2022

Beginners Guide to C++'s Best Kept Secret std::algorithm

Mike Shah

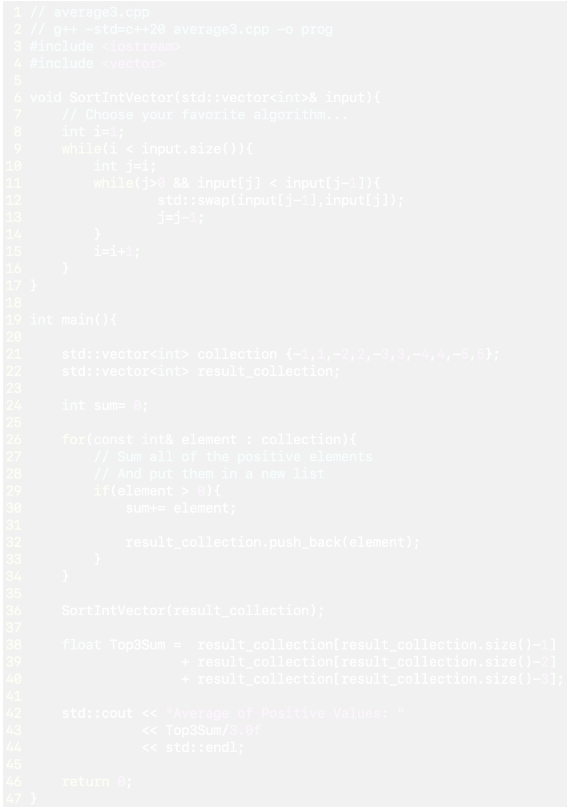
Social: [@MichaelShah]
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

11:00-12:00, Wed, 6th July 2022

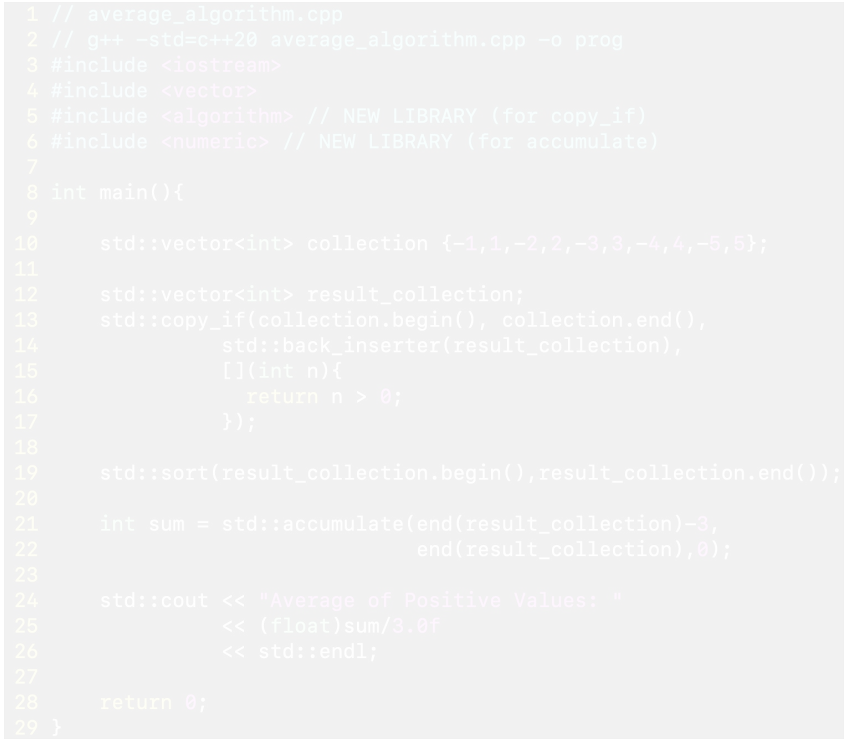
60 minutes | Introductory Audience

Goal(s) for today

Today’s talk is almost all motivation

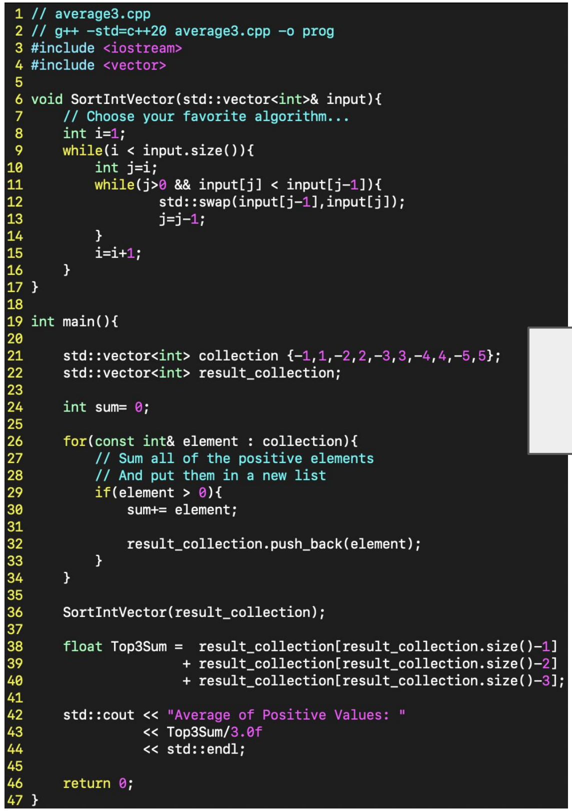


```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

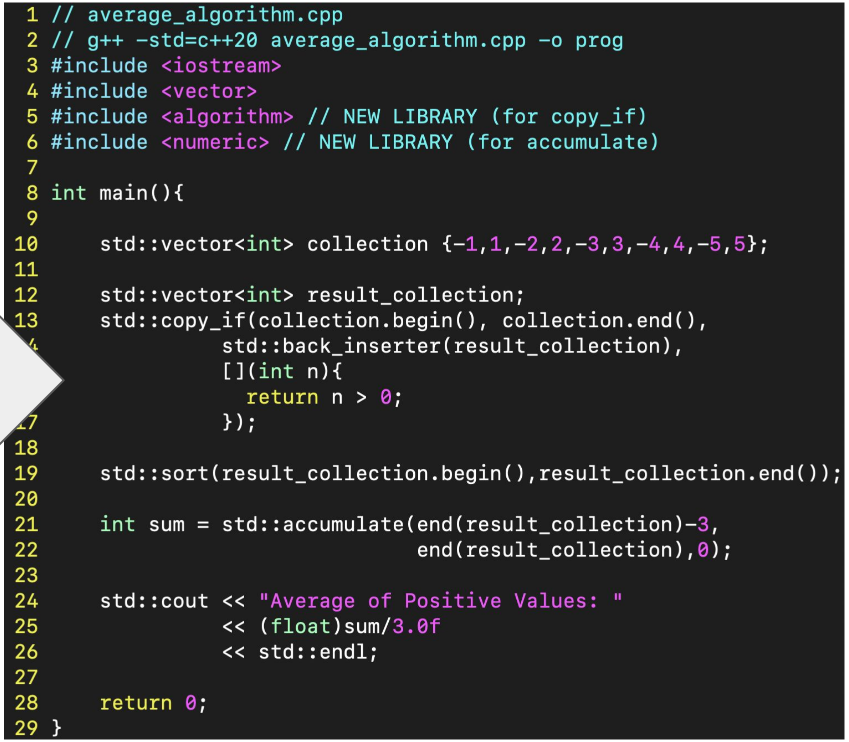


```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });

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

    int sum = std::accumulate(end(result_collection)-3,
                              end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Convince you to change your code from the left side to the right side



