C++ONLINE

FRANCES BUONTEMPO

TALK:

DON'T BE NEGATIVE

2025

Our mission:

Remove/ignore negative numbers from a container/range/stuff

- I edit ACCU’s Overload magazine
  - https://accu.org/journals/nonmembers/overload_cover_members/
- Programmer (C++ plus some Python and C#) (mostly in finance)
- Author

O'REILLY
Introducing C++
The Easy Way to Start Learning Modern C++
Early Release RAW & UNEDITED
Frances Buontempo
Learn C++ by Example
Covers versions 11 to 23
MANNING
Frances Buontempo
The Pragmatic Programmers
Genetic Algorithms and Machine Learning for Programmers
Create AI Models and Evolve Solutions
Frances Buontempo
edited by Tammy Coron

Introducing C++

Learn C++ by Example

Genetic Algorithms and Machine Learning for Programmers

Where am I?

- https://mastodon.social/@fbuontempo
- https://bsky.app/profile/fbuontempo.bsky.social
- https://x.com/fbuontempo
    - (formerly https://twitter.com/fbuontempo)
- https://www.linkedin.com/in/francesbuontempo/
- https://buontempoconsulting.blogspot.com/
- (Sometimes)

Outline

- Start sensibly
    - {-1, 4, -7, 0}
- Try various containers
    - std::vector<int>
    - Generalise vector and int
- Learn stuff
- Try silly things
- Learn more stuff

Display non-negatives?

```
for(int x : {-1, 4, -7, 0})
{
    if(x >= 0)
        std::cout << x << '\n';
}
```

Display non-negatives?

```
for(int x : {-1, 4, -7, 0})
{
    if(x >= 0)
        std::println("{}", x);
}
```

But what if you want to keep the new values?

- Change the values - in place?
- Or create another container?
- Or a view?
- Let’s use some algorithms

std::erase_if from C++20

```
std::vector<int> erase_negatives(std::vector<int> numbers)
{
    std::erase_if(numbers, [](int x) {return x < 0;});
    return numbers;
}
```

Previously

```
std::vector numbers{-1, 4, -7, 0};
auto it = std::remove_if(numbers.begin(), numbers.end(),
            [](int x){ return x < 0; });
std::println("Numbers");
for(auto x : numbers)
{
    std::println("{}", x);
}
```

?

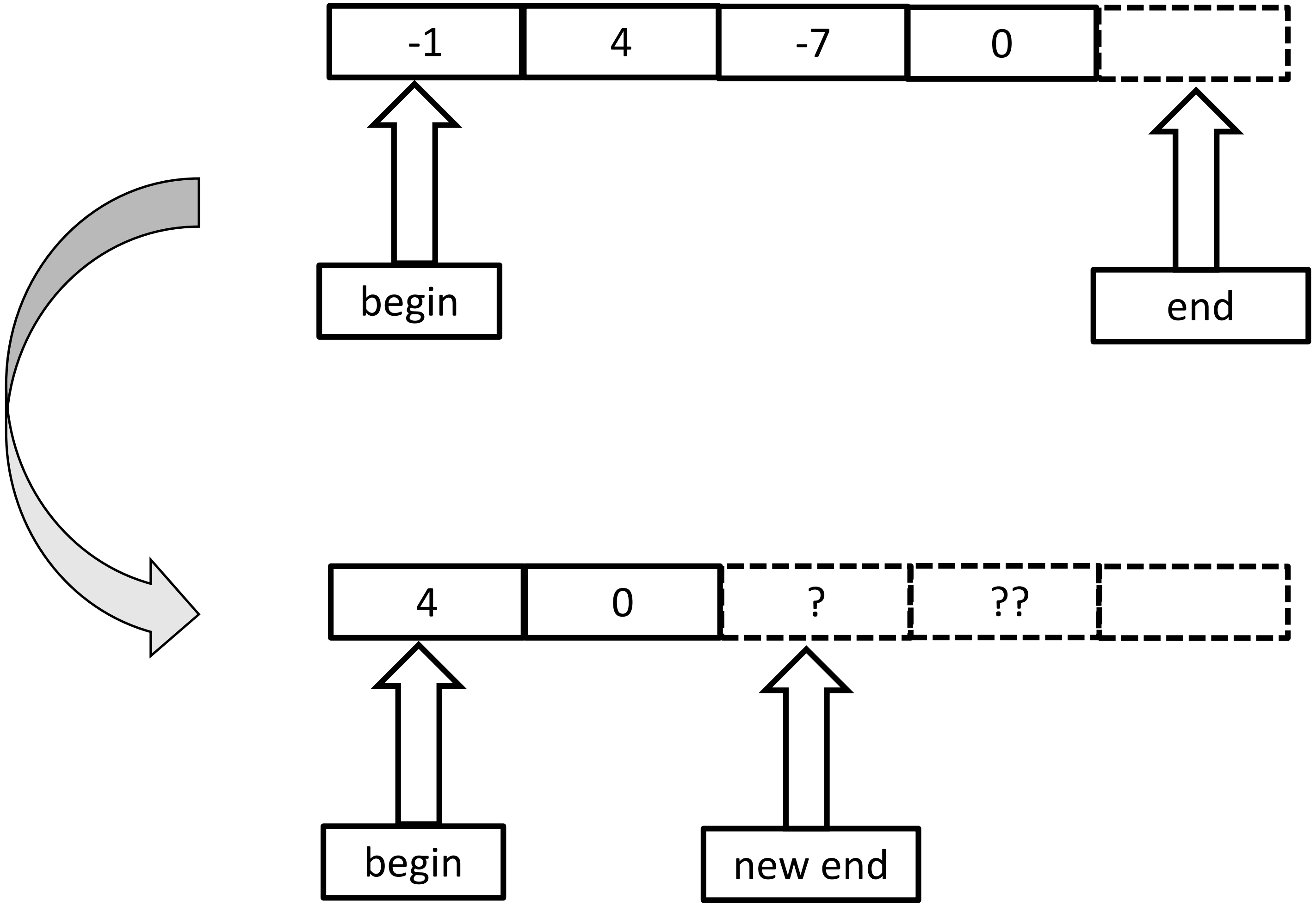
Maybe….

