18.3 — Introduction to standard library algorithms

learncpp.com/cpp-tutorial/introduction-to-standard-library-algorithms/

New programmers typically spend a lot of time writing custom loops to perform relatively simple tasks, such as sorting or counting or searching arrays. These loops can be problematic, both in terms of how easy it is to make an error, and in terms of overall maintainability, as loops can be hard to understand.

Because searching, counting, and sorting are such common operations to do, the C++ standard library comes with a bunch of functions to do these things in just a few lines of code. Additionally, these standard library functions come pre-tested, are efficient, work on a variety of different container types, and many support parallelization (the ability to devote multiple CPU threads to the same task in order to complete it faster).

The functionality provided in the algorithms library generally fall into one of three categories:

- **Inspectors** -- Used to view (but not modify) data in a container. Examples include searching and counting.
- **Mutators** -- Used to modify data in a container. Examples include sorting and shuffling.
- **Facilitators** -- Used to generate a result based on values of the data members. Examples include objects that multiply values, or objects that determine what order pairs of elements should be sorted in.

These algorithms live in the algorithms library. In this lesson, we'll explore some of the more common algorithms -- but there are many more, and we encourage you to read through the linked reference to see everything that's available!

Note: All of these make use of iterators, so if you're not familiar with basic iterators, please review lesson 18.2 -- Introduction to iterators.

Using std::find to find an element by value

`std::find` searches for the first occurrence of a value in a container. `std::find` takes 3 parameters: an iterator to the starting element in the sequence, an iterator to the ending element in the sequence, and a value to search for. It returns an iterator pointing to the element (if it is found) or the end of the container (if the element is not found).

For example:

```
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
    std::array arr{ 13, 90, 99, 5, 40, 80 };

    std::cout << "Enter a value to search for and replace with: ";
    int search{};
    int replace{};
    std::cin >> search >> replace;

    // Input validation omitted

    // std::find returns an iterator pointing to the found element (or the end of the
container)
    // we'll store it in a variable, using type inference to deduce the type of
    // the iterator (since we don't care)
    auto found{ std::find(arr.begin(), arr.end(), search) };

    // Algorithms that don't find what they were looking for return the end iterator.
    // We can access it by using the end() member function.
    if (found == arr.end())
    {
        std::cout << "Could not find " << search << '\n';
    }
    else
    {
        // Override the found element.
        *found = replace;
    }

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

Sample run when the element is found

```
Enter a value to search for and replace with: 5 234
13 90 99 234 40 80
```

Sample run when the element isn’t found

```
Enter a value to search for and replace with: 0 234
Could not find 0
13 90 99 5 40 80
```

Using std::find_if to find an element that matches some condition

Sometimes we want to see if there is a value in a container that matches some condition (e.g. a string that contains a specific substring) rather than an exact value. In such cases, `std::find_if` is perfect.

The `std::find_if` function works similarly to `std::find`, but instead of passing in a specific value to search for, we pass in a callable object, such as a function pointer (or a lambda, which we'll cover later). For each element being iterated over, `std::find_if` will call this function (passing the element as an argument to the function), and the function can return `true` if a match is found, or `false` otherwise.

Here's an example where we use `std::find_if` to check if any elements contain the substring "nut":

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

// Our function will return true if the element matches
bool containsNut(std::string_view str)
{
    // std::string_view::find returns std::string_view::npos if it doesn't find
    // the substring. Otherwise it returns the index where the substring occurs
    // in str.
    return (str.find("nut") != std::string_view::npos);
}

int main()
{
    std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

    // Scan our array to see if any elements contain the "nut" substring
    auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };

    if (found == arr.end())
    {
        std::cout << "No nuts\n";
    }
    else
    {
        std::cout << "Found " << *found << '\n';
    }

    return 0;
}
```

Output

```
Found walnut
```

If you were to write the above example by hand, you'd need at least three loops (one to loop through the array, and two to match the substring). The standard library functions allow us to do the same thing in just a few lines of code!

Using std::count and std::count_if to count how many occurrences there are

`std::count` and `std::count_if` search for all occurrences of an element or an element fulfilling a condition.