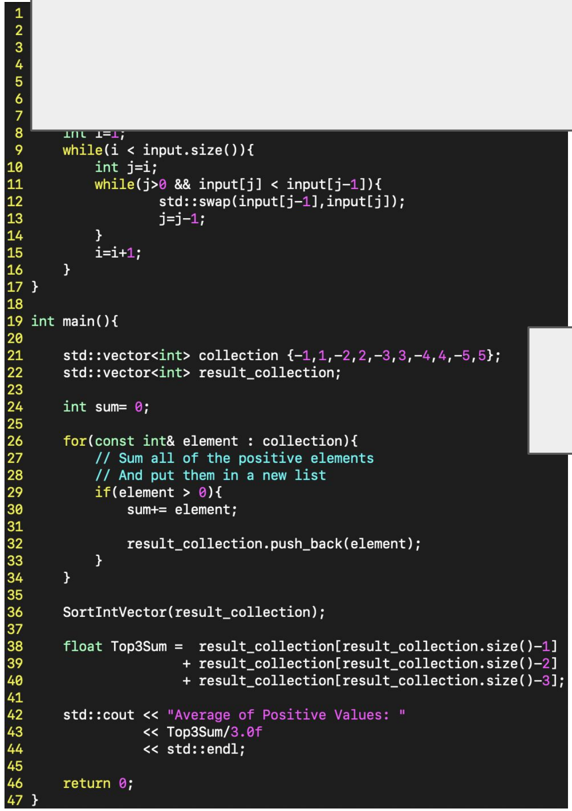
```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```



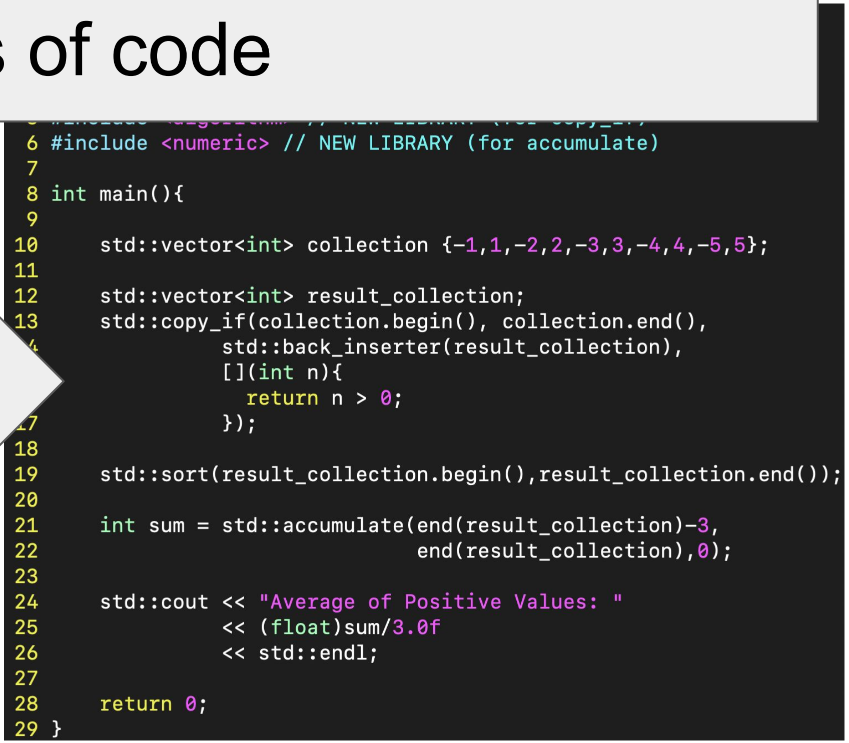
```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
              std::back_inserter(result_collection),
              [](int n){
                return n > 0;
              });

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

    int sum = std::accumulate(end(result_collection)-3,
                              end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Convince you to change your code from the left side to the right side

Fewer lines of code

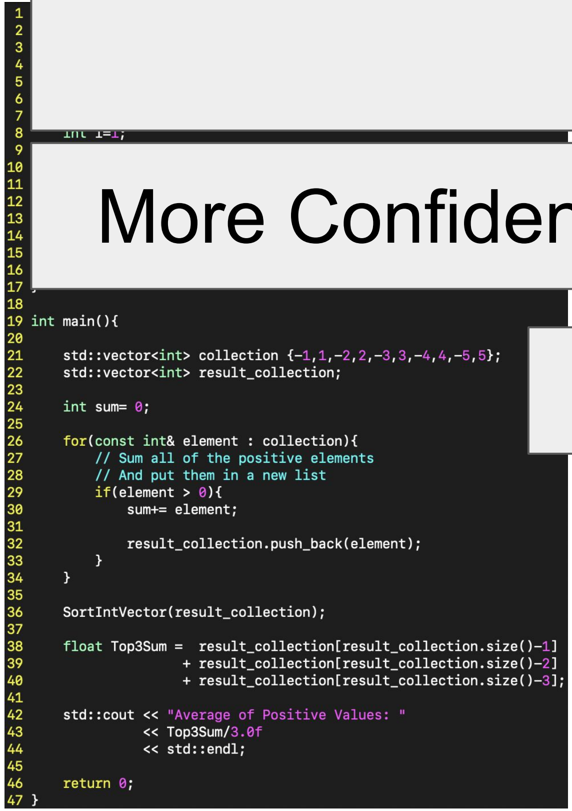


```
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```



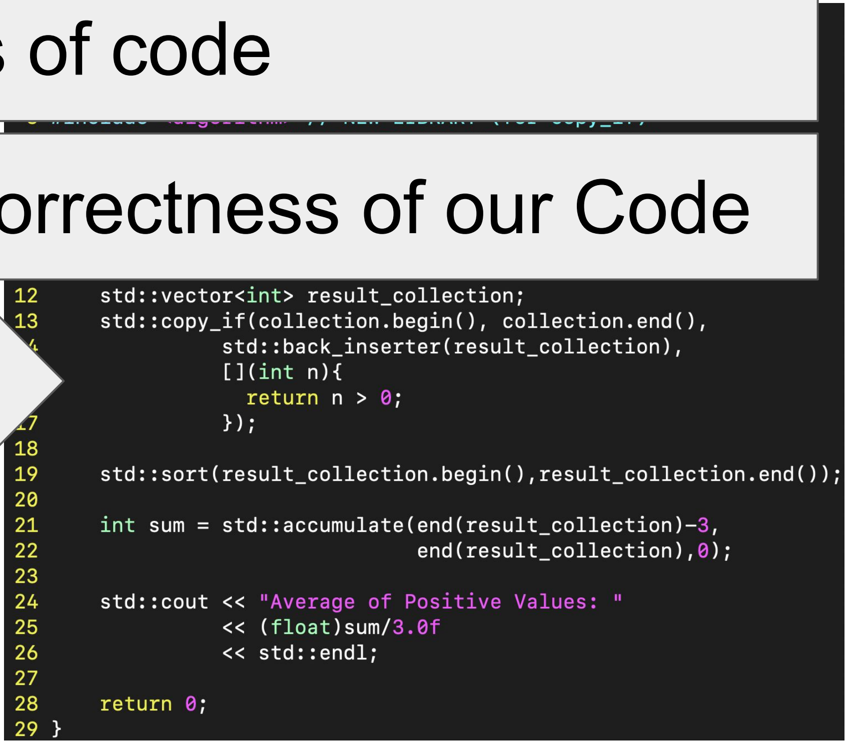
```
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
            std::back_inserter(result_collection),
            [](int n){
              return n > 0;
            });

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

    int sum = std::accumulate(end(result_collection)-3,
                              end(result_collection),0);

    std::cout << "Average of Positive Values: "
            << (float)sum/3.0f
            << std::endl;

    return 0;
}
```

Convince you to change your code from the left side to the right side



```
    int i=1;
```

```
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```



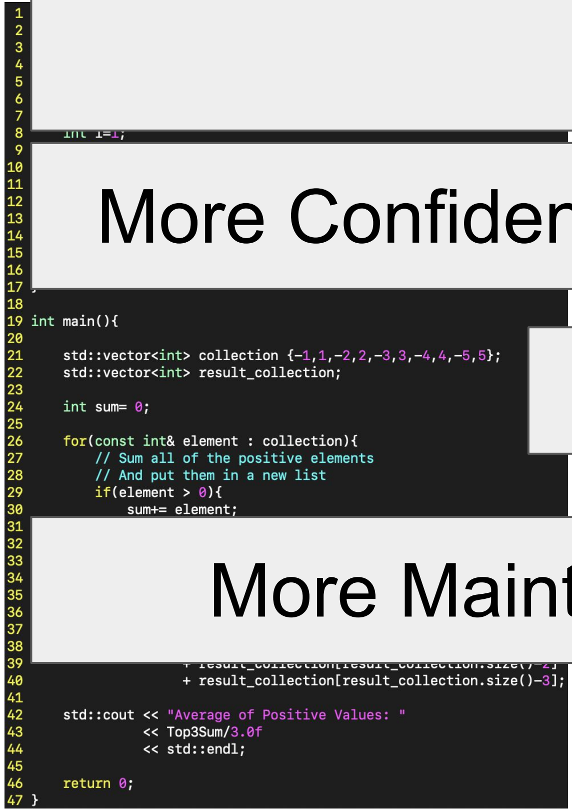
```
#include <algorithm> // NEW LIBRARY (for copy_if)
```

```
    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });

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

    int sum = std::accumulate(end(result_collection)-3,
                              end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Fewer lines of code

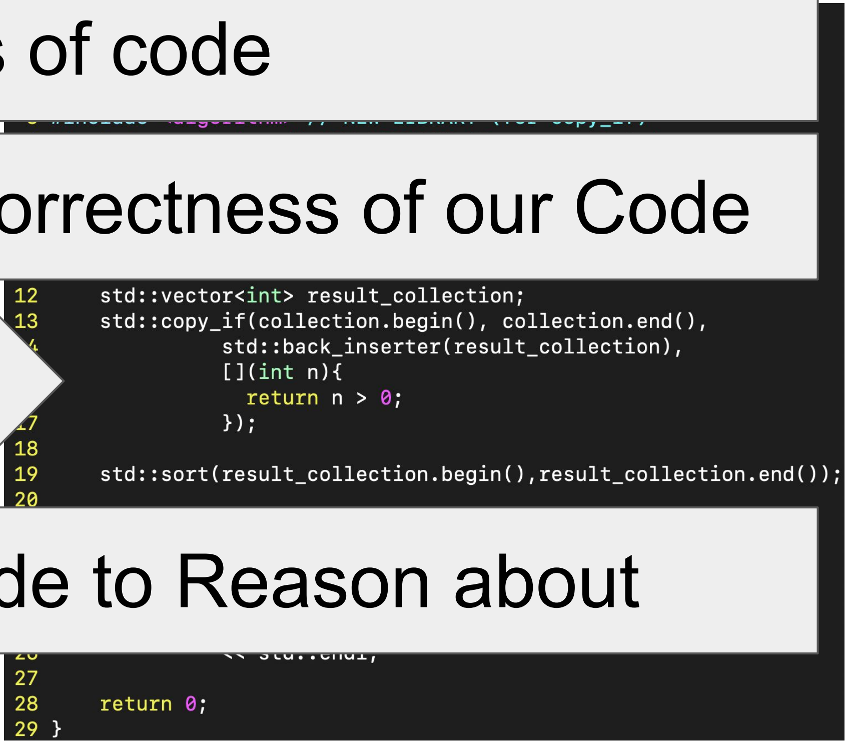
More Confidence in the Correctness of our Code

Convince you to change your code from the left side to the right side



