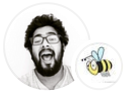
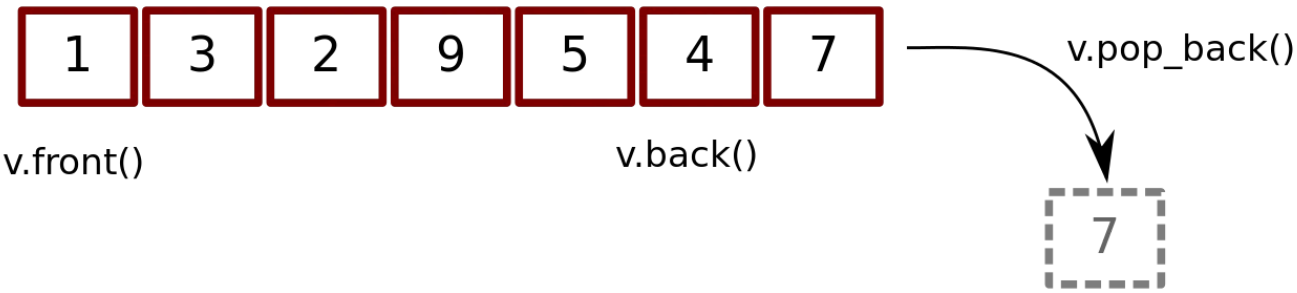
C++ STL Algorithms Cheat Sheet



Sushanth Kurdekar · Follow
Published in Logical Bee
34 min read · Jan 26, 2020

Listen Share



C++ STL Alogorithms

C++ STL Algorithms Cheat Sheet

The C++ Standard Template Library provides algorithms for functions that are commonly needed in everyday-scenarios, and for commonly used containers.

Now, I'm sure you're a great coder and all and could write your own functions, but you should probably use the Standard Library's functions whenever possible. They're very efficient, both memory and space wise, especially if you use STL containers.

Table of Contents

- find
- find_if
- find*if*not
- find_end
- find*first*of
- adjacent_find
- count
- count_if
- mismatch
- equal
- is_permutation
- search
- search_n
- lexicographical_compare
- copy
- copy_if
- copy_n
- copy_backward
- move
- move_backward
- swap
- swap_ranges
- iter_swap
- transform

- replace
- replace_if
- replace_copy
- replace*copy*if
- fill
- fill_n
- generate
- generate_n
- remove
- remove_if
- remove_copy
- remove*copy*if
- unique
- unique_copy
- reverse
- reverse_copy
- rotate
- rotate_copy
- random_shuffle
- shuffle
- is_partitioned
- partition
- stable_partition
- partition_copy

- partition_point
- sort
- stable_sort
- partial_sort
- partial*sort*copy
- is_sorted
- is*sort*until
- nth_element
- lower_bound
- upper_bound
- equal_range
- binary_search
- merge
- inplace_merge

- set_intersection
- set_difference
- set*symmetric*difference
- make_heap
- push_heap
- pop_heap
- sort_heap
- is_heap

Base Code and C++ version

> *For the sake of testing this code, we'll specify a very basic C++ file that our code would run in. Note that we include* `vector` *, because that's the container the examples will use, we're including* `iostream` *for* `cout` *, and we including* `algorithm` *for the actual STL algorithms.*

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
        {
        // INSERT STD:ALGORITHM EXAMPLE CODE HERE

        return 0;
        }
```

We'll use a simple makefile to compile and run, specifying C++11.

makefile

```
all:
        g++ main.cpp -std=c++11 -o run
        ./run
```

And so to actually run this code, in a UNIX operating system, in the terminal, simply type `make` in the directory where your makefile and main.cpp are.

Figure Something Out About Data

Need to do something with the data in your container? Look here.

all_of

Check if every element in the range satisfies the condition.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a lambda function that does something with the element of the container.
- Returns a bool. True if all elements satisfied the condition, false otherwise.

```
std::vector<int> v{ 5, 3, 7, 9, 4 };
auto lambda = [](int i) { return i > 1; };
bool allGreaterThanOne = std::all_of(v.begin(), v.end(), lambda); // true
```

any_of

Check if any element in the range satisfies the condition.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a lambda function that does something with the element of the container.
- Returns a bool. True if any element satisfied the condition, false otherwise.

```
std::vector<int> v{ 5, 3, 7, 9, 4 };
auto lambda = [](int i) { return i > 8; };
bool anyGreaterThanEight = std::any_of(v.begin(), v.end(), lambda);
// true
```

none_of

Check if none of the elements in the range satisfies the condition.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a lambda function that does something with the element of the container.
- Returns a bool. True if all elements fail the condition, false if any element passes the condition.

```
std::vector<int> v{ 5, 3, 7, 9, 4 };
auto lambda = [](int i) { return i > 10; };
bool noneGreaterThanTen = std::none_of(v.begin(), v.end(), lambda);
// true
```

for_each

Does something for each item in a range.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a lambda function that does something with the element of the container.
- Returns void.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };
auto lambda = [](int i) { std::cout << i << " "; };
std::for_each(v.begin(), v.end(), lambda); // Prints each element in
the container.
```

find

Find an item in a given range.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and an item that you want to find.
- Returns an iterator to the first element in the range that is equal to the item we specified.

```
std::vector<int> v{ 5, 3, 7, 9, 4 };
```

```
std::vector<int>::iterator it = std::find(v.begin(), v.end(), 3);
```

find_if

Find the first item that satisfies a condition.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a lambda function that returns a bool.
- Returns an iterator to the first element in the range that satisfies the lambda's condition.

```
std::vector<int> v{ 5, 3, 7, 9, 4 };

auto lambda = [](int i) { return i > 6; };

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(),
lambda);

int firstElementGreaterThanSix = *it; // 7
```

findifnot

Find the first item that does not satisfy a condition.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a lambda function that returns a bool.
- Returns an iterator to the first element in the range that does not satisfy the lambda's condition.

```
std::vector<int> v{ 5, 3, 7, 9, 4 };

auto lambda = [](int i) { return i > 6; };

std::vector<int>::iterator it = std::find_if_not(v.begin(), v.end(),
lambda);

int firstElementLessThanSix = *it; // 5
```

find_end

For a range, find the last occurence of a sequence in that range. (Ex. Get the last "oo" in "moo_cookies".)

- Pass in an iterator to the beginning of the first range, and an iterator to the end of the first range, and an iterator to the beginning of the sequence, and an iterator to the end of the sequence,
- Returns an iterator to the first item of the sequence.

```
std::string s = "moo_cookies";
std::string t = "oo";

std::string::iterator it = std::find_end(s.begin(), s.end(),
t.begin(), t.end());
// Points to the 'o' after the 'c'
```

findfirstof

For a range, find the first occurence of a sequence in that range. (Ex. Get the first “oo” in “moo_cookies”.)