Numbers:

{-1, 4, -7, 0}

End as:

{4, 0, -7, 0}



Remove-erase idiom – use it

```
std::println("stopping at it");
for(auto x : std::ranges::subrange(numbers.begin(), it))
{
   std::println("{}", x);
}
numbers.erase(it, numbers.end());
std::println("Numbers after erase");
for(auto x : numbers)
{
   std::println("{}", x);
}
```

```
stopping at it
4
0
```

```
Numbers after erase
4
0
```

Questions

- We’re ignoring a return value from erase.
  - It’s an iterator to end or new end (vital for loops erasing stuff)
  - We also ignored erase_if (number erased)
- How do we test this?
  - Or post-conditions (e.g. contracts)
- Just the ints?
- Only vectors?
  - What other containers does this work for?

Generalize vectors for now

```
std::erase_if(numbers, [](int x) {return x < 0;});
```

Not just ints

```
template<typename T>
std::vector<T> vector_erase_negatives(std::vector<T> numbers)
{
   std::erase_if(numbers, [](T x) {return x < T{};});
   return numbers;
}
```

Too general?

```
using namespace std::string_literals;
std::vector words{"hello "s, "everyone!"s};
auto erm = vector_erase_negatives(words);

[](T x) {return x < T{};}
==
[](std::string x) {return x < std::string{};}
```

//hello everyone!

Concepts

```
template <typename T>
concept NumericType = std::integral<T> || std::floating_point<T>;

template<NumericType T>
std::vector<T> numeric_erase_negatives(std::vector<T> numbers)
{
    std::erase_if(numbers, [](T x) {return x < T{};});
    return numbers;
}
```

✓

```
using namespace std::string_literals;
std::vector words{"hello "s, "everyone!"s};
auto erm = numeric_erase_negatives(words);
```

✗

Pause for recap

- {-1, 4, -7, 0}
- std::erase_if
- std::remove_if then erase
- template<typename T> std::vector<T>
- template <typename T> **concept** NumericType
  - template<**NumericType** T> std::vector<T>
- **auto result = numeric_erase_negatives({-1, 4, -7, 0});**

Initializer list

```
template<NumericType T>
std::vector<T> numeric_erase_negatives(std::vector<T> numbers)
{
    std::erase_if(numbers, [](T x) {return x < T{};});
    return numbers;
}

auto result = numeric_erase_negatives<int>({-1, 4, -7, 0});
```

Not just vectors – attempt 1

```
template<typename T>
T templated_erase_negatives(T numbers)
{
    std::erase_if(numbers, [](auto x) {return x < 0;});
    return numbers;
}
```

Not just vectors – attempt 2

```
template<typename C, typename T = typename C::value_type>
C templated_erase_negatives(C numbers)
{
    std::erase_if(numbers, [](T x) {return x < T{};});
    return numbers;
}
```

See https://devblogs.microsoft.com/oldnewthing/20190619-00/?p=102599

Are we good?

```
std::vector numbers{-1, 4, -7, 0};
for(int x : templated_erase_negatives(numbers))
    std::println("{}", x);

using namespace std::string_literals;
auto word = templated_erase_negatives("help"s);
std::println("{}", word);

templated_erase_negatives<int>({-1, 4, -7, 0});
```

Which containers?