```
int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;
```

```
                  + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```



```
    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });

    std::sort(result_collection.begin(),result_collection.end());
```

```
    return 0;
}
```

Fewer lines of code

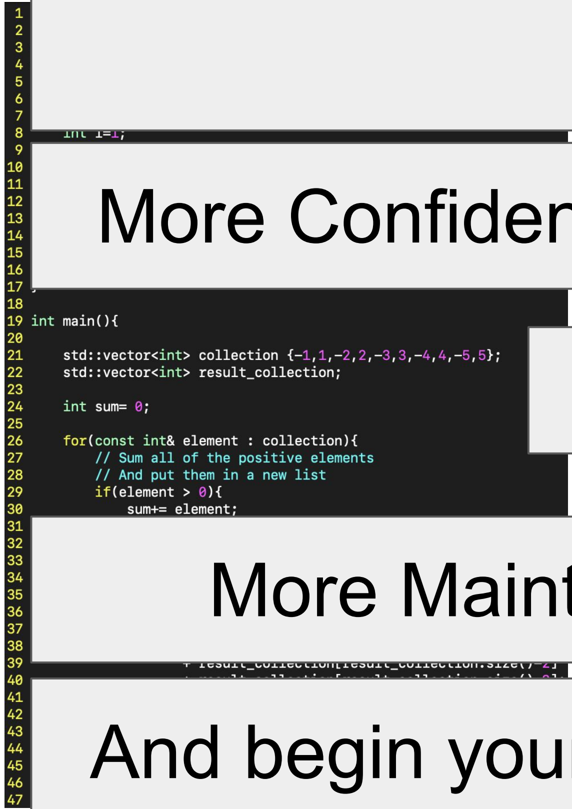
More Confidence in the Correctness of our Code

More Maintainable Code to Reason about

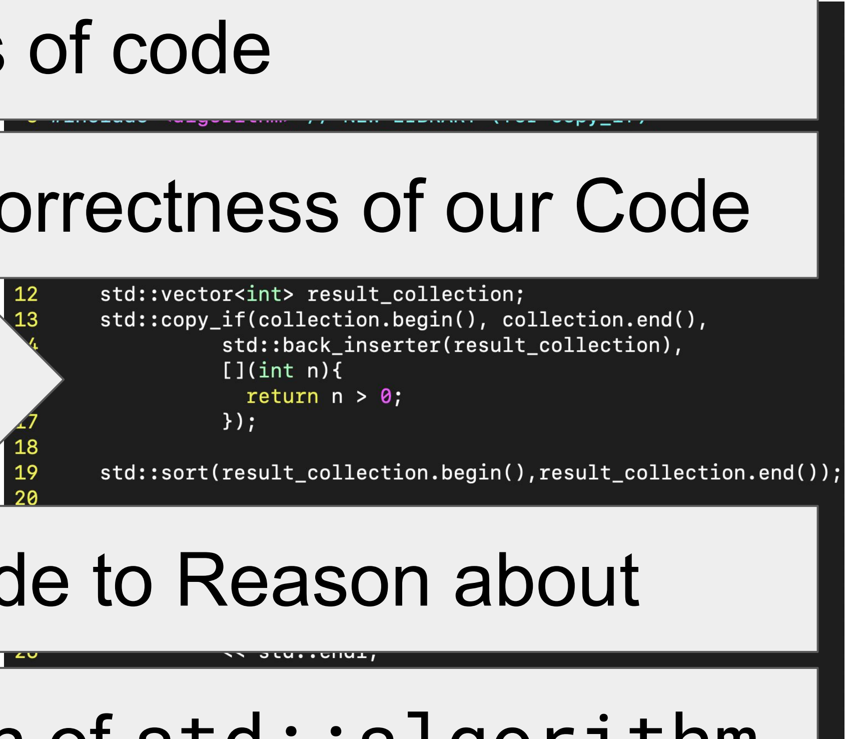
Convince you to change your code from the left side to the right side

Fewer lines of code

More Confidence in the Correctness of our Code



```
int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;
```

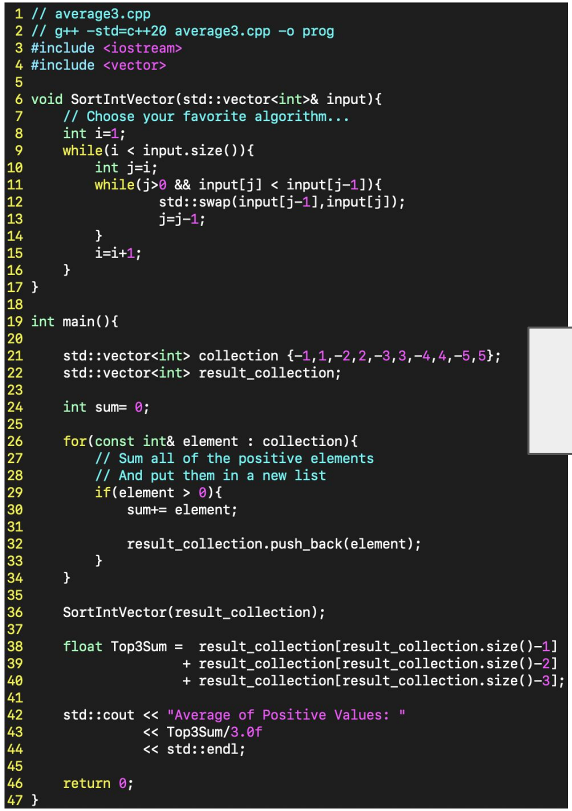


```
    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
            std::back_inserter(result_collection),
            [](int n){
              return n > 0;
            });

    std::sort(result_collection.begin(),result_collection.end());
```