- Pass in an iterator to the beginning of the first range, and an iterator to the end of the first range, and an iterator to the beginning of the sequence, and an iterator to the end of the sequence,
- Returns an iterator to the first item of the sequence.

```
std::string s = "moo_cookies";
std::string t = "oo";

std::string::iterator it = std::find_first_of(s.begin(), s.end(),
t.begin(), t.end());
// Points to the 'o' after the 'm
```

adjacent_find

Find the first occurrence of two consecutive elements that match in a range. (Ex. The “cc” in “accentt”.)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns an iterator to the first item in the match.

```
std::string s = "accentt";
```

```
std::string::iterator it = std::adjacent_find(s.begin(), s.end());
// Points to the first 'c'
```

count

Count the number of times an item appears in the range.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and the item we want to count.
- Returns an integer.

```
std::vector<int> v{ 5, 3, 7, 9, 3, 4 };

int countOfThree = std::count(v.begin(), v.end(), 3); // 2
```

count_if

Count the number of occurrences satisfying the lambda function.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a lambda function that returns true or false.
- Returns an integer.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };

auto lambda = [](int i) { return i > 2; };

int count = count_if(v.begin(), v.end(), lambda);
```

mismatch

Finds the first occurrence where two rangers differ.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and an iterator to the beginning of the second range.
- Returns a `pair` of iterators to the positions where the ranges occur.

```
std::vector<int> v1{ 5, 3, 7, 9 };
std::vector<int> v2{ 5, 3, 2, 9 };
```

```
std::pair<std::vector<int>::iterator, std::vector<int>::iterator> p
= std::mismatch(v1.begin(), v1.end(), v2.begin());

int element1 = *p.first; // 7
int element2 = *p.second; // 2
```

equal

Check if the elements in two ranges are equal.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and an iterator to the beginning of the second range.
- Returns a bool. True if the elements are equal, false otherwise.

```
std::vector<int> v1{ 5, 4, 6 };
std::vector<int> v2{ 5, 4, 6 };

bool isEqual = std::equal(v1.begin(), v1.end(), v2.begin()); // true
```

is_permutation

Check if a range is a permutation of another.

A “permutation” is if the items in one range can be rearranged to form the other range. (Ex. “dog” can be rearranged to form “god”.)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and an iterator to the beginning of the second range.
- Returns a bool. True if the elements are permutations, false otherwise.

```
std::vector<char> v1{ 'g', 'o', 'd' };
std::vector<char> v2{ 'd', 'o', 'g' };

bool isPermutation = std::is_permutation(v1.begin(), v1.end(),
v2.begin()); // true
```

search

Check if a range contains a certain sequence. (Ex. Does “this” contain “is”?)

- Pass in an iterator to the beginning of the first range, and an iterator to the end of the first range, and an iterator to the beginning of the second range

(sequence), and an iterator to the end of the second range (sequence).

- Returns an iterator to the first item in the range that begins the sequence.

```
std::vector<char> v1{ 't', 'h', 'i', 's' };
std::vector<char> v2{ 'i', 's' };

std::vector<char>::iterator it = std::search(v1.begin(), v1.end(),
v2.begin(), v2.end());
// Points to the 'i' in v1
```

search_n

Search a range for a certain number of a specific item. (Ex. Find two ‘e’s in a row in the word ‘esteem’.)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and the count of the number of the item we need to find, and the item we’re trying to find.
- Returns an iterator to the first item in the range that begins the sequence.

```
std::vector<char> v{ 'e', 's', 't', 'e', 'e', 'm' };

std::vector<char>::iterator it = std::search_n(v.begin(), v.end(),
2, 'e');
// Points to the 'e' after the 't'.
```

lexicographical_compare

Lexographically compare two items to find out which is ‘smaller’.

- Pass in an iterator to the beginning of the first range, and an iterator to the end of the first range, an iterator to the beginning of the second range, and an iterator to the end of the second range.
- Returns a bool. True if the first range is lexographically less than the second range.

```
std::string s = "abc";
std::string t = "def";
```

```
bool sIsSmaller = std::lexicographical_compare(s.begin(), s.end(),
t.begin(), t.end()); // true
```

Modify/Copy A Range

A range is going to be edited or copied.

copy

Copy the elements from one range into another.

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, and an iterator to the beginning of the copy range.
- Returns an iterator to the element after the last one we wrote to in the copy range.

```
std::vector<int> v1{ 1, 2, 3, 4 };
std::vector<int> v2(4);

std::vector<int>::iterator it = std::copy(v1.begin(), v1.end(),
v2.begin());

// v2 is { 1 2 3 4 }
// 'it' points to the element after 4 in v2.
```

copy_n

Copy the first n elements from one range into another.

- Pass in an iterator to the beginning of the first range, an int representing how many elements to copy, and an iterator to the beginning of the copy range.
- Returns an iterator to the element after the last one we wrote to in the copy range.

```
std::vector<int> v1{ 1, 2, 3, 4 };
std::vector<int> v2(2);

std::vector<int>::iterator it = std::copy_n(v1.begin(), 2,
v2.begin());

// v2 is { 1 2 }
// "it" points to the element after 2 in v2.
```

copy_if

Copy elements from one range into another if the condition we specify is met.

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, an iterator to the beginning of the copy range, and a lambda function that returns a bool.
- Returns an iterator to the element after the last one we wrote to in the copy range.

```
std::vector<int> v1{ 1, 2, 3, 4 };
std::vector<int> v2(2);

auto lambdaIsEven = [](int i) { return i % 2 == 0; };

std::vector<int>::iterator it = std::copy_if(v1.begin(), v1.end(),
v2.begin(), lambdaIsEven);

// v2 is { 2 4 }
// "it" points to the element after 4 in v2.
```

copy_backward

Copy the elements from one range into another, starting from the back elements and going to the front.

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, and an iterator to the end of the copy range.
- Returns an iterator to the last element we wrote to in the copy range (which is the beginning of the copy range).

```
std::vector<int> v1{ 1, 2, 3, 4 };
std::vector<int> v2(4);

std::vector<int>::iterator it = std::copy_backward(v1.begin(),
v1.end(), v2.end());

// v2 is { 1 2 3 4 }
// "it" points to the 1 in v2.
```

move

Move the elements from one range into another. The elements in the original range are valid, but may not be what they were before the move.

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, and an iterator to the beginning of the move range.
- Returns an iterator to the element after the last one we wrote to in the move range.

```
std::vector<int> v1{ 1, 2, 3, 4 };
std::vector<int> v2(4);

std::vector<int>::iterator it = std::move(v1.begin(), v1.end(),
v2.begin());

// v1 is { 1 2 3 4 } (It just happens to be unchanged.)
// v2 is { 1 2 3 4 }
// 'it' points to the element after 4 in v2.
```

move_backward

Move the elements from one range into another, starting from the back elements and going to the front. The elements in the original range are valid, but may not be what they were before the move.

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, and an iterator to the end of the move range.
- Returns an iterator to the last element we wrote to in the move range (which is the beginning of the move range).

```
std::vector<int> v1{ 1, 2, 3, 4 };
std::vector<int> v2(4);

std::vector<int>::iterator it = std::move_backward(v1.begin(),
v1.end(), v2.end());

// v1 is { 1 2 3 4 } (It just happens to be unchanged.)
// v2 is { 1 2 3 4 }
// 'it' points to the 1 in v2.
```

swap

Swaps the values of two items.

- Pass in the first and the second item.
- Returns void.