- vector, string good
    - and deque, list, set, multiset (why is there even a multiset?) …
- What about array?
- What about a map?
- (Fran wonders if other containers work or not)

Array

```
std::array a{-1, 4, -7, 0};
auto got = templated_erase_negatives(a);
```

```
error: no matching function for call to 'erase_if(std::array<int, 5>&,
templated_erase_negatives<std::array<int, 5> >(std::array<int,
5>)::<lambda(int)>)'
  147 |     std::erase_if(numbers, [](T x) {return x < T{};});
```

(Tries to match deque, string, list, map, vector, set)

Maps

- std::erase_if needs a predicate of the container’s value type
- std::erase_if(numbers, [](T x) {return x < T{};});

- A std::map’s key is key_type and value_type is **std::pair<const Key, T>**
- And it has a compare: key_compare

Map

```
std::map<int, char>  m{ {-1, 'c'}, {3, 'd'} };
template<typename C, typename T = typename C::value_type,
    typename K = typename C::key_type,
    typename Cmp = typename C::key_compare>
C templated_erase_negatives_special(C numbers)
{
    std::erase_if(numbers, [](T x) {return Cmp{}(x.first, K{});});
    return numbers;
}
```

Views

```
#include <ranges>
 std::vector input{-1, -2,  3};

 for(int x : input | std::views::filter([](int i) {return i>=0;}))
 {
    std::println("{}", x);
 }
```

More generally

```
for(int x : input | std::views::filter([](auto i) {return i>=decltype(i){};}))
{
    std::println("{}", x);
}
```

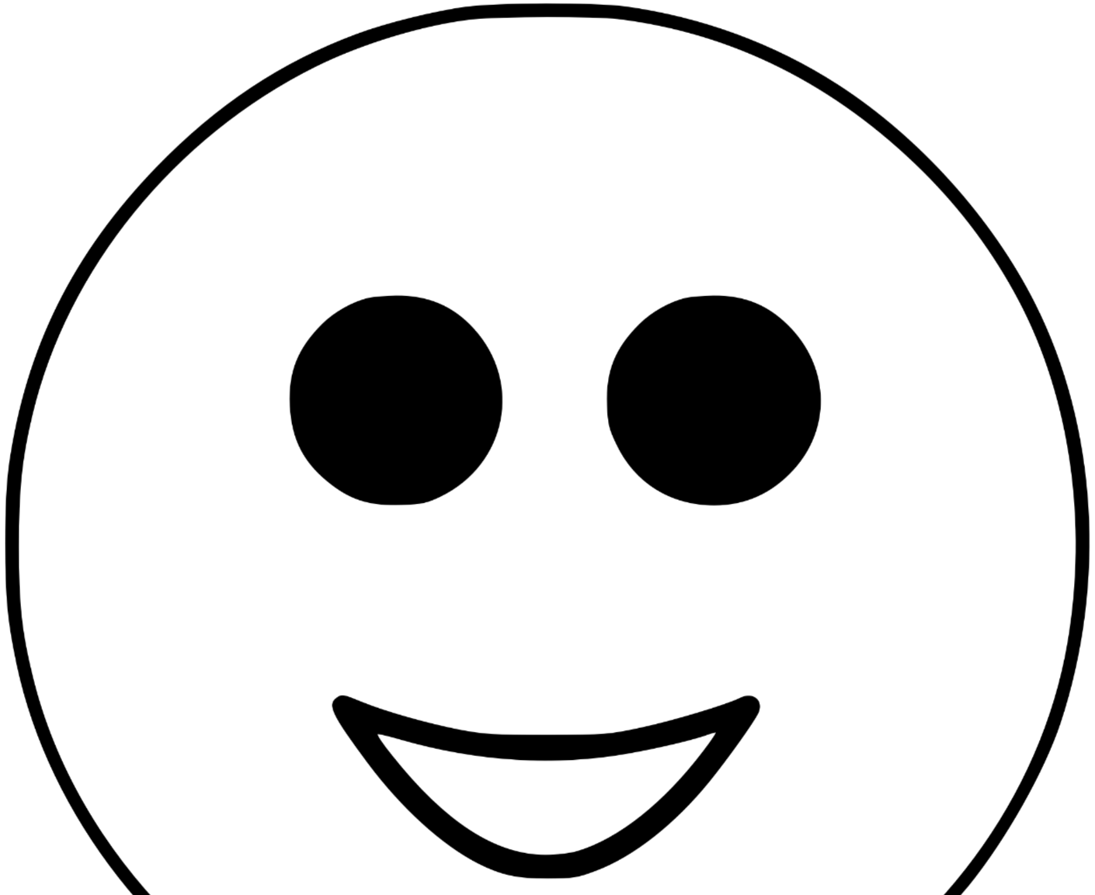
Auto means a template

```
for(int x : input |
  std::views::filter([]<typename T>(T i) {
        return i >= T{};
    })
  )
{
    std::println("{}", x);
}
```