More Maintainable Code to Reason about

And begin your exploration of `std::algorithm`

I will be your tour guide on this journey



```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                  + result_collection[result_collection.size()-2]
                  + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

Folkestone “Zig Zag path”

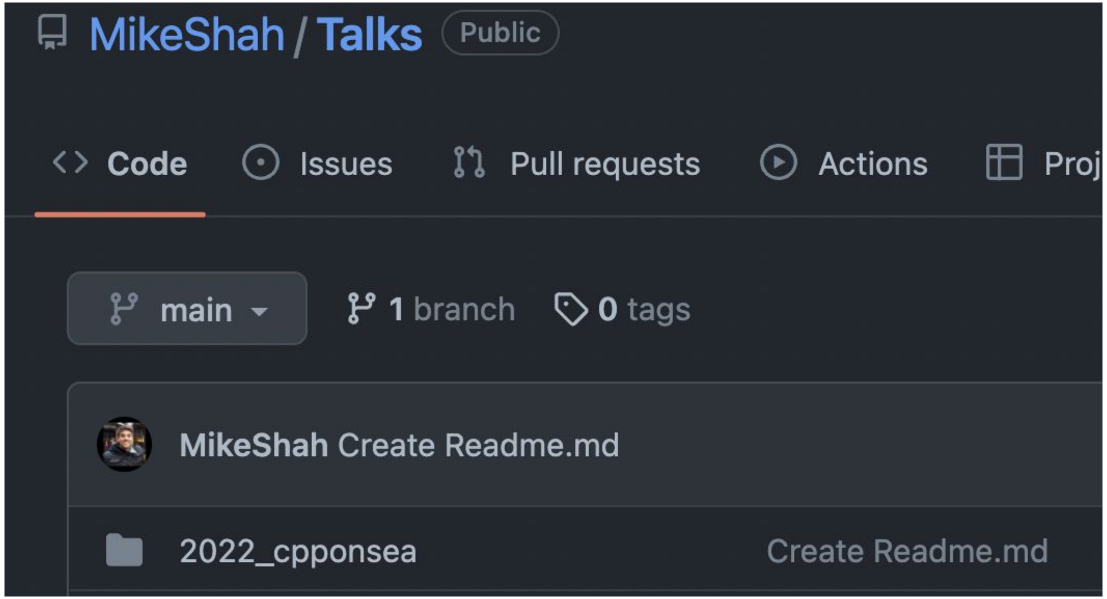
Your Tour Guide for Today

by Mike Shah

- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I teach courses in computer systems, computer graphics, and game engine development.
  - My research in program analysis is related to performance building static/dynamic analysis and software visualization tools.
- I do consulting and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects
  - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of computer graphics, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io
- More online training coming at courses.mshah.io

Code for the talk

- Located here: https://github.com/MikeShah/Talks/tree/main/2022_cpponsea


MikeShah / Talks Public

MikeShah Create Readme.md

2022_cpponsea Create Readme.md

Abstract

The abstract that you read and enticed you to join me is here!

One of the most beautiful parts of the standard library is also a best kept secret to beginning C++ programmers -- `std::algorithm`. Most new C++ programmers do not know the standard algorithms library exists! Often in introductory texts, online tutorials or university courses, `std::algorithm` cannot even be found in the table of contents! In this talk, I would like to provide a proper introduction to `std::algorithm`. I will introduce the library, show how you can rewrite your current code using `std::algorithm`, and also justify why you should be using std::algorithm today and in the future. After this talk, my goal is for beginner C++ programmers to leave excited about uncovering a new paradigm for programming in C++.

What you're going to learn today

- For folks/students newer to C++
  - I'm going to give you an introduction to a library that too often is a secret-- `std::algorithm`
  - I hope it will change the way you consider writing C++ in your respective domain
- For faculty, trainers, and those with more experience
  - I hope to provide you examples to motivate your students/colleagues to use more `std::algorithm` and (generally C++20 and beyond) in your courseware.

Pretend these seats are filled :)
https://pixnio.com/free-images/2017/03/11/2017-03-11-16-47-11-550x413.jpg

A Quick Story

(about my long journey learning C++, and why this talk exists)

https://s0.geograph.org.uk/geophotos/03/09/13/3091367_5ca99096.jpg

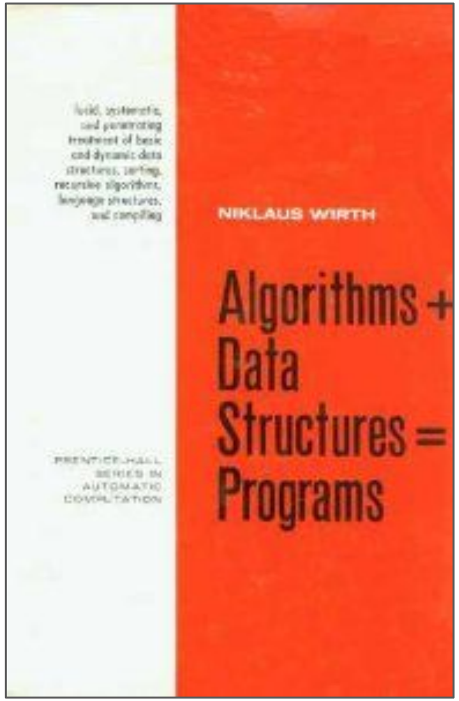
My Very Traditional Journey Learning C++

- I began formally learning a 'custom version' of C++ in university around 2008/2009.
  - Something strange though, was that the course never once mentioned `std::algorithm`!
  - It was very much a 'C with Classes' type of instruction.
- So I really did not know `#include <algorithm>` existed for years!
  - So what is `std::algorithm`?
  - And why do I need it--I have been writing software programs just fine without it for years!
  - (next slide)

https://m.economictimes.com/thumb/msid-76642954,width-1200,height-900,resizemode-4,imgsize-330838/although-a-separate-court-case-established-early-holmes-novels-are-in-the-public-domain-the-lawsuit-alleges-the-detective-only-developed-feelings-in-the-last-10-books-.jpg

Algorithms + Data Structures = Programs

- Niklaus Wirth’s 1976 book’s title captures a good definition of what a software program is.

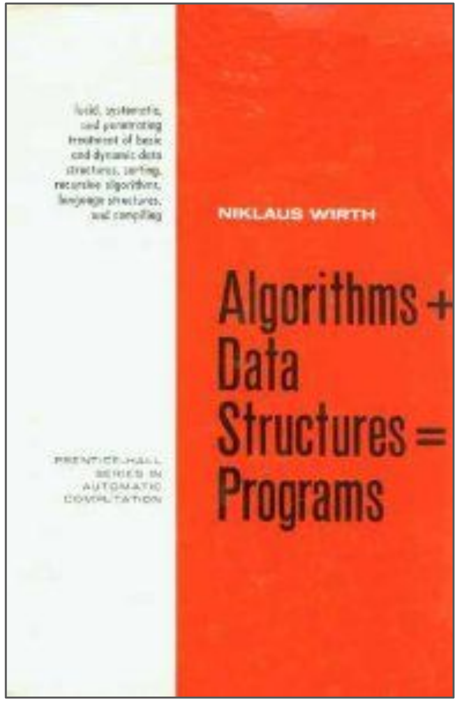

lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling
NIKLAUS WIRTH
Algorithms + Data Structures = Programs
PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

(I own a copy of this wonderful book!)

Algorithms + Data Structures = Programs

- Niklaus Wirth's 1976 book's title captures a good definition of what a software program is.

- Algorithms* -- usually are indicated with at least one loop and a series of function calls
- Data Structures -- usually some sort of container -- like `std::vector` in C++


lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling
NIKLAUS WIRTH
Algorithms + Data Structures = Programs
PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

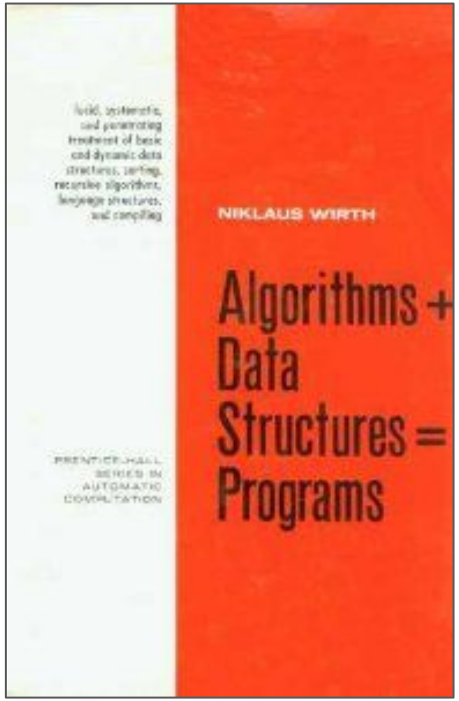
(I own a copy of this wonderful book!)

*Of course -- not all algorithms require or are defined as having a loop.

Algorithms + **Data Structures** = Programs

- Niklaus Wirth's 1976 book's title captures a good definition of what a software program is.

- Algorithms* -- usually are indicated with at least one loop and a series of function calls
- Data Structures -- usually some sort of container -- like `std::vector` in C++



lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

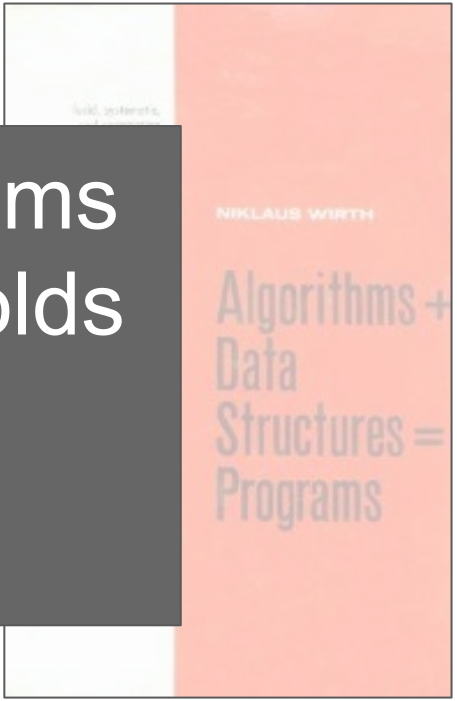
(I own a copy of this wonderful book!)

Algorithms + Data Structures = Programs

- Niklaus Wirth's 1976 book's title captures a good definitio

- Algo
  leas
- Data
  cont

So let's write a few programs and see if this equation holds in C++



(I own a copy of this wonderful book!)

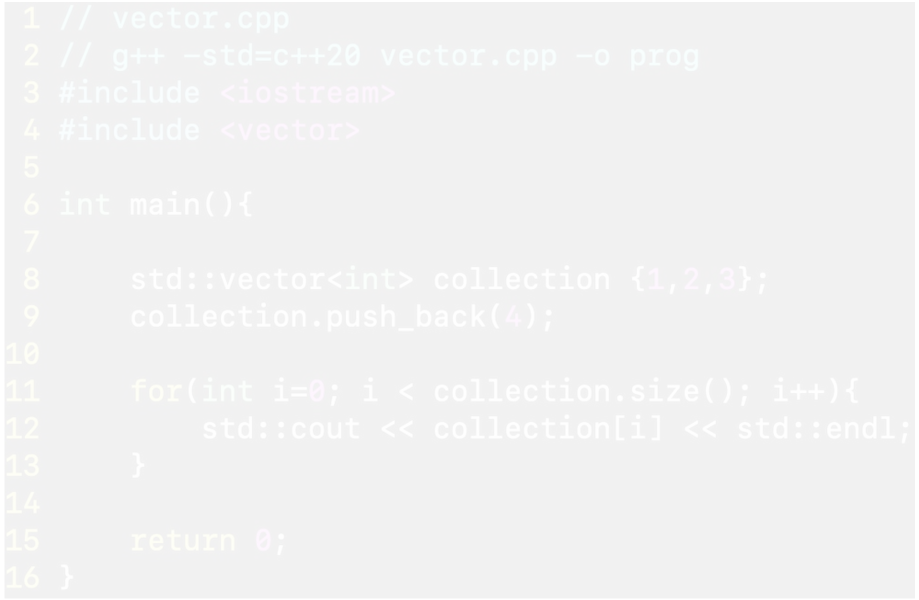
This is Programming - Part 1
`std::vector`

Starting with how ***I* have observed many learn C++ as a beginner**

This is Programming - Part 1
`std::vector`

Vector Part 1

- To the right will be an example program introducing the container `std::vector`
- It reasonably shows that `std::vector` is a built-in *container* available in the Standard Template Library (STL)

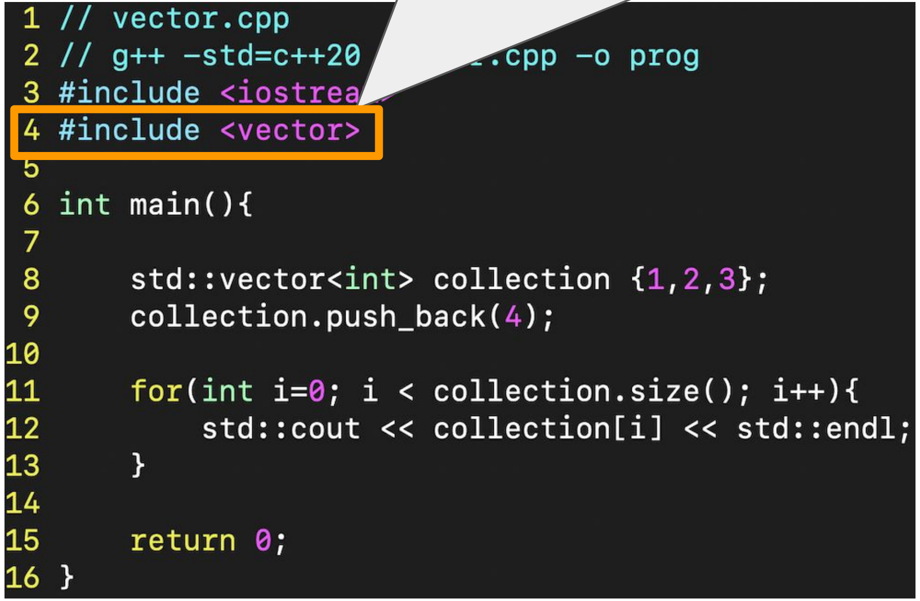