```
int a = 5;
int b = 10;

std::swap(a, b); // Now a = 10 and b = 5.
```

swap_ranges

Given two ranges, swaps the items in them (using the `swap` function).

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, and an iterator to the beginning of the second range.
- Returns an iterator to the element after the last one we wrote to in the second range.

```
std::vector<int> v1{ 1, 2, 3, 4 };
std::vector<int> v2{ 5, 6, 7, 8 };

std::vector<int>::iterator it = std::swap_ranges(v1.begin(),
v1.end(), v2.begin());

// v1 is { 5 6 7 8 }
// v2 is { 1 2 3 4 }
// 'it' points to the element after 4 in v2.
```

iter_swap

Swap the values that two iterators point too.

- Pass in an iterator to the first value, and an iterator to the second value.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };

std::vector<int>::iterator it1 = v.begin(); // Points to 1
std::vector<int>::iterator it2 = v.end() - 1; // Points to 4

std::iter_swap(it1, it2);
```

```
// v is now { 5 2 3 4 1 }
```

transform

Given one or two ranges, performs an operation on each value of the range and stores that result in another range. (For example, multiply each value in a vector by some number and store it in another vector. Or add each value in two vectors and put that into another vector.)

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, (optionally, an iterator to the beginning of the second range if there is one), and an iterator to the beginning of the resulting range, and a lamba expression that accepts the element type(s) and returns the element type that does the operation we want to want to do on the range(s).
- Returns an iterator to the element after the last one we wrote to in the resulting range.

Example with just one range:

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::vector<int> res(5);

auto lambdaAddOne = [](int i) { return i + 1; };

std::vector<int>::iterator it = std::transform(v.begin(), v.end(),
res.begin(), lambdaAddOne);

// res is { 2 3 4 5 6 }
```

Example with two ranges:

```
std::vector<int> v1{ 1, 2, 3 };
std::vector<int> v2{ 4, 5, 6 };
std::vector<int> res(3);

auto lambdaMultiply = [](int i, int j) { return i * j; };

std::vector<int>::iterator it = std::transform(v1.begin(), v1.end(),
v2.begin(), res.begin(), lambdaMultiply);

// res is { 4 10 18 }
```

replace

For a range, replaces an old value (that we specify) with a new value (that we specify). (Ex. Replace all 3's with 7's.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, the old value, and the new value.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4, 3, 5 };
std::replace(v.begin(), v.end(), 3, 7);
// res is now { 1 2 7 4 7 5 }
```

replace_if

For a range, replaces an element that passes a certain condition with a new value (that we specify). (Ex. Replace all negative elements with a 0.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, a lambda expression that returns a bool, and the new value.
- Returns void.

```
std::vector<int> v{ 1, 2, -3, 4, -5 };
auto lambdaIsNegative = [](int i) { return i < 0; };
std::replace_if(v.begin(), v.end(), lambdaIsNegative, 0);
// res is now { 1 2 0 4 0 }
```

replace_copy

For a range, copies the elements into another range, and replaces any element (that we specify) with a new value (that we specify) for the new range. Doesn't change the old range. (Ex. Replace all 3's with 7's.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, an iterator to the beginning of the new range, the old value, and the new value.

- Returns an iterator to the element after the last one we wrote to in the resulting range.

```
std::vector<int> v{ 1, 2, 3, 4, 3, 5 };
std::vector<int> res(6);

//auto lambdaIsNegative = [](int i) { return i < 0; };

std::vector<int>::iterator it = std::replace_copy(v.begin(),
v.end(), res.begin(), 3, 7);

// v is still { 1 2 3 4 3 5 }
// res is now { 1 2 7 4 7 5 }
// 'it' points to the element after 5 in res.
```

replacecopyif

For a range, copies the elements into another range, and replaces any element that passes a certain condition with a different value (that we specify) for the new range. Doesn't change the old range. (Ex. Replace all negative elements with a 0.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, an iterator to the beginning of the new range, a lambda expression that returns a bool, and the new value.
- Returns an iterator to the element after the last one we wrote to in the resulting range.

```
std::vector<int> v{ 1, 2, -3, 4, -5 };
std::vector<int> res(5);

auto lambdaIsNegative = [](int i) { return i < 0; };

std::vector<int>::iterator it = std::replace_copy_if(v.begin(),
v.end(), res.begin(), lambdaIsNegative, 0);

// v is still { 1 2 -3 4 -5 }
// res is now { 1 2 0 4 0 }
// 'it' points to the element after the second 0 in res.
```

fill

For a range, fill every element with an item (that we specify). (Ex. Fill a range with all 7s.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and the value that we want to fill it with.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::fill(v.begin(), v.end(), 7);
// v is now { 7 7 7 7 7 }
```

fill_n

For a range, fill every element with an item (that we specify). (Ex. Fill the first 3 elements of a range with 7s.)

- Pass in an iterator to the beginning of the range,the number of elements we want to fill, and the value that we want to fill it with.
- Returns an iterator to the element after the last one we wrote to.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::fill_n(v.begin(), 3, 7); // Fill the first 3, only.
// v is now { 7 7 7 4 5 }
```

generate

For a range, assigns each element with a value generated by a function (that we specify). (Ex. Use a random number generator function to determine what to assign each element.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and a function (that accepts no parameters) that returns the type of the range.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
auto lambdaReturnNine = []() { return 9; };
```

```
std::generate(v.begin(), v.end(), lambdaReturnNine);

// v is now { 9 9 9 9 9 }
```

generate_n

For a range, assigns each element with a value generated by a function (that we specify) for the first n elements. (Ex. Use a random number generator function to determine what to assign each element for the first 3 elements.)

- Pass in an iterator to the beginning of the range, the number of elements we want assign to, and a function (that accepts no parameters) that returns the type of the range.
- Returns an iterator to the element after the last one we wrote to.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
auto lambdaReturnNine = []() { return 9; };
std::generate_n(v.begin(), 3, lambdaReturnNine);
// v is now { 9 9 9 4 5 }
```

remove

For a range, removes any element that has a certain value (that you specify). This is done by moving a valid element into the removed element's spot. (The container size does not change.) You'll know where the updated range ends by checking where the returned iterator is. Order is not preserved, and the extra elements at the end of the range are in some valid state.

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and the value to be removed.
- Returns an iterator after the last valid element in the range.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::vector<int>::iterator it = std::remove(v.begin(), v.end(), 2);
// v is now { 1 3 4 5 5 }
```

```
// 'it' points to the element after the first 5 in v.
```

remove_if

For a range, removes any element that passes a certain condition (that you specify). This is done by moving a valid element into the removed element's spot. (The container size does not change.) You'll know where the updated range ends by checking where the returned iterator is. Order is not preserved, and the extra elements at the end of the range are in some valid state.

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and a lambda function that returns a bool.
- Returns an iterator after the last valid element in the range.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };

auto lambdaIsOdd = [](int i) { return i % 2 != 0; };

std::vector<int>::iterator it = std::remove_if(v.begin(), v.end(),
lambdaIsOdd);

// v is now { 2 4 3 4 5 }

// 'it' points to the element after the first 4 in v.
```

remove_copy

For a given range, copies each element into a new range, except for elements with a certain value (that you specify). The original range is untouched.