Maps (again)

```
std::map<int, char>  m{ {-1, 'c'}, {3, 'd'} };
for (int value : m | std::views::keys
                   | std::views::filter([](int x){ return x >= 0; }))
    std::println("{}", value);

for (char value : m | std::views::values
                    | std::views::filter([](char x){ return x != 'c'; }))
    std::println("{}", value);
```

And now for something(s)… silly



Remove ALL the things

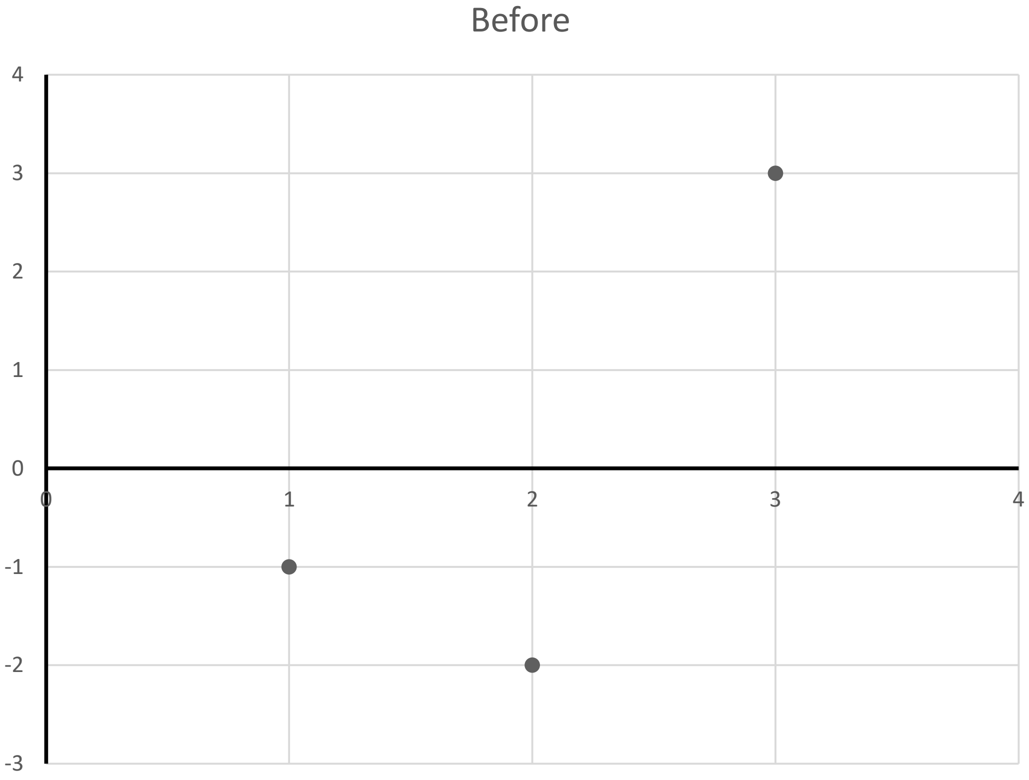
```
template<typename C>
C all_gone(C numbers)
{
    return C{};
}

std::vector input{-1, -2,  3};
auto got = all_gone(input);
```

