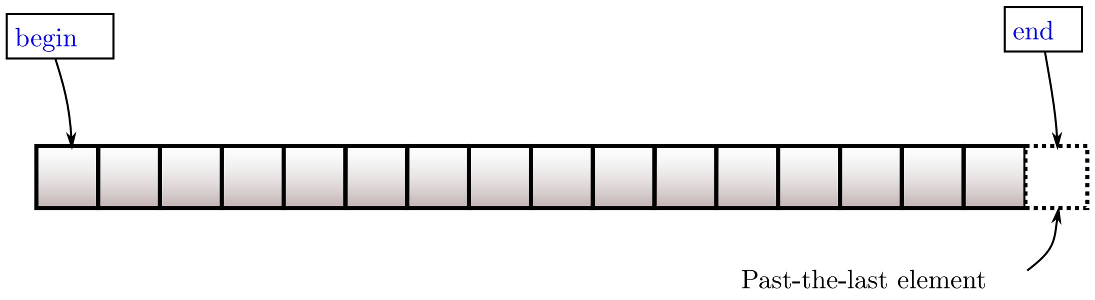
Iterators act as pointers that navigate (or traverse) through the elements of a container. They provide a common interface to access container elements without exposing the underlying structure.

Info

From the C++ ISO standard:

> Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators.

All STL containers provide iterators to access their elements, allowing us to write generic code that works with any container type.
We can request iterators from containers using member functions like `begin()` and `end()`. These functions return iterators pointing to the first and one-past-the-end elements, respectively.



Info

Iterator kinds
Iterators are classified into different categories based on their capabilities:

1. Input Iterators: Read from a sequence.
2. Output Iterators: Write to a sequence.
3. Forward Iterators: Read/write access, move forward.
4. Bidirectional Iterators: Move forward and backward.
5. Random-Access Iterators: Directly access any element.

Going down this list, each iterator type is also an iterator of the previous type (so a random-access iterator is also bidirectional).
Different containers provide different iterator types based on their capabilities.
For instance, `std::vector` provides random-access iterators, while `std::list` provides only bidirectional iterators, and `std::forward_list` provides only forward iterators.

Source code

```cpp
#include <vector>
#include <iostream>
int main(){
  std::vector<int> v = {1, 2, 3, 4, 5};
  // Get an iterator to the first element
  std::vector<int>::iterator it = v.begin(); // One normally uses auto here
  // Dereference the iterator to access the element
  std::cout << *it << std::endl; // Prints 1
  // Move to the next element
  ++it;
  std::cout << *it << std::endl; // Prints 2
  // Move to the previous element
  --it;
  std::cout << *it << std::endl; // Prints 1
  // Going over all elements
  for(auto it = v.begin(); it != v.end(); ++it){
    std::cout << *it << std::endl;
  }
  // Equivalent to:
  for(auto &x: v){
    std::cout << x << std::endl;
  }
}
```

Info

Iterator Operations

- Dereferencing: `*it` accesses the element.
- Incrementing: `++it` moves to the next element.
- Decrementing: `--it` moves to the previous element (if supported).
- Arithmetic: `it + n, it - n` (random-access iterators).

All access operations of a container can (and often are) implemented using iterators. For example, the `std::vector` class provides the `operator[]` to access elements, but this operator is implemented using iterators, in a way similar to the following:

Source code

```cpp
template <typename T>
T& std::vector<T>::operator[](size_t index) {
  return *(begin() + index);
}
```

Advanced

C++20 iterator Concepts
The <`iterator`> header comes with a family of concepts that define the requirements for different kinds of iterators. These concepts are used to constrain template parameters to only accept iterators that meet the specified requirements. For instance, this is a function that works only for random-access iterators:

Source code

```cpp
#include <vector>
#include <list>
#include <iterator>
template <std::random_access_iterator It>
void foo(It begin, It end){
  // Do something with the iterators
}
int main(){
  std::vector v = {1,2,3};
  foo(v.begin(), v.end());
  std::list l = {1,2,3};
  //foo(l.begin(), l.end()); // Error: list<int>::iterator does not
  ↪  satisfy std::random_access_iterator
}
```

Warning

Iterators are NOT pointers
Although iterators behave like pointers in many ways by design, they are not pointers necessarily. They are objects that encapsulate the logic to navigate through a container's elements. An iterator allows the same operations as a pointer (dereferencing, incrementing, etc.), but it may not be implemented as a pointer.
For instance, imagine that we need a list of integers with numbers 0 to N. We could store these numbers in a vector, which takes memory. Instead, we could use an iterator to provide these numbers on the fly, in this case that is known as a counting iterator. We could implement it as follows:

Source code

```
#include <numeric>
#include <iostream>
// A counting iterator that can be used with the STL algorithms
template<class T=int>
class counting_iterator {
  T current;
public:
  counting_iterator(T start = 0) : current(start) {}
  const T& operator*() const { return current; }
  // Makes this a forward iterator
  counting_iterator& operator++() { ++current; return *this; }
  // Makes this a bidirectional iterator
  counting_iterator& operator--() { --current; return *this; }
  // Makes this a random-access iterator
  counting_iterator operator+(T n) const {
    return counting_iterator(current + n);
  }
  counting_iterator operator-(T n) const {
    return counting_iterator(current - n);
  }
  T operator[](std::size_t n) const { return current + n; }
  // Comparison operator, required to know when to stop iterating
  bool operator==(const counting_iterator& other) const {
    return current == other.current;
  }
  bool operator!=(const counting_iterator& other) const {
    return current != other.current;
  }
};
int main(){
  counting_iterator it;
  std::cout<<(*it)<<'\n'; // Returns 0
  std::cout<<it[5]<<'\n'; // Returns 5
  std::cout<<it[12334455]<<'\n'; // Returns 12334455
  int res = std::accumulate(it, it + 1000, 0);
  std::cout << res << '\n'; // Returns 499500
  ++it; // Advance the iterator
  std::cout<<(*it)<<'\n'; //Returns 1;
}
```

This class behaves very closely like a pointer if we do not look at the implementation.
Dereferencing the iterator would return the current number, and incrementing the iterator would move to the next number. This way, we can provide a sequence of numbers without storing them in memory.

