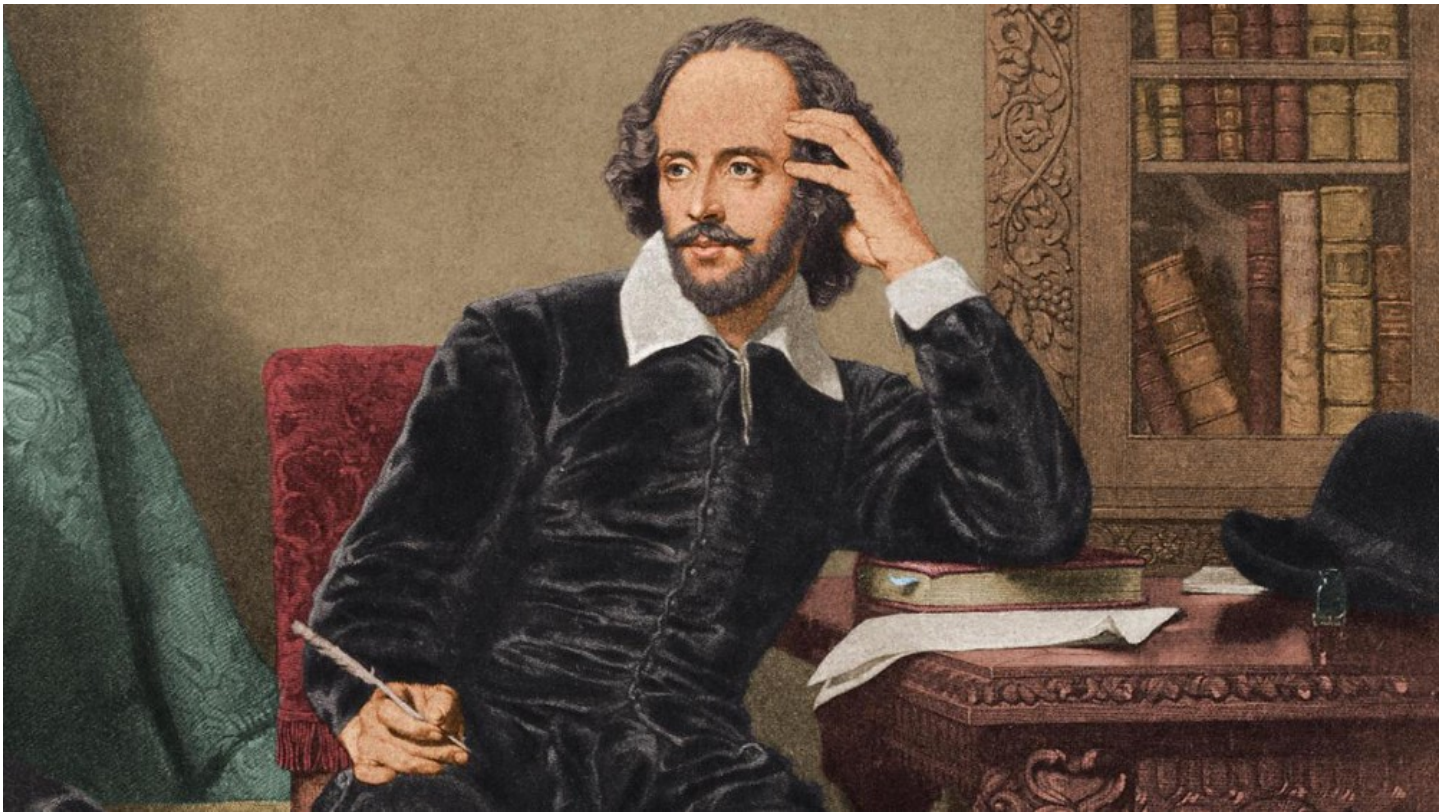


How I discovered the C++ algorithm library and learned not to reinvent the wheel



Mottakin Chowdhury

Apr 5, 2019 · 5 min read ★



To C++, or not to C++, that is the question. 🤔 — William Shakespeare (1564–1616)

The other day out of curiosity, I looked into the C++ algorithm library. And found out quite a good number of cool features!

This literally amazed me.

Why? I mean I have mostly written C++ throughout my university life. And it was particularly because of my love-hate relationship with competitive programming.

And very unfortunately, I had never really taken advantage of this amazing library C++ has offered us.

Gosh I felt so naïve!


So I decided it was time to stop being naive and get to know the usefulness of C++ algorithms — at least at a higher level. And as an old man once said, *sharing knowledge is power* — so here I am.

Disclaimer: *I have heavily used features from C++11 and beyond. If you are not quite familiar with newer editions of the language, the code snippets I have provided here might seem a bit clumsy. On the other hand, the library we discuss here is far more self-sufficient and elegant than anything I have written below. Feel free to find mistakes and point them out. And also, I could not really consider many of C++17 additions in this post, as most of its features are yet to be brought into life in GCC.*

So without further ado, let's begin!