- Pass in an iterator to the beginning of the original range, an iterator to the end of the original range, an iterator to the beginning of the copy range, and the value that you want to skip.
- Returns an iterator after the last valid element in the copy range.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::vector<int> res(5);

std::vector<int>::iterator it = std::remove_copy(v.begin(), v.end(),
res.begin(), 2);

// v is now { 1 3 4 5 0 } // (The zero is from the initialization.)
```

```
// 'it' points to the element after the 5 in res
```

removecopyif

For a given range, copies each element into a new range, except for elements that pass a certain condition (that you specify). The original range is untouched.

- Pass in an iterator to the beginning of the original range, an iterator to the end of the original range, an iterator to the beginning of the copy range, and a lambda function that returns a bool.
- Returns an iterator after the last valid element in the copy range.

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::vector<int> res(5);

auto lambdaIsOdd = [](int i) { return i % 2 != 0; };

std::vector<int>::iterator it = std::remove_copy_if(v.begin(),
v.end(), res.begin(), lambdaIsOdd);

// v is now { 2 4 0 0 0 } // (The zeroes are from the
initialization.)

// 'it' points to the element after the 4 in res
```

unique

For a range, remove consecutive duplicate elements. (Ex. { 2 2 3 3 4 4 4 3 } -> { 2 3 4 3 ? ? ? ? }) This is done by moving a valid element into the removed element's spot. (The container size does not change.) You'll know where the updated range ends by checking where the returned iterator is. Order is not preserved, and the extra elements at the end of the range are in some valid state.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns an iterator after the last valid element in the range.

```
std::vector<int> v{ 2, 2, 3, 3, 4, 4, 4, 3 };

std::vector<int>::iterator it = std::unique(v.begin(), v.end());

// v is now { 2 3 4 3 4 4 4 3 }
```

```
// 'it' points to the element after the second 3 in v.
```

unique_copy

For a range, copies all elements that don't have a duplicate neighbor into a new range (that you specify). (Ex. { 2 2 3 3 4 4 4 3 } -> { 2 3 4 3 }) The original range is untouched.

- Pass in an iterator to the beginning of the original range, an iterator to the end of the original range, and an iterator to the beginning of the copy range.
- Returns an iterator after the last copied element in the copy range.

```
std::vector<int> v{ 2, 2, 3, 3, 4, 4, 4, 3 };
std::vector<int> res(8);

std::vector<int>::iterator it = std::unique_copy(v.begin(), v.end(),
res.begin());

// v is still { 2 3 4 3 4 4 4 3 }
// res is now { 2 3 4 3 0 0 0 0 } (The zeroes are from
initialization.)

// 'it' points to the element after the second 3 in res.
```

reverse

For a range, reverses the order of the elements. (Ex. { 1 2 3 4 } -> { 4 3 2 1 } .)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4 };

std::reverse(v.begin(), v.end());

// v is { 4 3 2 1 }
```

reverse_copy

For a range, copies the elements in verse order into a new range (that you specify). (Ex. { 1 2 3 4 } -> { 4 3 2 1 } .) The original range is untouched.

- Pass in an iterator to the beginning of the original range, an iterator to the end of the original range, and an iterator to the beginning of the copy range.
- Returns an iterator after the last copied element in the copy range.

```
std::vector<int> v{ 1, 2, 3, 4 };
std::vector<int> res(4);

std::vector<int>::iterator it = std::reverse_copy(v.begin(),
v.end(), res.begin());

// v is { 1 2 3 4 }
// res is { 4 3 2 1 }
// 'it' points to the element after 1 in res
```

rotate

For a range, rotates the elements (in a circular fashion) so that a specified becomes the new first element, and every other element is shifted over.

- Pass in an iterator to the beginning of the range, an iterator to the element we want to be the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4 };

std::vector<int>::iterator it = v.begin() + 2; // Points to 3

std::rotate(v.begin(), it, v.end());

// v is now { 3 4 1 2 }
```

rotate_copy

For a range, copies the elements into a new range if the elements were rotating (in a circular fashion) so that a specified becomes the new first element, and every other element is shifted over. The original range is untouched.

- Pass in an iterator to the beginning of the original range, an iterator to the element we want to be the beginning of the copy range. an iterator to the end of the original range, and an iterator to the beginning of the copy range.
- Returns an iterator after the last copied element in the copy range.

```
std::vector<int> v{ 1, 2, 3, 4 };
std::vector<int> res(4);

std::vector<int>::iterator it1 = v.begin() + 2; // Points to 3
std::vector<int>::iterator it2 = std::rotate_copy(v.begin(), it1,
v.end(), res.begin());

// v is still { 1 2 3 4 }
// res is { 3 4 1 2 }
// 'it2' points to the element after 2 in res
```

random_shuffle

For a range, rearranges the elements randomly. (`random_shuffle` uses `rand()` as the random number generator.)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4 };

std::random_shuffle(v.begin(), v.end());
// v is { 1 4 2 3 }
```

shuffle

For a range, rearranges the elements randomly using a uniform random number generator (that you must provide).

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and a random number generator.
- Returns void.

You should really be using `random` or some other library to provide a random number generator. Since the code example isn't simple, I'll just provide a link to cplusplus.com's shuffle page for sample code..

Partitioned

Operate on partioned data, which is split in half based on a certain condition.

is_partitioned

Check if the data in a range is partitioned according to a condition you specify. (Ex. Are the odd elements separated from the even elements?)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and a lambda that returns a bool.
- Returns a bool. True if the data is in the beginning all return true and the data afterward all return false. Returns false otherwise.

```
std::vector<int> v{ 5, 3, 1, 8, 4, 6 };

auto lambdaIsOdd = [](int i) { return i % 2 != 0; };

bool isPartitioned = std::is_partitioned(v.begin(), v.end(),
lambdaIsOdd); // true
```

Note that if the data is all false and then all true, `is_partitioned` will return false. Make sure your lambda will return "true" for the front data. For example..

```
std::vector<int> v{ 5, 3, 1, 8, 4, 6 };

auto lambdaIsEven = [](int i) { return i % 2 == 0; };

bool isPartitioned = std::is_partitioned(v.begin(), v.end(),
lambdaIsEven); // false
```

isPartitioned is false even though the data is clearly partitioned. That’s because the first partition returns false and the second partition returns true.

partition

Rearranges the elements in a range so that the first group returns true for the condition you specify. (Ex. Move all odd elements to the front.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and a lambda that returns a bool.

- Returns an iterator to the first element of the second partition.

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7 };

auto lambdaIsOdd = [](int i) { return i % 2 != 0; };

std::vector<int>::iterator it = std::partition(v.begin(), v.end(),
lambdaIsOdd);

// v is now { 1 7 3 5 4 6 2 }
```

stable_partition

Rearranges the elements in a range so that the first group returns true for the condition you specify. (Ex. Move all odd elements to the front.) Order is preserved.