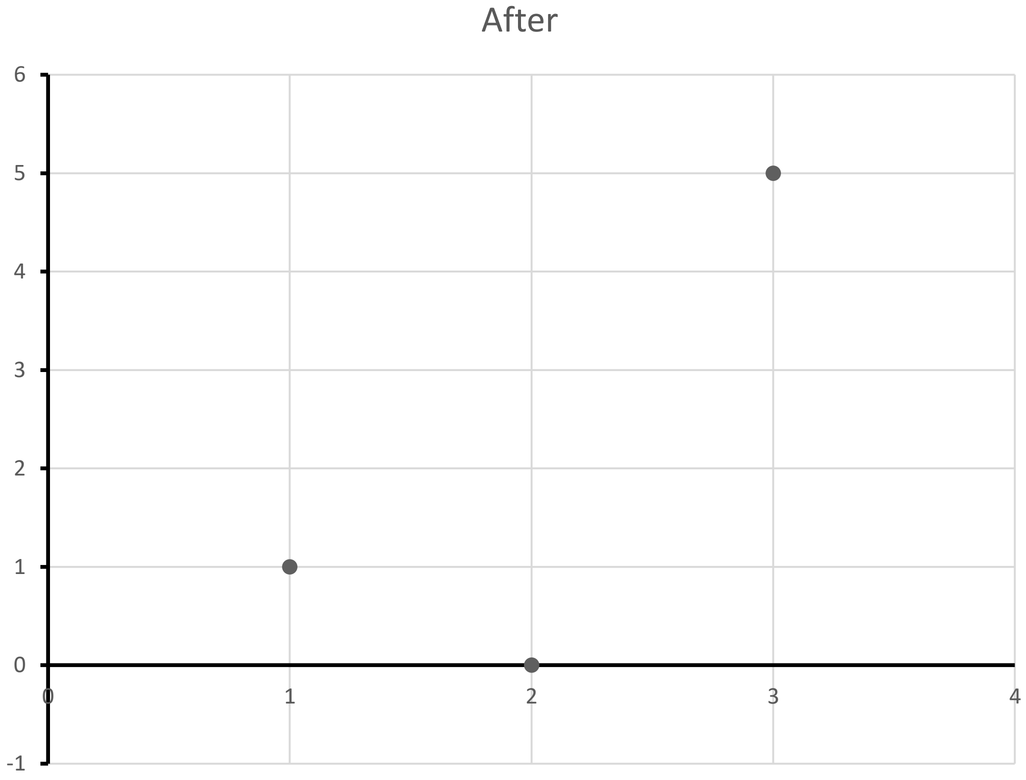
What does this do?

```
template<typename C>
C all_gone(C numbers)
{
    return C{};
}

std::array a{-1, 4, -7, 0, 2};
for(int x : all_gone(a))
    std::println("{}", x);
```

```
0
0
0
0
0
```

Increase ALL the things





Start at nothing

```
template<typename C>
C start_at_nothing(C numbers)
{
    auto least = *std::ranges::min_element(numbers);
    C output;
    std::ranges::transform(numbers, std::back_inserter(output),
        [least] (auto x) { return x - least;} );
    return output;
}

std::vector input{-1, -2,  3};
auto got = start_at_nothing(input);
```

Sorted?!

```
std::vector numbers{-1, 4, -7, 0, 2};
auto count = std::ranges::count_if(numbers,
        [](auto x) {return x < decltype(x){};});
std::ranges::sort(numbers); //default operator<
numbers.erase(numbers.begin(), numbers.begin() + count);
for(auto x: numbers)
{
    std::println("{}", x);
}
```

```
0
2
4
```

Don’t sort ALL the things

```
std::vector numbers{-1, 4, -7, 0, 2};
auto count = std::ranges::count_if(numbers,
            [](auto x) {return x < decltype(x){};});
std::ranges::nth_element(numbers, numbers.begin() + count);
numbers.erase(numbers.begin(), numbers.begin() + count);
for(auto x: numbers)
{
   std::println("{}", x);
}
```

```
0
2
4
```

Sorting wastes time – Partition instead

```
std::vector numbers{-1, 4, -7, 0, 2};
auto it = std::partition(numbers.begin(), numbers.end(),
                  [](auto x) { return x < 0;} );
std::vector non_negative(it, numbers.end());
```

At the start

```
std::vector numbers{-1, 4, -7, 0, 2};

auto it = std::partition(numbers.begin(), numbers.end(),
                          [](auto x) { return x >= 0;} );
std::vector non_negative_first(numbers.begin(), it);
```

Meh

- Partition looks suspiciously like remove_if.
  - BUT both put required elements first.
  - partition puts un-needed ones last {-1, 4, -7, 0}; -> {-1, -7, 4, 0} (or {4, 0, -7, -1}
  - remove_if does not specify what's last {-1, 4, -7, 0}; -> {4, 0, -7, 0};
  - Different complexity guarantees (remove is quicker)
- stuff.erase() no good for arrays
- Let's try more algos
  - Some proper Comp Sci
  - Some computational statistics