1. `all_of` `any_of` `none_of`

These functions simply look for whether `all`, `any` or `none` of the elements of a container follows some specific property defined by you. Check the example below:



```
std::vector<int> collection = {3, 6, 12, 6, 9, 12};

// Are all numbers divisible by 3?
bool divby3 = std::all_of(begin(collection), end(collection), [](int x) {
    return x % 3 == 0;
});
// divby3 equals true, because all numbers are divisible by 3

// Is any number divisible by 2?
bool divby2 = std::any_of(begin(collection), end(collection), [](int x) {
    return x % 2 == 0;
```

Notice how in the example, the *specific property* is passed as a lambda function.

So `all_of`, `any_of`, `none_of` look for some specific property in your `collection`. These functions are pretty much self explanatory on what they are supposed to do. Along with the introduction of **lambdas** in C++11, they are pretty handy to use.

2. `for_each`

I have always been so accustomed to using age-old `for` loop that this cute thing never crossed my sight. Basically, `for_each` applies a function to a range of a container.

If you are a JavaScript developer, the above code should ring a bell.

3. `count` `count_if`

Pretty much like the functions described in the beginning, `count` and `count_if` both look for specific properties in your given collection of data.

And a result, you receive the **count** that matches your given value, or has the given property that you provide in the form of a lambda function.

4. `find_if`

Say you want to find the first element in your collection satisfying a particular property. You can use `find_if`.

Remember, as shown in the above example, you will get the **iterator** to the **first element** that matches your given property. So what if you want to find all the elements that match the property using `find_if`?





An abstract art to look at if you are getting bored. 😊 ([Steve Johnson](#) on [Unsplash](#))

5. `generate`

This function essentially changes the values of your collection, or a range of it, based on the **generator** you provide. The generator is a function of the form $\tau f()$; where τ is a compatible type with our collection.

In the above example, notice that we are actually changing our collection *in-place*. And the generator here is the lambda function we provided.

6. `shuffle`

From the standard of C++17, `random_shuffle` has been removed. Now we prefer `shuffle` which is more effective, given that it takes advantage of the header `random`.

Note that we are using Mersenne Twister, a pseudo-random number generator introduced in C++11.

Random number generators have become far more mature in C++ with the introduction of `random` library and inclusion of better methods.

7. `nth_element`

This function is quite useful, given that it has an interesting complexity.

Say you want to know the n -th element of your collection if it was sorted, but you do not want to sort the collection to make an $O(n \log(n))$ operation.

What would you do?

Then `nth_element` is your friend. It finds the desired element in $O(n)$.

Interestingly, `nth_element` may or may not make your collection sorted. It will just do whatever order it takes to find the n-th element. Here is an interesting discussion on [StackOverflow](#).

And also, you can always add your own comparison function (like we added lambdas in previous examples) to make it more effective.

8. `equal_range`

So let's say you have a sorted collection of integers. You want to find the range in which all the elements have a specific value. For example:

In this code, we are looking for a **range** in the **vector** that holds all **5**. The answer is **(2~4)**.

Of course we can use this function for our own custom property. You need to ensure that the property you have aligns with the order of the data. See [this article for reference](#).

Finally, `lower_bound` and `upper_bound` both can help you to achieve the same that you achieved using `equal_range`.

9. `merge inplace_merge`

Imagine you have two sorted collections (what a fun thing to imagine, right?), you want to merge them, and you also want the merged collection to remain sorted. How would you do that?

You can just add the second collection to the first one and sort the result again which adds an extra $O(\log(n))$ factor. Instead of that, we can just use `merge` .

On the other hand, do you remember when implementing *merge sort*, we need to merge two sides of our array? `inplace_merge` can be conveniently used for that.

Look at this tiny *merge sort* based on the example given in [c++reference](#):

How cool is that!



Speaking of cool, here is a cool guy. 😎 ([Dawid Zawita](#) on [Unsplash](#))

10. `minmax` `minmax_element`

`minmax` returns the minimum and maximum of the given two values, or the given list. It returns a pair and it can also provide the functionality of your own comparison method. `minmax_element` does the same for your container.

11. `accumulate` `partial_sum`

`accumulate` does what it says, it *accumulates* values of your collection in the given range, using the initial value and a binary operation function. See for yourself:

So how is the value of `custom` calculated?

At the beginning, `accumulate` takes the initial value (0) to the argument `x`, the first value in the collection (6) to argument `y`, does the operation, then assigns it to the accumulated value. In the second call, it passes the accumulated value to `x` and the next element in the collection to `y`, and thus proceeds.

`partial_sum` does things much like `accumulate`, but it also keeps the result of first `n` terms in a destination container.

And of course as you expected, you can use your own custom operation.

12. `adjacent_difference`

You want to find the adjacent differences in your values, you can simply use this function.

Pretty simple, right?

But it can do much more. Look at this:

What do these two lines do? They find the first 10 Fibonacci numbers! Do you see how? 🤔

So that was it for today. Thanks for reading! I hope you learned something new.

I would definitely like to bring some new stuff for ya'll again in near future.

Cheers! 🍻

[Tech](#) [Programming](#) [Algorithms](#) [Coding](#) [Technology](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