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and a lambda that returns a bool.
- Returns an iterator to the first element of the second partition.

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7 };

auto lambdaIsOdd = [](int i) { return i % 2 != 0; };

std::vector<int>::iterator it = std::stable_partition(v.begin(),
v.end(), lambdaIsOdd);

// v is now { 1 3 5 7 2 4 6 }
```

partition_copy

For a range, copies the data which is true for a condition we specify into some range, and copies the data that is false into some other range.

- Pass in an iterator to the beginning of the original range, an iterator to the end of the original range, an iterator to the beginning of the “true” range, an iterator to the end of the “false” range, and a lambda that returns a bool.
- Returns a `pair` of iterators to the element after the last entered element for each of the copy ranges.

```
std::vector<int> v1{ 1, 2, 3, 4, 5, 6, 7 };
std::vector<int> v2(7);
```

```
std::vector<int> v3(7);

auto lambdaIsOdd = [](int i) { return i % 2 != 0; };

std::pair<std::vector<int>::iterator, std::vector<int>::iterator> p
= std::partition_copy(v1.begin(), v1.end(), v2.begin(), v3.begin(),
lambdaIsOdd);

// v2 is now { 1 3 5 7 0 0 0 } (The zeroes are from initialization.)
// v3 is now { 2 4 6 0 0 0 0 } (The zeroes are from initialization.)
```

partition_point

Get the first element in a partitioned range that returns false for a given condition. (The element that begins the partition.)

- Pass in an iterator to the beginning of the range, an iterator to the end of the range, and a lambda that returns a bool.
- Returns an iterator to the element.

```
std::vector<int> v{ 5, 3, 1, 8, 4, 6 };

auto lambdaIsOdd = [](int i) { return i % 2 != 0; };

std::vector<int>::iterator it = std::partition_point(v.begin(),
v.end(), lambdaIsOdd);
// Points to 8
```

Sorting

Sorting data.

sort

Sort the elements in a container for the given range.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };

std::sort(v.begin(), v.end());

// v is now { 1, 2, 3, 5, 7 }.
```

stable_sort

Sort the elements in a container for the given range, but the sort is stable.

A “stable” sort means that for equal elements, an element that was ahead of another before the sort will be ahead after the sort.

```
std::vector<int> v{ 5, 3, 7, 3, 2, 1 }; // { 5, 3a, 7, 3b, 2, 1 }
std::stable_sort(v.begin(), v.end());
// v is now { 1, 2, 3a, 3b, 5, 7 }
```

Stable sort is best used for objects.

partial_sort

For a position in a range, make sure all the elements from the beginning of the range up to that position are the smallest elements in the range and are sorted. (The rest of the elements (which are up to the position until the end of the range) aren’t in any specific order.)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and an iterator to the position we want to partially sort for.
- Returns void.

```
std::vector<int> v{ 5, 1, 3, 7, 6, 2, 9, 4 };
std::vector<int>::iterator it = v.begin() + 3;
std::partial_sort(v.begin(), v.end(), it);
// v ends up as { 1, 2, 3, 4, 5, 6, 7, 9}
```

In the code example, v ends up fully sorted, but only the first three elements are guaranteed to be sorted.

partialsortcopy

Copies the items from a range into a second range, in sorted order.

- Pass in an iterator to the beginning of the first range, and an iterator to the end of the first range, and an iterator to the beginning of the range we want to copy

into, and an iterator to the end of the range we want to copy into,

- Returns an iterator to the item after where we finished writing to in the copy range.

```
std::vector<int> v1{ 5, 1, 3, 7, 6, 2, 9, 4 };
std::vector<int> v2(6); // Initialize v2 with 6 elements.

std::vector<int>::iterator itStart = v1.begin() + 1;
std::vector<int>::iterator itEnd = v1.end() - 1;

std::partial_sort_copy(itStart, itEnd, v2.begin(), v2.end());

// v2 ends up as { 1, 2, 3, 6, 7, 9}
// 5 and 4 are skipped since they aren't in range. v2 is in sorted
order.
```

is_sorted

Check whether a given range is sorted.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns a bool. True if the range is sorted, false if it's not sorted.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };

bool isSorted = std::is_sorted(v.begin(), v.end()); // false
```

issortuntil

Finds the first element in a range that isn't sorted.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns an iterator to the first unsorted element.

```
std::vector<int> v{ 2, 3, 4, 1, 5 };
```

```
std::vector<int>::iterator it = std::is_sorted_until(v.begin(),
v.end());
// Points to the 1
```

nth_element

For the position that you specify in a range, places the element at that position that would be there if the range was sorted. (“What would the nth element be if this range was sorted” without actually sorting the entire range.)

- Pass in an iterator to the beginning of the range, an iterator to the position we want to find out about, and an iterator to the end of the range.
- Returns void. (The range is edited, though.)

```
std::vector<int> v{ 5, 3, 7, 9, 4 };
// v sorted would be { 3, 4, 5, 7, 9 }
std::vector<int>::iterator it = v.begin() + 1; // 2nd element in v.
std::nth_element(v.begin(), it, v.end());
int element = v[1]; // 4
```

Binary Search

These algorithms are about binary searching O(log N) a sorted range for a value.

lower_bound

Returns an iterator for the lower bound of an element in a range.

“Lower bound” means the first element that is not less than the element we’re searching for.

For example, for vector [3, 3, 4, 4, 4, 5, 7], lower*bound(4) points to the very first 4 (at index 2). lower*bound(6) points to 7, since that’s the first element not less than 6.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and the element you want to search for.
- Returns an iterator to the element. (Or to the container’s end if the element doesn’t exist.)

```
std::vector<int> v{ 3, 3, 4, 4, 4, 5, 7 };

std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(),
4);

int index = it - v.begin(); // 2
```

upper_bound

Returns an iterator for the upper bound of an element in a range.

"Upper bound" means the first element that is greater than the element we're searching for.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and the element you want to search for.
- Returns an iterator to the element. (Or to the container's end if the element doesn't exist.)

```
std::vector<int> v{ 3, 3, 4, 4, 4, 5, 7 };

std::vector<int>::iterator it = std::upper_bound(v.begin(), v.end(),
4);

int index = it - v.begin(); // 5
```

equal_range

Finds the starting index and ending index for an element in a range. (Meaning if we have duplicates of an item in a container, and we want to find out the range of where those items are.)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and the element you want to search for.
- Returns an `pair` of iterators of the element type.

```
std::vector<int> v{ 3, 3, 4, 4, 4, 5, 7 };

std::pair<std::vector<int>::iterator, std::vector<int>::iterator> p
= std::equal_range(v.begin(), v.end(), 4);

std::vector<int>::iterator startIterator = p.first;
std::vector<int>::iterator endIterator = p.second;
```

```
int startIndex = startIterator - v.begin(); // 2
int endIndex = endIterator - v.begin(); // 5
```

binary_search

Returns true if an element exists in a sorted range. Returns false otherwise. Does a binary search to find the item.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range, and the element you want to search for.
- Returns a bool.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };
std::sort(v.begin(), v.end());
bool twoExists = std::binary_search(v.begin(), v.end(), 2); // true
```

Sorted Data Operations

Is your data sorted? If yes, then do these efficient operations on them.

merge

Given two sorted ranges, combine them to form one larger sorted range.