```
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

Vector Part 1

Here’s what could be a new library (STL Container) to a beginner -- `std::vector`

- To the right will be an example program introducing the container `std::vector`
- It reasonably shows that `std::vector` is a built-in *container* available in the Standard Template Library (STL)

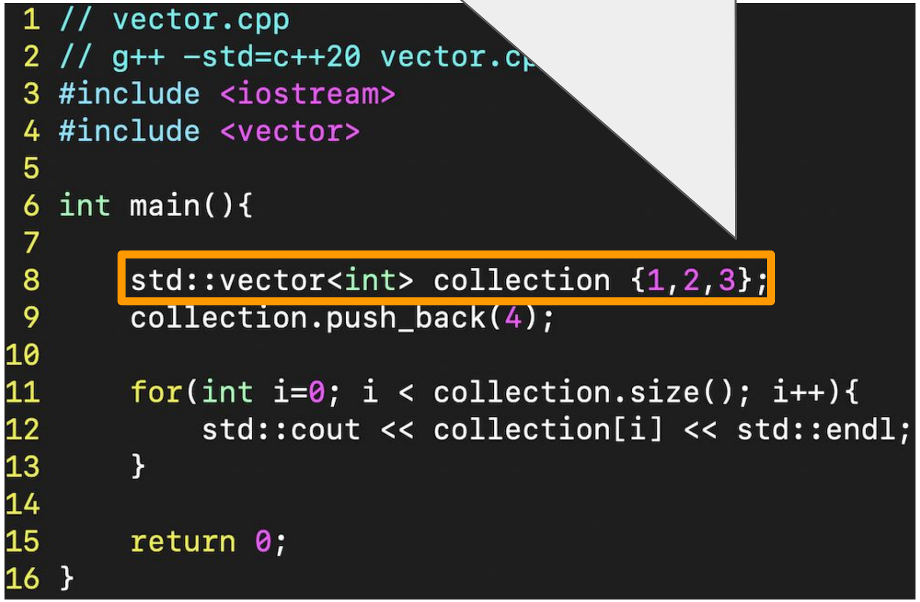


```
// vector.cpp
// g++ -std=c++20 [illegible].cpp -o prog
#include <iostrea
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

Vector Part 1

- Example creating a `std::vector` named ‘collection’
- We use an initializer_list to populate ‘collection’.

- To the right will be an example program introducing the container `std::vector`
- It reasonably shows that `std::vector` is a built-in *container* available in the Standard Template Library (STL)

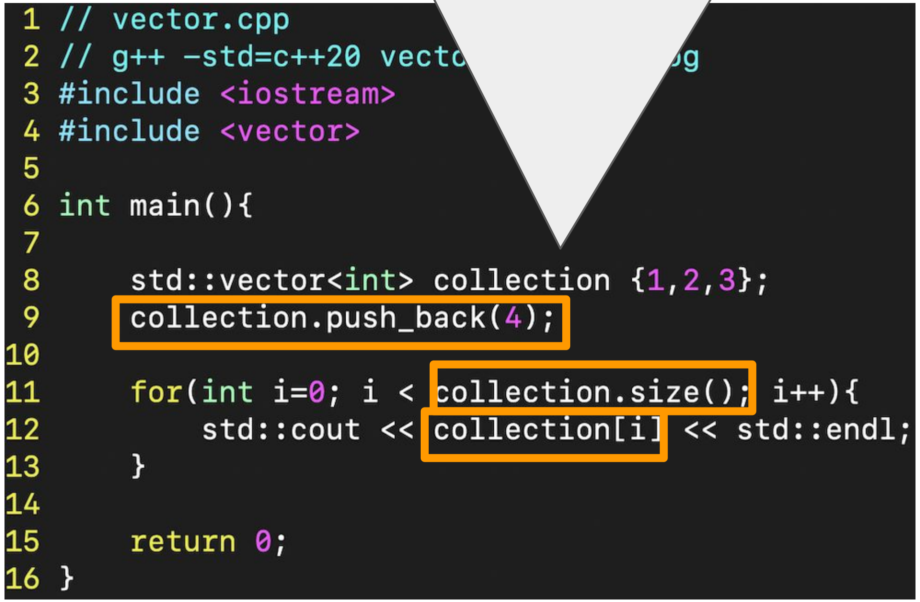


```
// vector.cpp
// g++ -std=c++20 vector.cp
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

Vector Part 1

- And now I introduce a few common member functions that operate on our *collection*: `push_back`, `size()`, and `operator[]`

- To the right will be an example program introducing the container `std::vector`
- It reasonably shows that `std::vector` is a built-in *container* available in the Standard Template Library (STL)