In the following example, we'll count how many elements contain the substring "nut":

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

bool containsNut(std::string_view str)
{
	return (str.find("nut") != std::string_view::npos);
}

int main()
{
	std::array<std::string_view, 5> arr{ "apple", "banana", "walnut", "lemon",
"peanut" };

	auto nuts{ std::count_if(arr.begin(), arr.end(), containsNut) };

	std::cout << "Counted " << nuts << " nut(s)\n";

	return 0;
}
```

Output

```
Counted 2 nut(s)
```

Using std::sort to custom sort

We previously used `std::sort` to sort an array in ascending order, but std::sort can do more than that. There's a version of `std::sort` that takes a function as its third parameter that allows us to sort however we like. The function takes two parameters to compare, and returns true if the first argument should be ordered before the second. By default, `std::sort` sorts the elements in ascending order.

Let's use `std::sort` to sort an array in reverse order using a custom comparison function named `greater`:

```
#include <algorithm>
#include <array>
#include <iostream>

bool greater(int a, int b)
{
    // Order @a before @b if @a is greater than @b.
    return (a > b);
}

int main()
{
    std::array arr{ 13, 90, 99, 5, 40, 80 };

    // Pass greater to std::sort
    std::sort(arr.begin(), arr.end(), greater);

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

Output

```
99 90 80 40 13 5
```

Once again, instead of writing our own custom loop functions, we can sort our array however we like in just a few lines of code!

Our `greater` function needs 2 arguments, but we're not passing it any, so where do they come from? When we use a function without parentheses (), it's only a function pointer, not a call. You might remember this from when we tried to print a function without parentheses and `std::cout` printed "1". `std::sort` uses this pointer and calls the actual `greater` function with any 2 elements of the array. We don't know which elements `greater` will be called with, because it's not defined which sorting algorithm `std::sort` is using under the hood. We talk more about function pointers in a later chapter.

Tip

Because sorting in descending order is so common, C++ provides a custom type (named `std::greater`) for that too (which is part of the functional header). In the above example, we can replace:

```
std::sort(arr.begin(), arr.end(), greater); // call our custom greater function
```

with:

```
std::sort(arr.begin(), arr.end(), std::greater{}); // use the standard library
greater comparison
// Before C++17, we had to specify the element type when we create std::greater
std::sort(arr.begin(), arr.end(), std::greater<int>{}); // use the standard library
greater comparison
```

Note that the `std::greater{}` needs the curly braces because it is not a callable function. It's a type, and in order to use it, we need to instantiate an object of that type. The curly braces instantiate an anonymous object of that type (which then gets passed as an argument to std::sort).

For advanced readers

To further explain how `std::sort` uses the comparison function, we'll have to take a step back to a modified version of the selection sort example from lesson 18.1 -- Sorting an array using selection sort.

```
#include <iostream>
#include <iterator>
#include <utility>

void sort(int* begin, int* end)
{
    for (auto startElement{ begin }; startElement != end-1; ++startElement)
    {
        auto smallestElement{ startElement };

        // std::next returns a pointer to the next element, just like (startElement +
1) would.
        for (auto currentElement{ std::next(startElement) }; currentElement != end;
++currentElement)
        {
            if (*currentElement < *smallestElement)
            {
                smallestElement = currentElement;
            }
        }

        std::swap(*startElement, *smallestElement);
    }
}

int main()
{
    int array[]{ 2, 1, 9, 4, 5 };

    sort(std::begin(array), std::end(array));

    for (auto i : array)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

So far, this is nothing new and `sort` always sorts elements from low to high. To add a comparison function, we have to use a new type, `std::function<bool(int, int)>`, to store a function that takes 2 int parameters and returns a bool. Treat this type as magic for now, we will explain it in chapter 20.

```
void sort(int* begin, int* end, std::function<bool(int, int)> compare)
```

We can now pass a comparison function like `greater` to `sort`, but how does `sort` use it? All we need to do is replace the line

```
if (*currentElement < *smallestElement)
```

with

```
if (compare(*currentElement, *smallestElement))
```

Now the caller of `sort` can choose how to compare two elements.