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, an iterator to the beginning of the second range, an iterator to the end of the second range, and an iterator to the beginning of the resulting range that you want to place all of the data into.
- Returns an iterator to the end of the resulting range after placing the last item into it.

```
std::vector<int> v1{ 1, 3, 5, 9 };
std::vector<int> v2{ 2, 4, 6, 7 };
std::vector<int> res(8);
std::vector<int>::iterator it = std::merge(v1.begin(), v1.end(),
v2.begin(), v2.end(), res.begin());
// res is { 1, 2, 3, 4, 5, 6, 7, 9 }
// 'it' points to the element after 9 in res
```

inplace_merge

Given two consecutive sorted ranges (in the same container), do an in-place merge to sort the entire range.

- Pass in an iterator to the beginning of the first sorted range, an iterator to the beginning of the second sorted range, and an iterator to the end of the second sorted range.
- Returns void.

```
std::vector<int> v{ 6, 7, 8, 1, 2, 3 };
std::inplace_merge(v.begin(), v.begin()+3, v.end());
// v is now { 1, 2, 3, 6, 7, 8 }
```

includes

Check if a given a sorted range contains a second (sorted) range (that you specify). (Ex. Does “abcmopxyz” contain “mop”?)

- Pass in an iterator to the beginning of the first sorted range, an iterator to the end of the first sorted range. an iterator to the beginning of the second sorted range, and an iterator to the second sorted range.
- Returns a bool. True if the first range contains the second range, false if it doesn’t.

```
std::string s = "abcmopxyz";
std::string t = "mop";
bool containsMop = std::includes(s.begin(), s.end(), t.begin(),
t.end()); // true
```

set_union

Given two sorted ranges, creates a new sorted range from the elements that exist in either range. (The union.)

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, an iterator to the beginning of the second range, an iterator to

the end of the second range, and an iterator to the beginning of the resulting range that you want to place all of the data into.

- Returns an iterator to the element after the last element we placed in the resulting range.

```
std::vector<int> v1{ 1, 2, 3 };
std::vector<int> v2{ 1, 1, 2, 4 };
std::vector<int> res(7);

std::vector<int>::iterator it = std::set_union(v1.begin(), v1.end(),
v2.begin(), v2.end(), res.begin());

// res is { 1 1 2 3 4 0 0 } (Zeroes were from the initialization)
// 'it' points to the element after 4 in res.
```

Notice that there are only two 1's in the resulting range. `set_union` will take the maximum number of an element from one range; it won't just add them.

set_intersection

Given two sorted ranges, created a new sorted range from the elements that exist in both ranges. (The intersection.)

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, an iterator to the beginning of the second range, an iterator to the end of the second range, and an iterator to the beginning of the resulting range that you want to place all of the data into.
- Returns an iterator to the element after the last element we placed in the resulting range.

```
std::vector<int> v1{ 1, 2, 3 };
std::vector<int> v2{ 1, 1, 2, 4 };
std::vector<int> res(7);

std::vector<int>::iterator it = std::set_intersection(v1.begin(),
v1.end(), v2.begin(), v2.end(), res.begin());

// res is { 1 2 0 0 0 0 0 } (Zeroes were from the initialization)
// 'it' points to the element after 2 in res.
```

set_difference

Given two sorted ranges, created a new sorted range from the elements that exist in the first range that don't exist in the second range. (The difference.)

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, an iterator to the beginning of the second range, an iterator to the end of the second range, and an iterator to the beginning of the resulting range that you want to place all of the data into.
- Returns an iterator to the element after the last element we placed in the resulting range.

```
std::vector<int> v1{ 1, 2, 3 };
std::vector<int> v2{ 1, 1, 2, 4 };
std::vector<int> res(7);

std::vector<int>::iterator it = std::set_difference(v1.begin(),
v1.end(), v2.begin(), v2.end(), res.begin());

// res is { 3 0 0 0 0 0 0 } (Zeroes were from the initialization)
// 'it' points to the element after 3 in res.
```

setsymmetricdifference

Given two sorted ranges, created a new sorted range from the elements that exist in one of the ranges but doesn't exist in the other range. (The symmetric difference.)

- Pass in an iterator to the beginning of the first range, an iterator to the end of the first range, an iterator to the beginning of the second range, an iterator to the end of the second range, and an iterator to the beginning of the resulting range that you want to place all of the data into.
- Returns an iterator to the element after the last element we placed in the resulting range.

```
std::vector<int> v1{ 1, 2, 3 };
std::vector<int> v2{ 1, 1, 2, 4 };
std::vector<int> res(7);

std::vector<int>::iterator it =
std::set_symmetric_difference(v1.begin(), v1.end(), v2.begin(),
v2.end(), res.begin());

// res is { 1 3 4 0 0 0 0 } (Zeroes were from the initialization)
// 'it' points to the element after 4 in res.
```

Notice the 1 in res. That's there because v2 has an extra 1 that v1 does not have.

Heap

Algorithms regarding heap data structures.

A "heap" is a binary tree that always has maximum element as the root node (top of the tree). It's used in priority queues. (It can also have the min element as the root.)

make_heap

Rearranges the items in a range so that they form a heap.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 1, 2, 3, 4 };
std::make_heap(v.begin(), v.end());
// v is now { 4 2 3 1 }
```

push_heap

If the first elements of a range are a heap, but the last item isn't, then `push_heap` will place that last item into it's correct position.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 4, 2, 3, 1 }; // v is already a heap.
v.push_back(5); // v is no longer a heap.
std::push_heap(v.begin(), v.end());
// v is now { 5 4 2 3 1 }, which is a heap.
```

pop_heap

Given a range for a heap, places the max element at the end of the range, and rearranges the rest of the elements to be a heap again.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 5, 4, 2, 3, 1 }; // v is a heap.
std::pop_heap(v.begin(), v.end());
// v is now { 4 2 3 1 5 }. (The first 4 elements are a heap.)
```

sort_heap

If a range is a heap, rearrange it into sorted order.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns void.

```
std::vector<int> v{ 5, 4, 2, 3, 1 }; // v is a heap.
std::sort_heap(v.begin(), v.end());
// v is now { 1 2 3 4 5 }.
```

is_heap

Check if a range is a heap.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns a bool. True if the range is a heap, false if it's not a heap.

```
std::vector<int> v{ 5, 4, 2, 3, 1 }; // v is a heap.
bool isHeap = std::is_heap(v.begin(), v.end()); // true
```

isheapuntil

Find the first place in a range that makes the range fail being a heap.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns an iterator to the first element that makes the range fail being a heap.