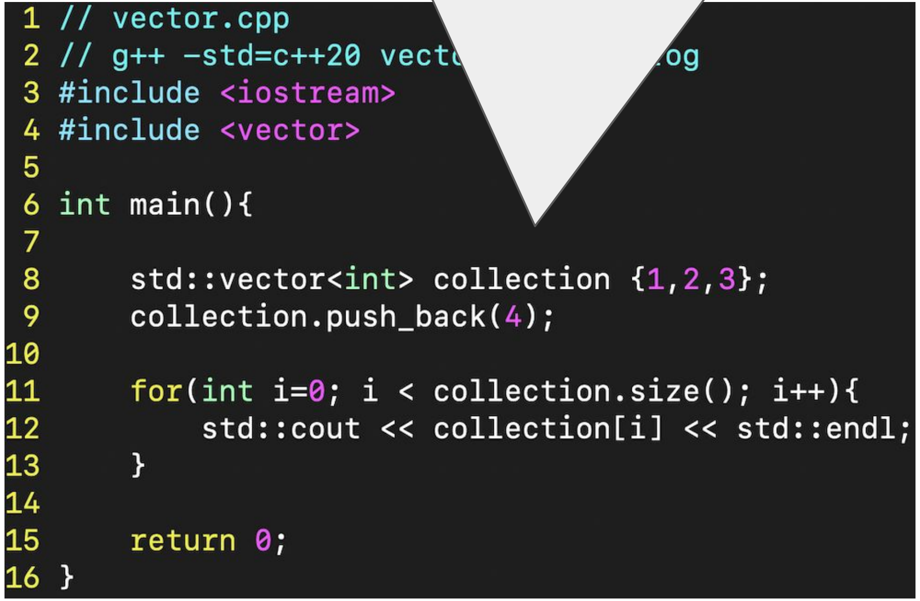


```
// vector.cpp
// g++ -std=c++20 vecto          g
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

Vector Part 1

- And at this point, often we may move onto teaching students another container.

- To the right will be an example program introducing the container `std::vector`
- It reasonably shows that `std::vector` is a built-in *container* available in the Standard Template Library (STL)

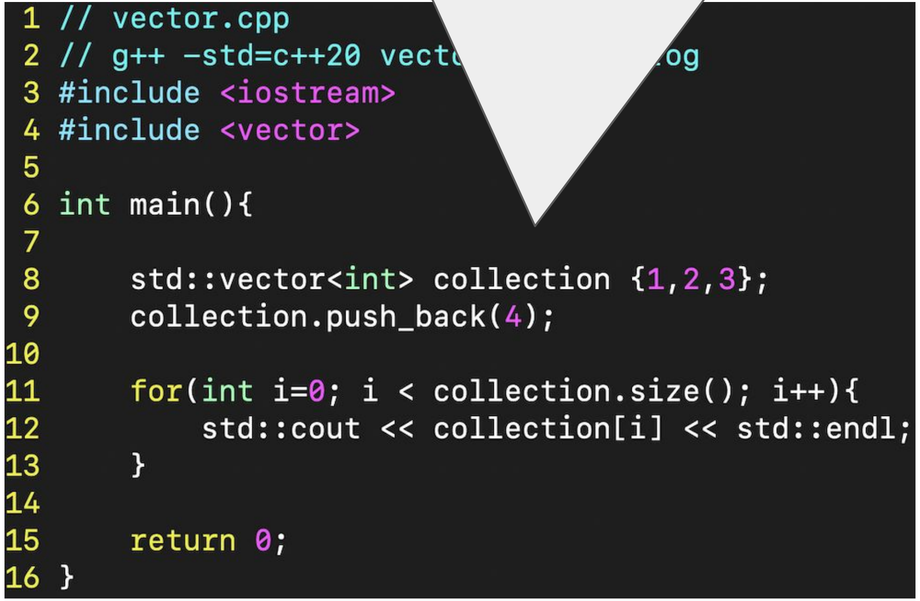


```
// vector.cpp
// g++ -std=c++20 vect                  og
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

- And our equation holds!
    - Algorithms + Data Structures = Programs

Vector Part 1

- To the right will be an example program introducing the container `std::vector`
- It reasonably shows that `std::vector` is a built-in *container* available in the Standard Template Library (STL)



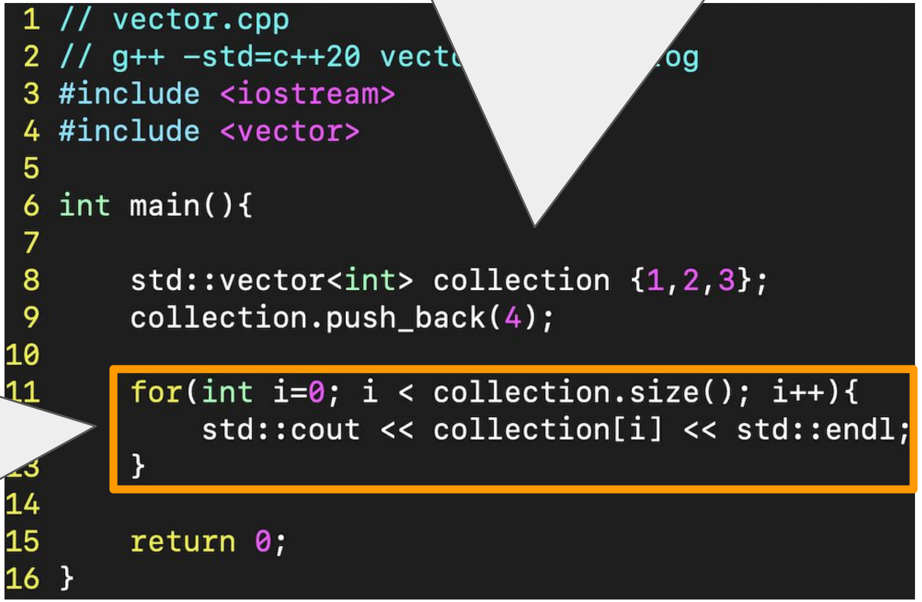
```cpp
// vector.cpp
// g++ -std=c++20 vect                og
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

- And our equation holds!
  - **Algorithms** + Data Structures = Programs

Vector Part 1

- To the right will be an example program introducing the container
-

We have an **algorithm** (A loop that runs '`i`' iterations)



```
// vector.cpp
// g++ -std=c++20 vecto          og
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