Advice

Fancy iterators
Even with their usefulness, as you have seen writing a "fancy" iterator comes with a lot of boilerplate code. One rarely writes iterators from scratch. Instead, one uses the iterators provided by the STL or other libraries.
The STL does not provide a lot of iterator types, but many libraries do. For instance, the Boost library provides a lot of iterator types, such as the counting iterator shown above. These iterators can be used with the STL algorithms to provide a lot of functionality without having to write a lot of code.
Once you start thinking in iterators, you'll see that many problems can be solved with them, and you'll start to appreciate the power of the STL.
The GPU programming library Thrust, for instance, provides a lot of fancy iterators that can be used with the STL algorithms to run code on the GPU. This is a very powerful feature that allows you to write code that runs on the GPU with minimal changes. However, this library is a bit awkward to integrate if your project is not GPU-oriented.

Advanced

Ranges in C++20
In C++20, the concept of ranges was introduced. A range is a pair of iterators that define a sequence of elements. This concept is used in many algorithms to specify the range of elements to operate on. For instance, the `std::accumulate` function takes a range of elements to sum. This is a more general concept than the pair of iterators, as it allows for more flexibility and better error checking.
With it, C++20 introduced the `std::views` namespace, which provides some fancy iterator types that can be used with the STL algorithms. For instance, the `std::views::iota` provides essentially the same functionality as the counting iterator above.

Source code

```cpp
#include <numeric>
#include <iostream>
#include <ranges>
int main(){
  // A list of numbers from 0 to 9 that requires no memory
  for(int i: std::views::iota(0, 10)){
    std::cout << i << std::endl;
  }
}
```

Ranges is a very recent addition to the language and is not yet widely used. However, it is expected to become the standard way to work with sequences in C++. Alas, it is such a large, complex and advanced topic that we will probably not be able to cover it in this course.

3 Algorithms

The STL provides a rich set of algorithms that perform operations like searching, sorting, and modifying data. These algorithms are template functions that operate on ranges defined by iterators.

Algorithms typically take iterators as arguments:

Source code

```
std::algorithm_name(start_iterator, end_iterator, other_parameters);
```

Info

Combining Iterators and Algorithms
For example, the `std::accumulate` algorithm (in the `<numeric>` header) calculates the sum of elements in a range:

Source code

```
std::vector v = {1, 2, 3, 4, 5};
int init_value = 0;
int sum = std::accumulate(v.begin(), v.end(), init_value);
// sum = 1 + 2 + 3 + 4 + 5 = 15
```

Info

There are quite a bunch of them, check the documentation for a complete list. Some of the most common ones are:

- `std::accumulate`: Sums the elements in a range.
- `std::sort`: Sorts the elements in a range.
- `std::find`: Finds an element in a range.
- `std::copy`: Copies elements from one range to another.
- `std::transform`: Applies a function to each element in a range.
- `std::for_each`: Applies a function to each element in a range.
- `std::count`: Counts the number of elements in a range.

Advice

Why would you use `std::for_each` instead of a loop?

- It's more expressive: The algorithm name tells you what it does.
- It's more flexible: You can pass a lambda function to it.
- It's more efficient: The algorithm can be optimized for the specific container by the implementors of the STL.

For me, the most important reason is that "thinking in algorithms" normally results in much more efficient code and, critically, allows you to pull tricks like this:

Source code

```
#include <algorithm>
#include <vector>
#include <tbb/parallel_for_each.h>
#include <thrust/for_each.h>
#include <thrust/device_vector.h>
int main(){
  std::vector<int> v = {1, 2, 3, 4, 5};
  // A sequential version
  std::for_each(v.begin(), v.end(), [](int& x){ x *= 2; });
  // A CPU parallel version using TBB
  tbb::parallel_for_each(v.begin(), v.end(), [](int& x){ x *= 2; });
  // A GPU version using Thrust
  thrust::device_vector<int> dv = v; // A vector that uses GPU memory
  // This will run on the GPU
  thrust::for_each(dv.begin(), dv.end(), [] __device__ (int& x){ x *= 2;
  ↪  });
}
```

With minimal changes (essentially changing the namespace), you can run the same code on the CPU in parallel or on the GPU. This is the power of thinking in algorithms, and of the STL.

Advanced

C++17 Parallel Algorithms
In C++17, the STL introduced parallel versions of some algorithms. Algorithms offer an overload with a first argument called a "policy" that specifies the execution policy. The policy can be sequential, parallel, or parallel with a specific number of threads. For instance, in order to sort a vector in parallel:

Source code

```cpp
#include <algorithm>
#include <execution>
#include <vector>
int main(){
  std::vector<int> v = {5, 4, 3, 2, 1};
  std::sort(std::execution::par_unseq, v.begin(), v.end());
}
```

4 Exercises

Goal

Repeat the Zipf's law exercise from last lesson, but this time, instead of using only containers (like `std::map`), use iterators and algorithms to solve it.

- You are only allowed to use the `std::vector` as a container.
- Reuse the function you wrote to read the file.

hint

For each of the individual problems that you have to solve, check the list of algorithms in the documentation to see if there is one that can help you.