```
std::vector<int> v{ 5, 4, 2, 3, 1, 200 };

std::vector<int>::iterator it = std::is_heap_until(v.begin(),
v.end()); // Points to 200
```

Min and Max

These algorithms are related to finding the minimum/maximum element in a range, or when comparing two items.

min

Compares two items and returns the smaller one.

- Pass in two items (`int` s, for example) as parameters.
- Returns the item's type.

```
int m = std::min(5, 2); // 2
```

max

Compares two items and returns the larger one.

- Pass in two items (`int` s, for example) as parameters.
- Returns the item's type.

```
int m = std::max(5, 2); // 5
```

minmax

Kind of useless in my opinion. Not even going to post the implementation. see `minmax_element` for something useful.

min_element

Returns an iterator to the smallest element for a given range.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns an iterator of the type of the collection.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };
std::vector<int>::iterator it = std::min_element(v.begin(),
v.end());
// 'it' points to the 1 in v
```

max_element

Returns an iterator to the largest element for a given range.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns an iterator of the type of the collection.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };
std::vector<int>::iterator it = std::max_element(v.begin(),
v.end());
// 'it' points to the 7 in v
```

minmax_element

Returns a `pair` of iterators to the smallest and the largest elements for a given range. (The smallest is the first item, and the largest is the second item.)

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns an `std::pair` of iterators of type of the collection.

```
std::vector<int> v{ 5, 3, 7, 2, 1 };

std::pair<std::vector<int>::iterator, std::vector<int>::iterator> p
= std::minmax_element (v.begin(), v.end());

int smallest = *p.first; // 1
int largest = *p.second; // 7
```

Permutations

Permute data into it's next transformation.

next_permutation

Rearranges the items in the specified range into the next lexographically greater permutation.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns a bool. True if the range was transformed, false otherwise (if there was no such possible permutation).

```
std::string s = "abc";

std::next_permutation(s.begin(), s.end());

// s is now "acb"
```

prev_permutation

Rearranges the items in the specified range into the previous lexographically ordered permutation.

- Pass in an iterator to the beginning of the range, and an iterator to the end of the range.
- Returns a bool. True if the range was transformed, false otherwise (if there was no such possible permutation).

```
std::string s = "acb";