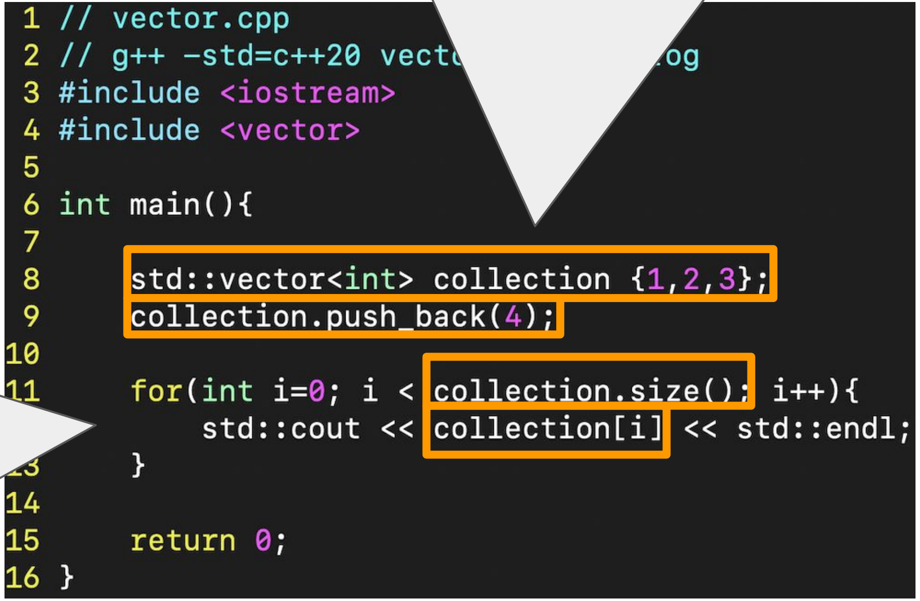
Vector Part 1

- And our equation holds!
  - Algorithms + **Data Structures** = Programs

- To the right will be an example program introducing the container

-

We have a **data structure** to store some computation

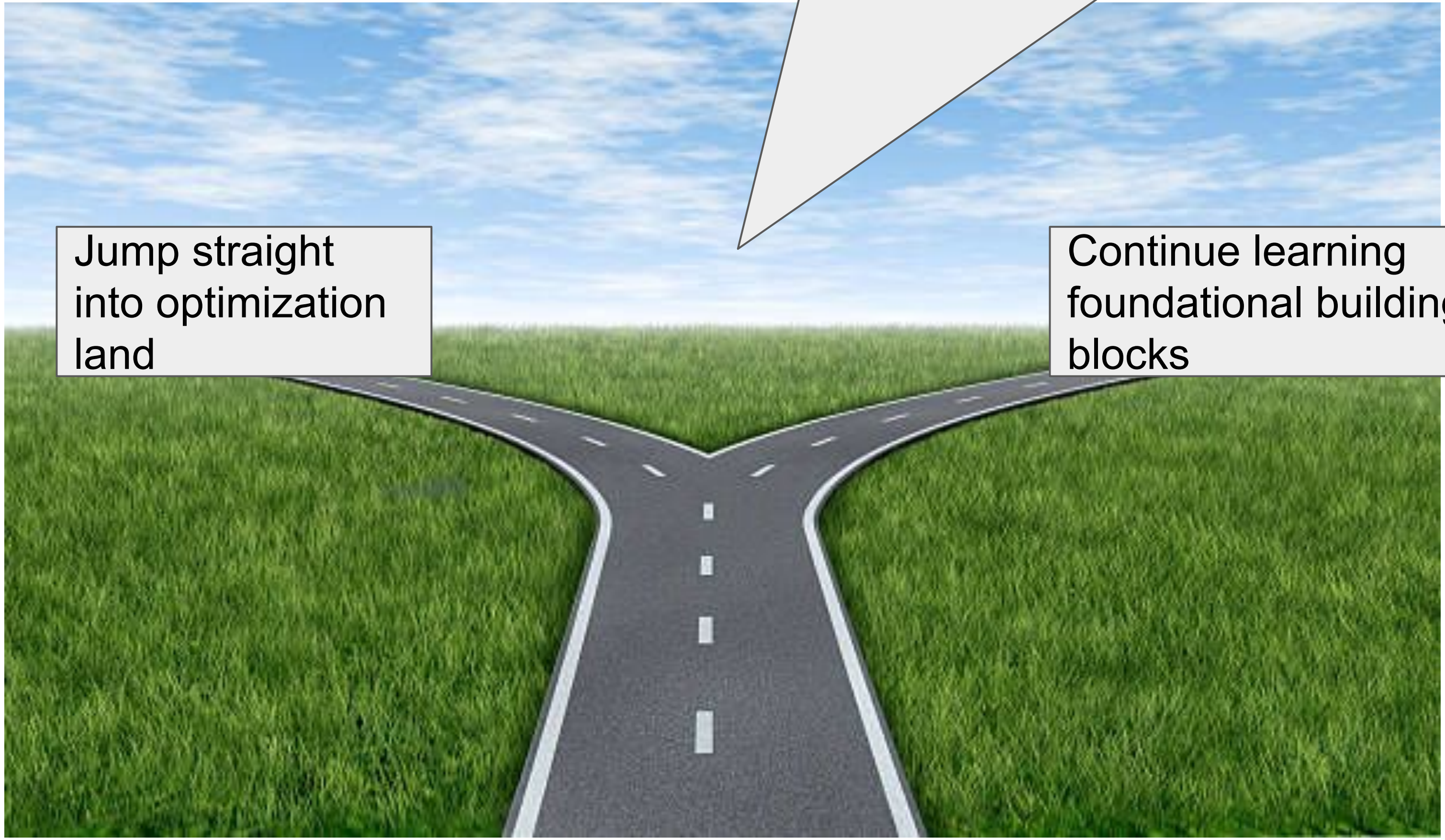


```
// vector.cpp
// g++ -std=c++20 vect              og
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

Next progression -- Write ‘**better C++**’ code

This is Programming - Part 2
`std::vector`

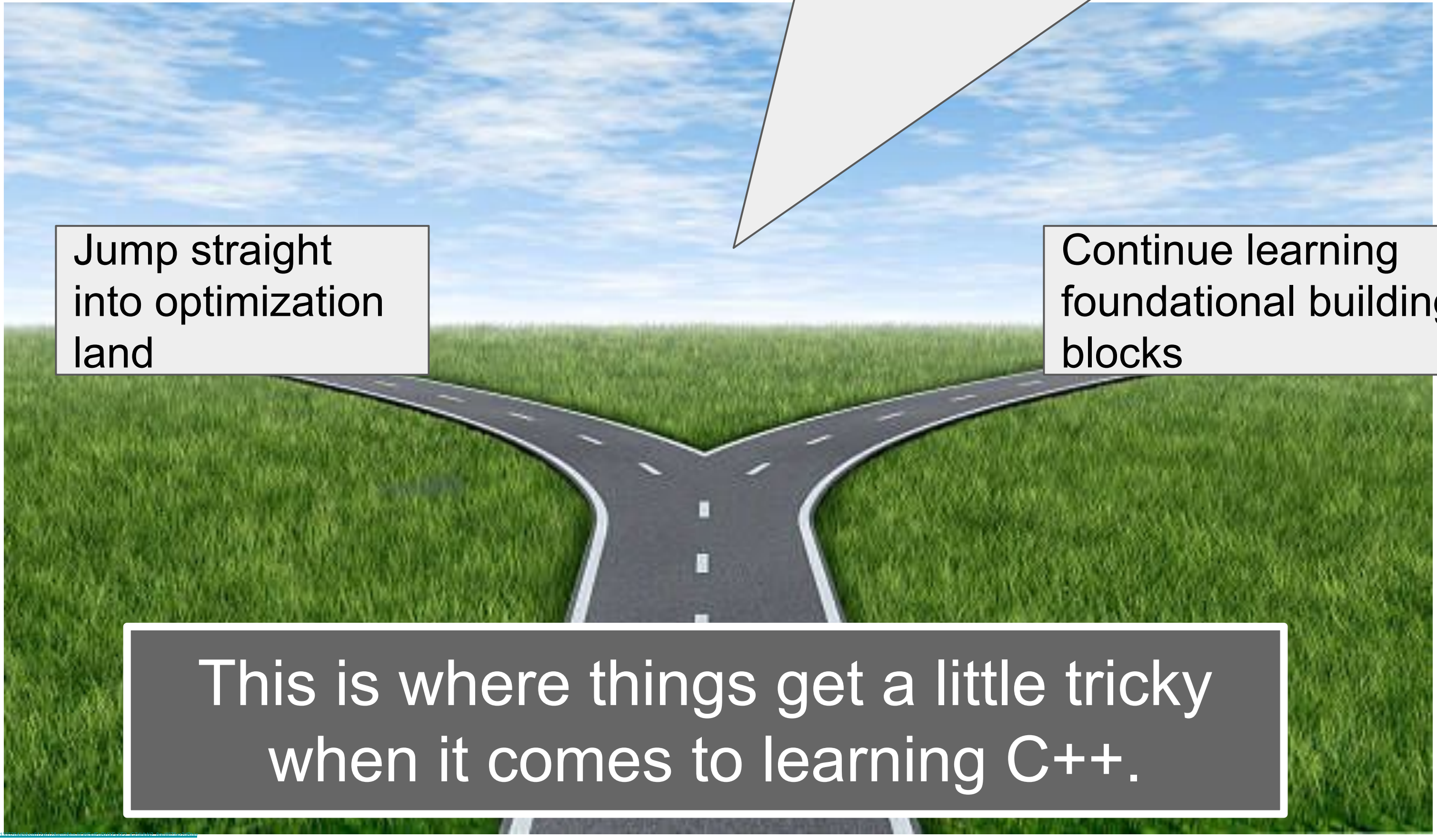
A choice for how to -- write ‘**better C++**’ code



Jump straight into optimization land

Continue learning foundational building blocks

A choice for how to -- write ‘**better C++**’ code



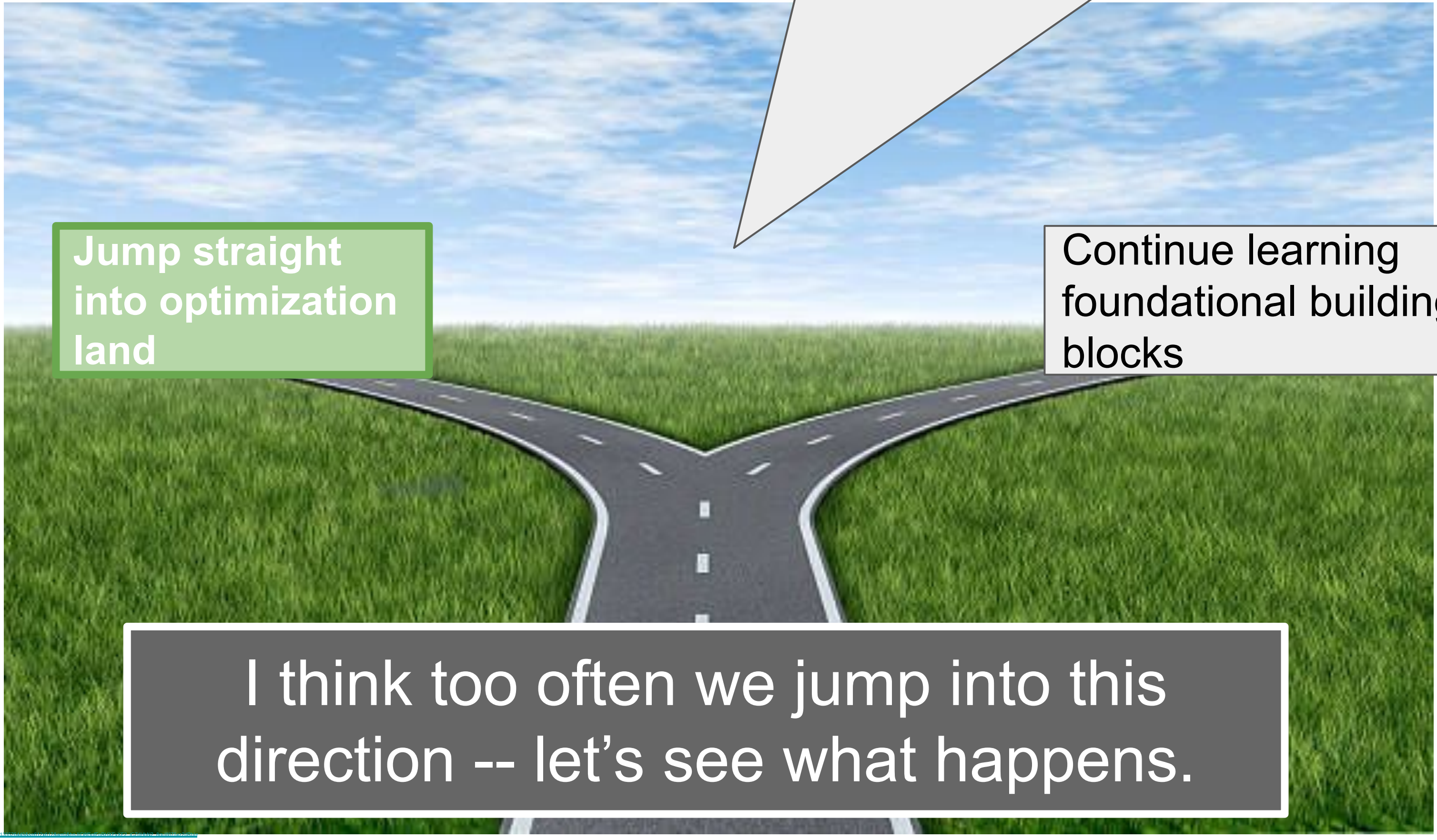
Jump straight into optimization land

Continue learning foundational building blocks

This is where things get a little tricky when it comes to learning C++.

https://media.istockphoto.com/photos/cross-roads-horizon-picture-id1355264492k=20&m=1355264492&s=612x612&w=0&h=[illegible]

A choice for how to -- write ‘**better C++**’ code



Jump straight into optimization land

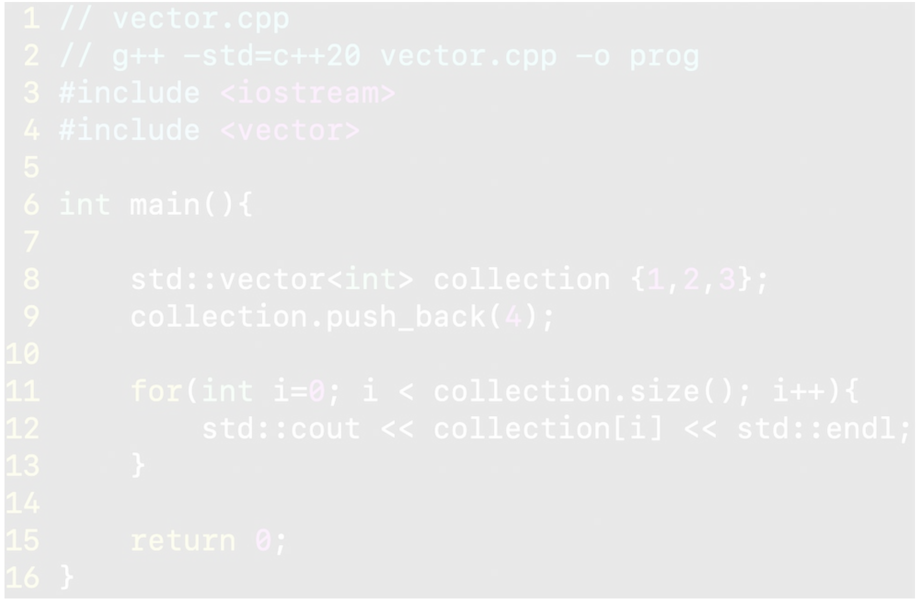
Continue learning foundational building blocks

I think too often we jump into this direction -- let’s see what happens.

https://media.istockphoto.com/photos/cross-roads-horizon-picture-id1355264492?k=20&m=1355264492&s=612x612&w=0&h=[illegible]

Vector Part 2 - A better version

- After introducing students to some small examples, we then want them to write “better C++” code
- “Better code” generally means:
  - More precise
  - More resilient to bugs
  - More performant
  - Easier to maintain

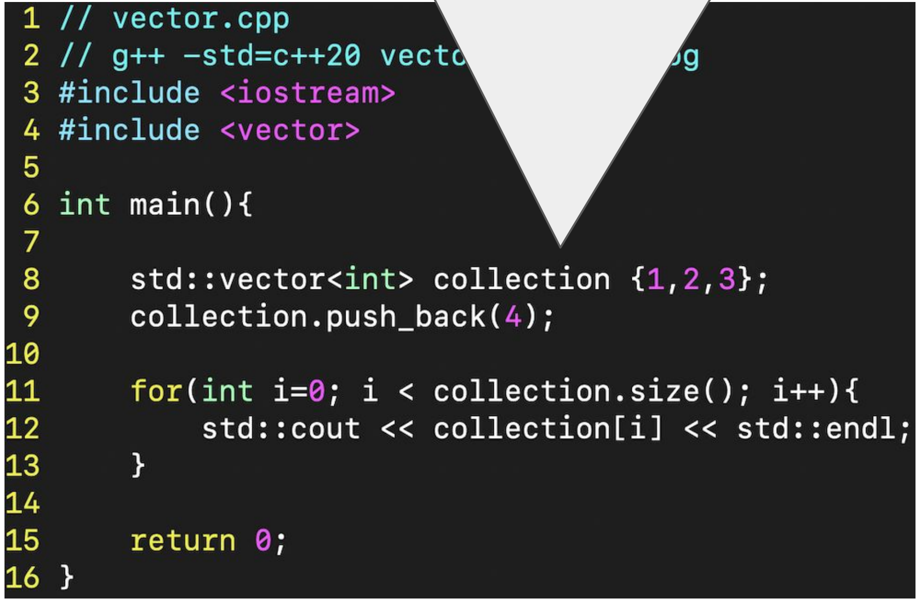