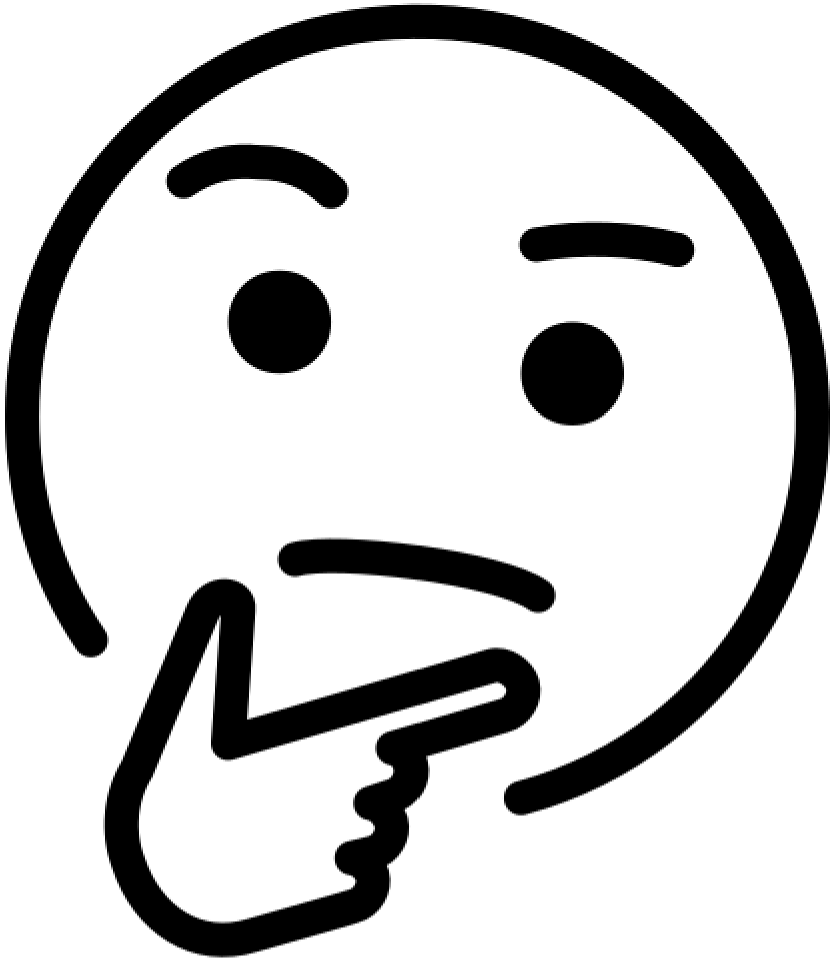
Recursion

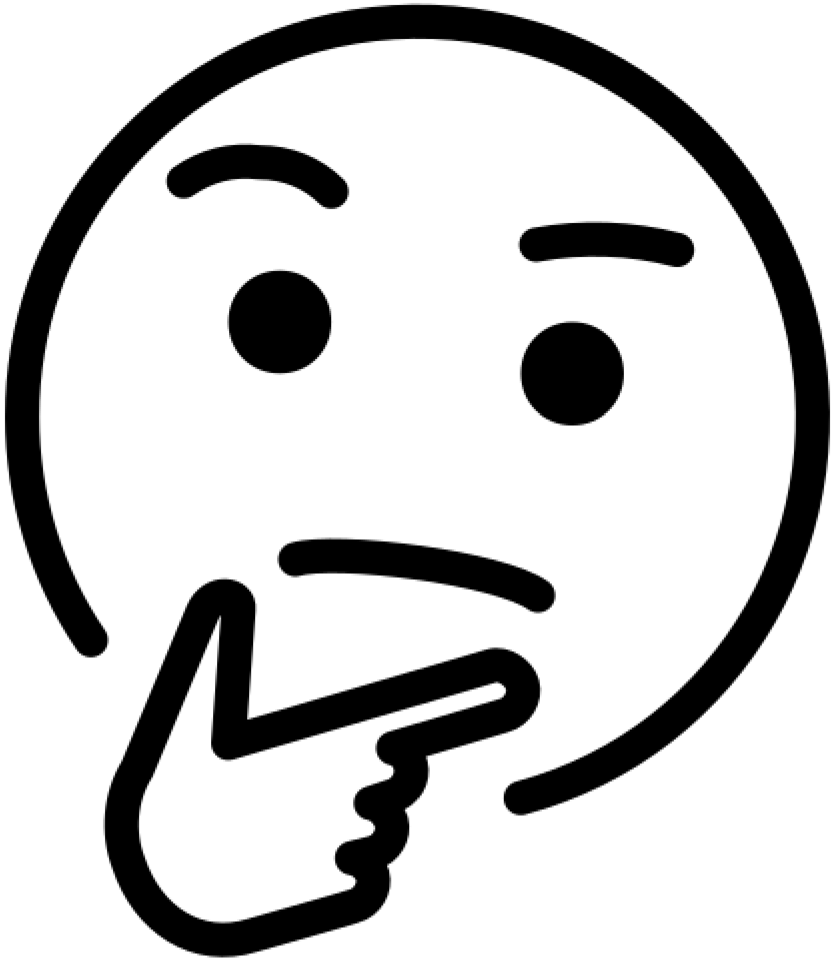
```
auto recurse(auto stuff)
{
    if(stuff.empty())
        return stuff;

    auto first = stuff.front();
    stuff.erase(stuff.begin());
    auto remains = recurse(stuff);

    if(first >= decltype(first){})
        remains.insert(remains.begin(), first);

    return remains;
}
```