std::prev_permutation(s.begin(), s.end());
```

```
// s is now "abc"
```

End Notes

Notes

- The `auto` keyword was purposely not used often in this documentation, but use it in your real code.
- Note that I say “range” a lot, even though I specifically use `vector.begin()` and `vector.end()` most of the time. Remember that your range can be anywhere in the container you want.
- Some algorithms also have an overloaded version to provide a custom comparator (for example, `stable_sort`), but I've ommitted those to keep the examples short.
- Pay careful attention to the ordering of the parameters, some are not intuitive. (Ex. `nth_element`)

Useful? buy me a coffee

For more checkout *logicalbee*

References

- cplusplus.com’s list of STL algorithms.
- Jonathan Boccara’s CppCon Talk about the STL Algorithms.

Competitive Programming | Cheatsheet | Cpp | C Programming | Algorithms

Follow

Written by Sushanth Kurdekar

17 Followers · Editor for Logical Bee

https://www.youtube.com/logicalbee

More from Sushanth Kurdekar and Logical Bee

Sushanth Kurdekar

How to fix macbook won't turn on—2020

I use my macbook day to day basis for my work, video editing, and freelancing stuff. one day suddenly my mac is not booting up, I'm...

1 min read · Aug 25, 2020

Sushanth Kurdekar in Logical Bee

How To Start A Blog In 2020 And Earn $5,000 Every Month

Short and simple, step-by-step guide on how to start a blog? and How to make money from that blog.

9 min read · Jul 28, 2020

See all from Sushanth Kurdekar

See all from Logical Bee

Recommended from Medium

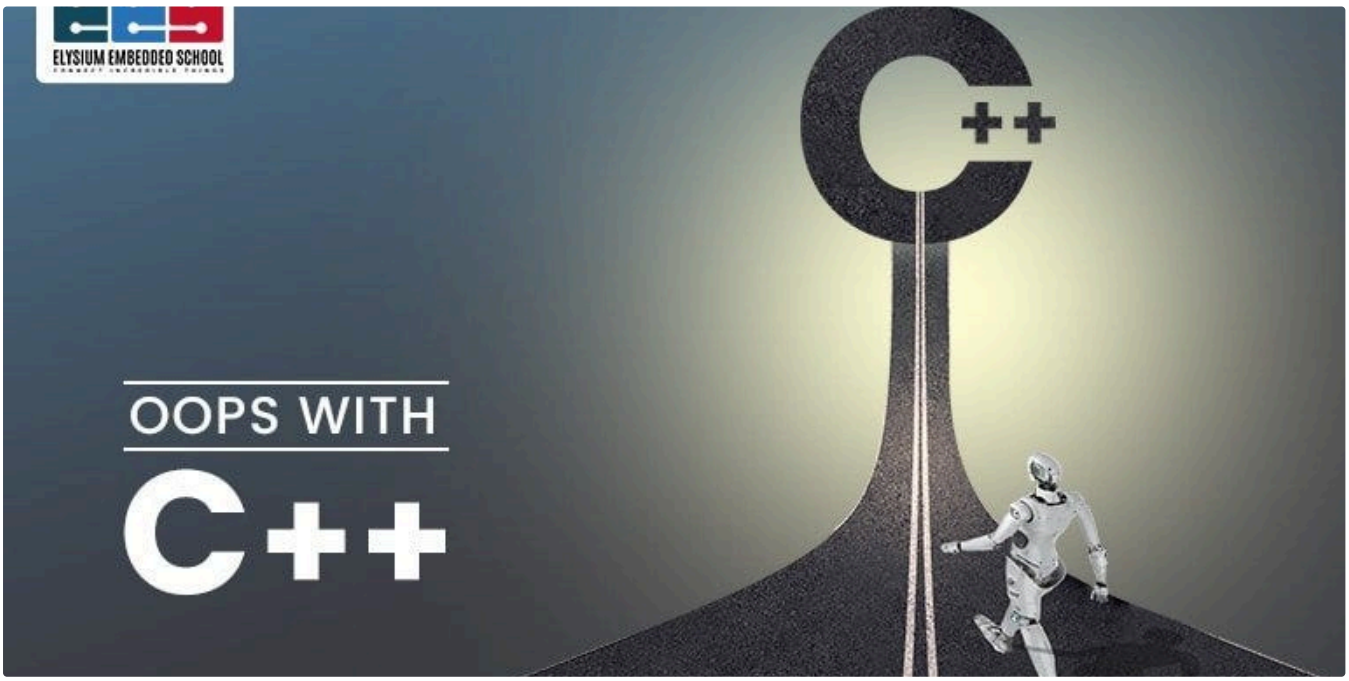
Serhii Olendarenko

Why C++ is a bad language

Or at least why is it considered to be so?

12 min read · Mar 11, 2024

116 1



Ankit Singh

“Mastering Object-Oriented Programming (OOP) in C++: A Comprehensive Guide”

Before we start learning about OOP (Object-Oriented Programming), let's figure out why it's important. We'll explore what we used before...

10 min read · Dec 24, 2023

60

Lists

Practical Guides to Machine Learning

10 stories · 1219 saves

General Coding Knowledge

20 stories · 1037 saves

Staff Picks

604 stories · 842 saves

Natural Language Processing

1309 stories · 794 saves

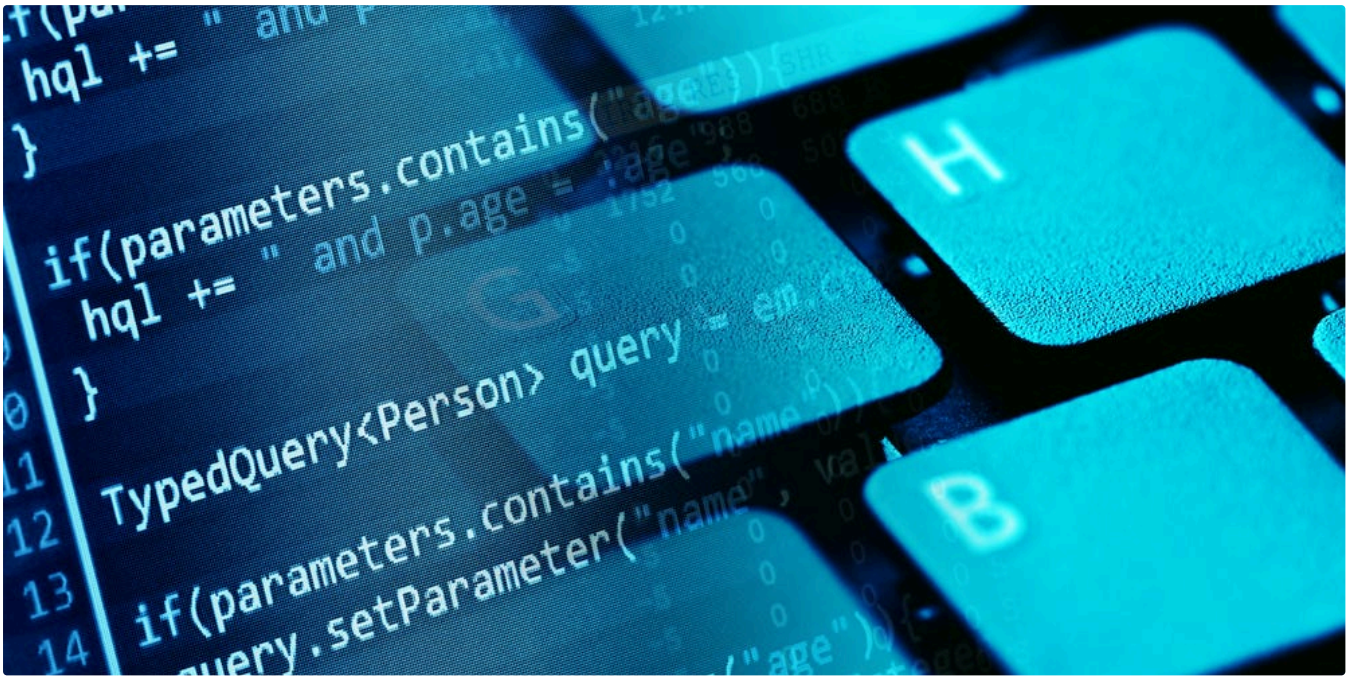
TechHara

C++—implement channel from scratch

A channel is a high-level communication abstraction that allows threads to send and receive data to each other. Channels are a fundamental...

5 min read · Oct 29, 2023

2 1



BeyondVerse

Algorithms for String Manipulation and Matching

In the realm of computer science, string manipulation and matching algorithms play a pivotal role in processing and analyzing textual .....

12 min read · Nov 30, 2023

19

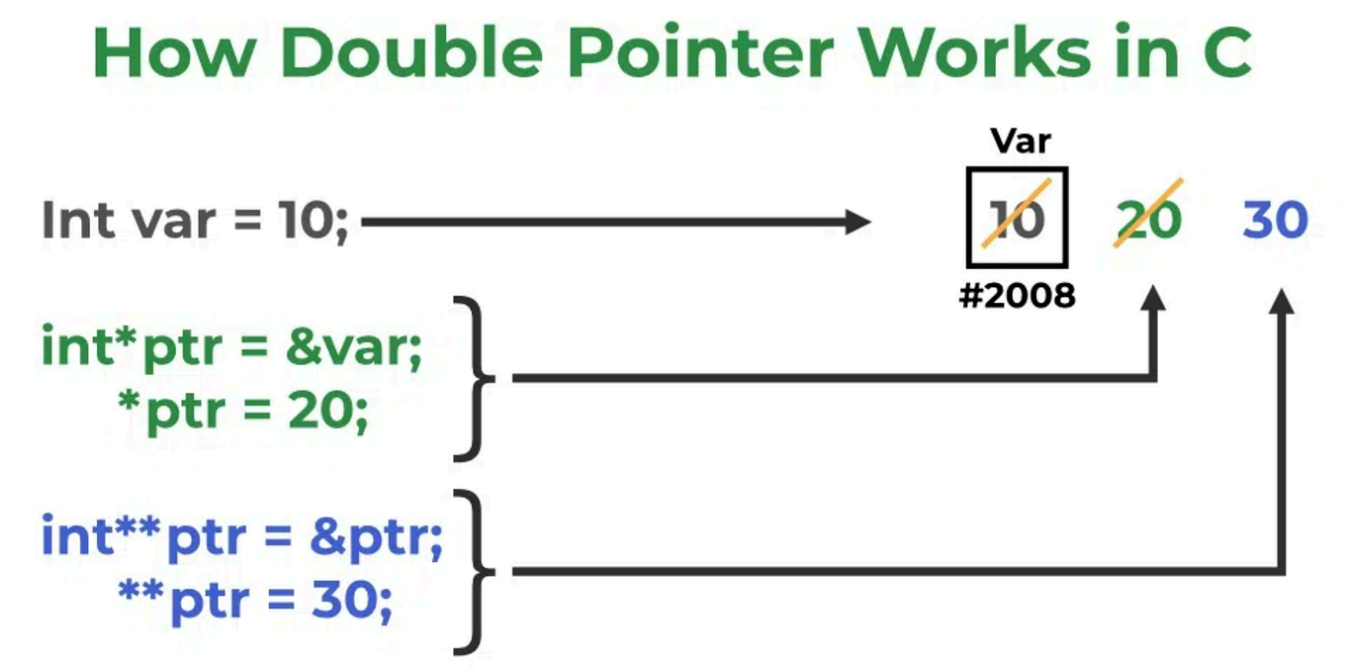
Muralikrishnan Rajendran

Unraveling the Speed Disparity: Why Python Lags Behind Java and C++

In the landscape of programming languages, Python's simplicity and readability have earned it a cherished spot among beginners and seasoned...

15 min read · Oct 8, 2023

214



Yu-Cheng Kuo in Nerd For Tech

C Interview Questions 01: Pointers & Endianness

Function pointer, pointer to pointer, const int *ptr, & undefined behavior

10 min read · Dec 1, 2023

 12  2