```
#include <functional> // std::function
#include <iostream>
#include <iterator>
#include <utility>

// sort accepts a comparison function
void sort(int* begin, int* end, std::function<bool(int, int)> compare)
{
    for (auto startElement{ begin }; startElement != end-1; ++startElement)
    {
        auto smallestElement{ startElement };

        for (auto currentElement{ std::next(startElement) }; currentElement != end;
++currentElement)
        {
            // the comparison function is used to check if the current element should
be ordered
            // before the currently "smallest" element.
            if (compare(*currentElement, *smallestElement))
            {
                smallestElement = currentElement;
            }
        }

        std::swap(*startElement, *smallestElement);
    }
}

int main()
{
    int array[]{ 2, 1, 9, 4, 5 };

    // use std::greater to sort in descending order
    // (We have to use the global namespace selector to prevent a collision
    // between our sort function and std::sort.)
    ::sort(std::begin(array), std::end(array), std::greater{});

    for (auto i : array)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

Using std::for_each to do something to all elements of a container

`std::for_each` takes a list as input and applies a custom function to every element. This is useful when we want to perform the same operation to every element in a list.

Here's an example where we use `std::for_each` to double all the numbers in an array:

```
#include <algorithm>
#include <array>
#include <iostream>

void doubleNumber(int& i)
{
    i *= 2;
}

int main()
{
    std::array arr{ 1, 2, 3, 4 };

    std::for_each(arr.begin(), arr.end(), doubleNumber);

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

Output

```
2 4 6 8
```

This often seems like the most unnecessary algorithm to new developers, because equivalent code with a range-based for-loop is shorter and easier. But there are benefits to `std::for_each`. Let's compare `std::for_each` to a range-based for-loop.

```
std::ranges::for_each(arr, doubleNumber); // Since C++20, we don't have to use
begin() and end().
// std::for_each(arr.begin(), arr.end(), doubleNumber); // Before C++20

for (auto& i : arr)
{
    doubleNumber(i);
}
```

With `std::for_each`, our intentions are clear. Call `doubleNumber` with each element of `arr`. In the range-based for-loop, we have to add a new variable, `i`. This leads to several mistakes that a programmer could do when they're tired or not paying attention. For one, there could be an implicit conversion if we don't use `auto`. We could forget the ampersand, and `doubleNumber` wouldn't affect the array. We could accidentally pass a variable other than `i` to `doubleNumber`. These mistakes cannot happen with `std::for_each`.

Additionally, `std::for_each` can skip elements at the beginning or end of a container, for example to skip the first element of `arr`, `std::next` can be used to advance begin to the next element.

```
std::for_each(std::next(arr.begin()), arr.end(), doubleNumber);
// Now arr is [1, 4, 6, 8]. The first element wasn't doubled.
```

This isn't possible with a range-based for-loop.

Like many algorithms, `std::for_each` can be parallelized to achieve faster processing, making it better suited for large projects and big data than a range-based for-loop.

Performance and order of execution

Many of the algorithms in the algorithms library make some kind of guarantee about how they will execute. Typically these are either performance guarantees, or guarantees about the order in which they will execute. For example, `std::for_each` guarantees that each element will only be accessed once, and that the elements will be accessed in forwards sequential order.

While most algorithms provide some kind of performance guarantee, fewer have order of execution guarantees. For such algorithms, we need to be careful not to make assumptions about the order in which elements will be accessed or processed.

For example, if we were using a standard library algorithm to multiply the first value by 1, the second value by 2, the third by 3, etc… we'd want to avoid using any algorithms that didn't guarantee a forwards sequential execution order!

The following algorithms guarantee sequential execution: `std::for_each`, `std::copy`, `std::copy_backward`, `std::move`, and `std::move_backward`. Many other algorithms (particular those that use a forward iterator) are implicitly sequential due to the forward iterator requirement.

Best practice

Before using a particular algorithm, make sure performance and execution order guarantees work for your particular use case.

Ranges in C++20

Having to explicitly pass `arr.begin()` and `arr.end()` to every algorithm is a bit annoying. But fear not -- C++20 adds *ranges*, which allow us to simply pass `arr`. This will make our code even shorter and more readable.

Conclusion

The algorithms library has a ton of useful functionality that can make your code simpler and more robust. We only cover a small subset in this lesson, but because most of these functions work very similarly, once you know how a few work, you can make use of most of them.

As an aside…

This video does a good job explaining various algorithms in the library in a concise way.

Best practice

Favor using functions from the algorithms library over writing your own functionality to do the same thing.