Rejection sampling

```
std::vector input{-1, 4, -7, 0, 2};
auto gen = std::mt19937{std::random_device{}()};
auto d = std::uniform_int_distribution(0ul, input.size());
std::vector<int> out;
while(out.empty() || std::ranges::any_of(out, [](int x) { return x < 0; } )) {
    out.clear();
    size_t wanted = d(gen);
    std::ranges::sample(input, std::back_inserter(out), wanted, gen);
}
for(auto x: out)
    std::println("{}", x);
```

Double trouble





And now with doubles

```
std::vector numbers{-1.1, 4.0,
    std::numeric_limits<double>::quiet_NaN(),
    -7.7, 0.0, 2.5};
auto it = std::partition(numbers.begin(), numbers.end(),
    [](auto x) { return x < 0;} );
std::vector non_negative(it, numbers.end());
for(int x : non_negative)
{
    std::println("{}", x);
}
//Gives NaN, 4.0, 0.0, 2.5
```

At the start, again

```
auto it = std::partition(numbers.begin(), numbers.end(),
    [](auto x) { return x >= 0;} );
//Also gives NaN, 4.0, 0.0, 2.5
```

What did we reject?

```
std::vector numbers{-1.1, 4.0,
    std::numeric_limits<double>::quiet_NaN(),
    -7.7, 0.0, 2.5};
auto neg_it = std::partition(numbers.begin(),
         numbers.end(),
         [](auto x) { return x >= 0;} );
// Partition put NaN, 4.0, 0.0, 2.5 at the start (or end)
std::vector negatives(neg_it, numbers.end());
for(auto x : negatives)
    std::println("{}", x);
```

```
-7.7
NaN
-1.1
```

Sorting with NaNs: NaN != NaN

“It is undefined behaviour to call std::sort() on an array containing NaNs using the default comparator ”

See Tristan Brindle’s CppOnSea talk

https://www.youtube.com/watch?v=d3t9YAmpN50

Strong order

- Fortunately, IEEE 754 also defines a *total order* for all floats, including NaNs
- In C++20, std::strong_order() on floats will use the IEEE total order

```
auto it_again = std::partition(numbers.begin(),
    numbers.end(),
    [](auto x) { return std::is_lt(std::strong_order(x, 0.0));} );
std::vector non_negative_now(numbers.begin(), it_again);
for(auto x : non_negative_now)
    std::println("{}", x);
std::println("leaving");
std::vector no_nans(it_again, numbers.end());
for(auto x : no_nans)
    std::println("{}", x);
```

```
No negatives nans
-1.1
-7.7
leaving
0
4
nan
2.5
```

Unsigned…?

- Beware comparing unsigned or size_t with ints

```
int main() {
    long a = -100;
    unsigned short b = 100;
    std::cout << (a < b);   // true
    size_t c = 100;
    std::cout << (a < c);   // false
}
```

```
//CppInsights:
long a = static_cast<long>(-100);
unsigned short b = 100;
std::cout.operator<<((a < static_cast<long>(b)));
size_t c = 100;
std::cout.operator<<((static_cast<unsigned long>(a) < c));
```

- See https://www.cppstories.com/2022/safe-int-cmp-cpp20/
  - E.g. std::cmp_greater from <utility>

