6.18 — Introduction to standard library algorithms

BY NASCARDRIVER ON JANUARY 3RD, 2020 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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 functionailty 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 reference linked above 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 **6.17 -- 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) };
```

```
19
20    // Algorithms that don't find what they were looking for return the end iterator.
21    // We can access it by using the end() member function.
      if (found == arr.end())
22    {
23      std::cout << "Could not find " << search << '\n';
24    }
25    else
26    {
27      // Override the found element.
28      *found = replace;
29    }
30
31    for (int i : arr)
32    {
33      std::cout << i << ' ';
34    }
35
36    std::cout << '\n';
37
38    return 0;
39  }
40
41
```

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 work similarly to `std::find`, but instead of passing in a value to search for, we pass in a callable object, such as a function pointer (or a lambda, which we'll cover later) that checks to see if a match is found. `std::find_if` will call this function for every element until a matching element is found (or no more elements remain in the container to check).

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

```
1   #include <algorithm>
2   #include <array>
3   #include <iostream>
4   #include <string_view>
5
6   // Our function will return true if the element matches
7   bool containsNut(std::string_view str)
8   {
9     // 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
10    // 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 two loops (one to loop through the array, and one 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!

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 ca
parison
```

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).

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

`std::for_each` takes a list as input, 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 beginners, because equivalent code with a range-based for-loop is shorter and easier. `std::for_each` allows the loop-body to be re-used and it can be parallelized, making it better suited for large projects and big data than a range-based for-loop.

Order of execution

Note that most of the algorithms in the algorithms library do not guarantee a particular order of execution. For such algorithms, take care to ensure any functions you pass in do not assume a particular ordering, as the order of invocation may not be the same on every compiler.

The following algorithms do guarantee sequential execution: `std::for_each`, `std::copy`, `std::copy_backward`, `std::move`, and `std::move_backward`.

Best practice

Unless otherwise specified, do not assume that standard library algorithms will execute in a particular sequence. `std::for_each`, `std::copy`, `std::copy_backward`, `std::move`, and `std::move_backward` have sequential guarantees.

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.

Best practice

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

 **6.x -- Chapter P.6 comprehensive quiz**

 **Index**

 **6.17 -- Introduction to iterators**

C++ TUTORIAL | PRINT THIS POST

6 comments to 6.18 — Introduction to standard library algorithms

kavin
January 28, 2020 at 10:03 am · Reply

Hi, i can't understand how this works?

```
1 return (str.find("nut") != std::string_view::npos);
```

what does the returned std::string_view::npos do?

Also under std::sort , we have 2 parameters in

```
bool greater(int a, int b)
```

How do we determine how many parameters to take ?(because so far we have taken 1 parameter for 1 argument...) I see function names "containsNut/doubleNumber". Which ones are the exact arguments to consider for the resp functions ?

nascardriver
January 29, 2020 at 3:43 am · Reply

`std::string_view::find` returns the index at which the substring was found. If it doesn't find the substring, it returns the special value `std::string_view::npos` (Similarly, `std::string::find` returns `std::string::npos`).

`std::sort` uses the function to compare 2 elements to each other. If it took only one argument, it wouldn't be able to compare it to anything. You can look this up on **cppreference** (We're using the version marked with (3)).

Ged
January 28, 2020 at 6:35 am · Reply

I wrote a code when the user has to put the input. We cannot input std::string_view. I wrote the function inside the std::find_if just to know it is possible to do it that way as well.

```
int main()
{
    int length{};
    std::cin >> length;
    std::vector<std::string> vector(length);

    for (auto& name : vector)
        std::cin >> name;

    std::vector<std::string_view> vector2(vector.size());

    std::copy(std::begin(vector), std::end(vector), std::begin(vector2));

    auto found{ std::find_if(std::begin(vector2), std::end(vector2),
        [](const std::string_view& word) {return (word.find("nut") != std::string_view::np
os); }) };

    if (found == std::end(vector2))
        std::cout << "No word contains - (nut)";
    else
        std::cout << *found << " contains the word - (nut)";
    return 0;

}

```

Question 1 Is the code above a good solution to the problem i wrote?

Question 2 When we write the whole function inside our algorithm like std::find_if, can we do something with "[]" or does it stand for a one time use nameless function?

Question 3 When there is a "_if" at the end, that means we need to use a function? If there is no "_if" where it is possible we need to use variable or something similar, but not a function?

Question 4 Why don't we need to use parenthesis at the end of our function "containsNut"? EXAMPLE auto nuts{ std::count_if(arr.begin(), arr.end(), containsNut) };

Question 5 When we are using standard library algorithms we use auto for our new variable. Is the data type is super long, is it good practice to be using auto instead of a specific data type?

nascardriver
January 28, 2020 at 6:54 am · Reply

1
You can use `std::find_if` for any type of `std::vector`. You don't need a `std::string_view`.
If you want to copy the `std::vector` for whatever reason, you can do so without `std::copy`:

```
1 std::vector<std::string_view> vector2(vector.begin(), vector.end());
```

The rest of your code looks good :)

2
That's a lambda function. Lambdas are covered in chapter 7.

3
There are functions without an "_if" postfix that require a function. Those with an "_if" postfix require a function.

4
We're not calling `containsNut`, `std::count_if` is calling `containsNut`. We're passing the functions itself, not the return value, to `std::count_if`. Function pointers are also covered in chapter 7.

5
If the type is obvious, very long, or doesn't matter, use `auto`. For example in the lambda you're passing to `std::find_if`, it's obvious that the parameter has the type of the vector's elements, so you can use `auto` without sacrificing readability.

Matt
January 4, 2020 at 4:05 am · Reply

I just had to :) Glad to see this site is still being updated!

```
1  // main.cpp
2  #include <algorithm>
3  #include <array>
4  #include <iostream>
5
6  void doubleNumber(int& i)
7  {
8      i *= 2;
9  }
10
11 void print(int& i)
12 {
13     std::cout << i << ' ';
14 }
15
16 int main()
17 {
18     std::array arr{ 1, 2, 3, 4 };
19
20     std::for_each(arr.begin(), arr.end(), doubleNumber);
21
22     std::for_each(arr.begin(), arr.end(), print); // yea!
23
24     std::cout << '\n';
```

```
25
26      return 0;
27  }
```

nascardriver
January 5, 2020 at 6:28 am · Reply

You don't even need to write a function do print lists:

```
1  std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>{ std::cout, " " });
```

We're using loops in the lessons, because they're easier for beginners. Maybe we'll add some single-line action in later lessons :)