Sensible

In place:

```
std::erase_if(numbers, [](int x) {return x < 0;});
```

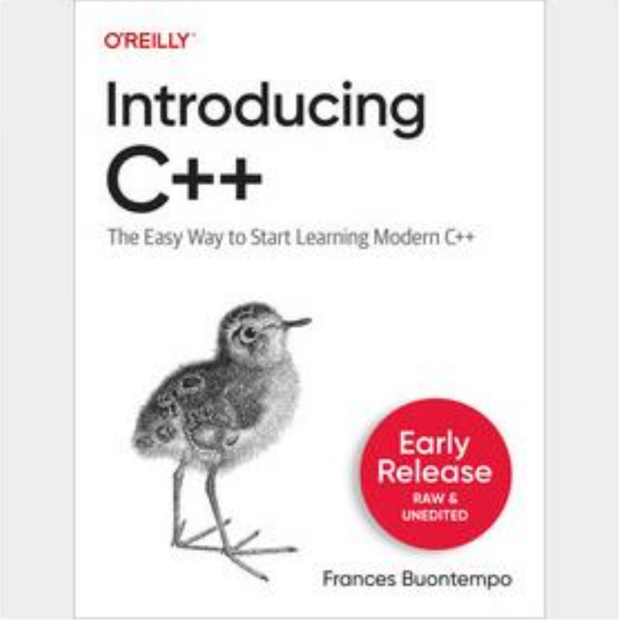
Copy:

```
std::vector<int> destination;
std::ranges::copy_if(input, std::back_inserter(destination),
                     [](int x) { return x >= 0; });
```

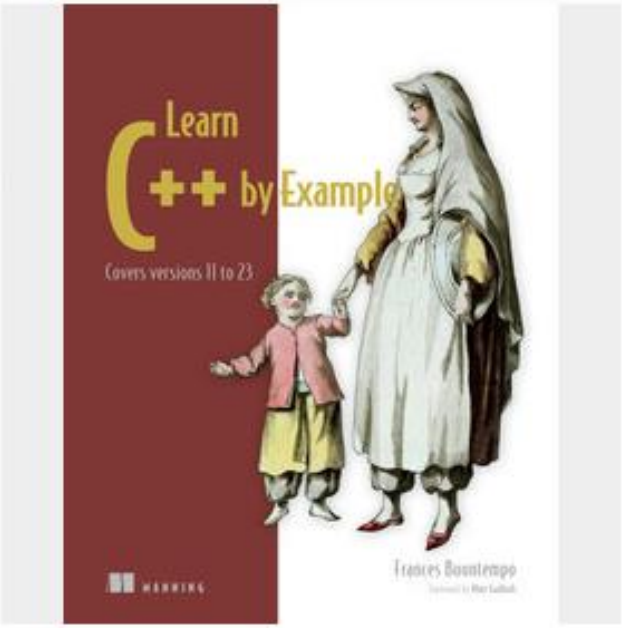
Homework

- Can you get something general that works for an initializer list?
  - What should it return?
- We didn’t use lower_bound
  - Sort, then values.erase(values.begin(),
    std::lower_bound(values.begin(), values.end(), 0));
- Try all the algorithms
- Try all the containers
- Practice
- Have fun

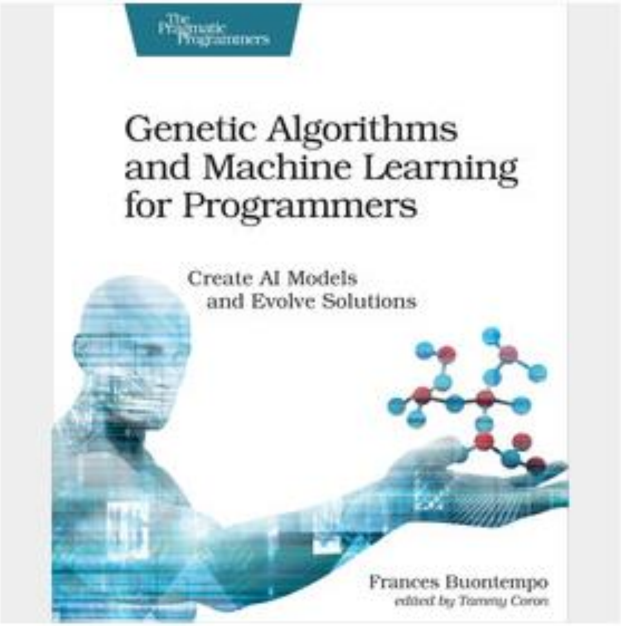
Books



Introducing C++



Learn C++ by Example


The Pragmatic Programmers
Genetic Algorithms and Machine Learning for Programmers
Create AI Models and Evolve Solutions
Frances Buontempo
edited by Tammy Coron

Genetic Algorithms and Machine Learning for Programmers