```cpp
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

- Let’s improve this code

Vector Part 2 - A b

- After introducing students to some small examples, we then want them to write “better C++” code
- “Better code” generally means:
  - More precise
  - More resilient to bugs
  - More performant
  - Easier to maintain

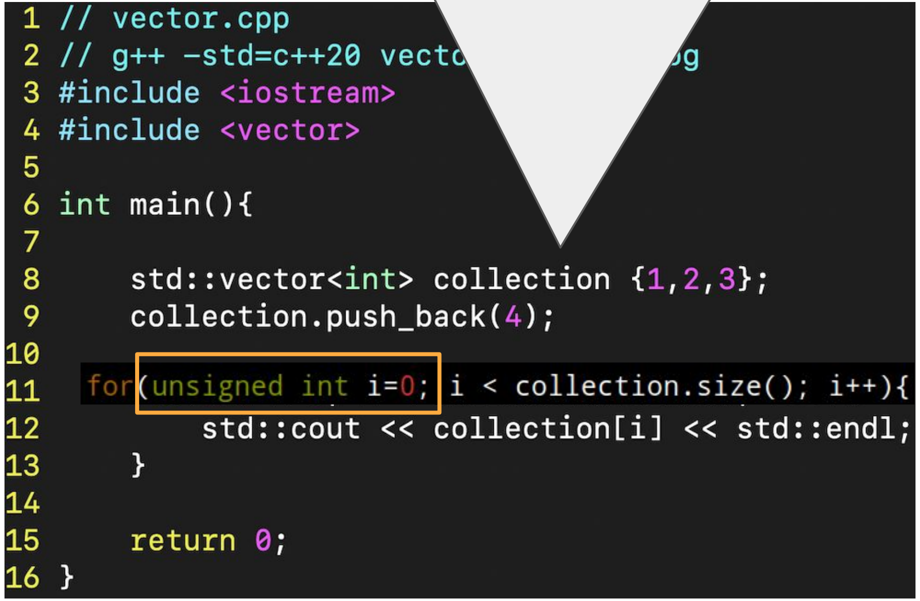


```
// vector.cpp
// g++ -std=c++20 vecto                g
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

Vector Part 2 - A b

- Use an **`unsigned int`** for our index
- (Also ensures consistent `unsigned int` comparison with `collection.size()`)

- After introducing students to some small examples, we then want them to write “better C++” code
- “Better code” generally means:
  - **More precise**
  - More performant
  - More resilient to bugs
  - Easier to maintain

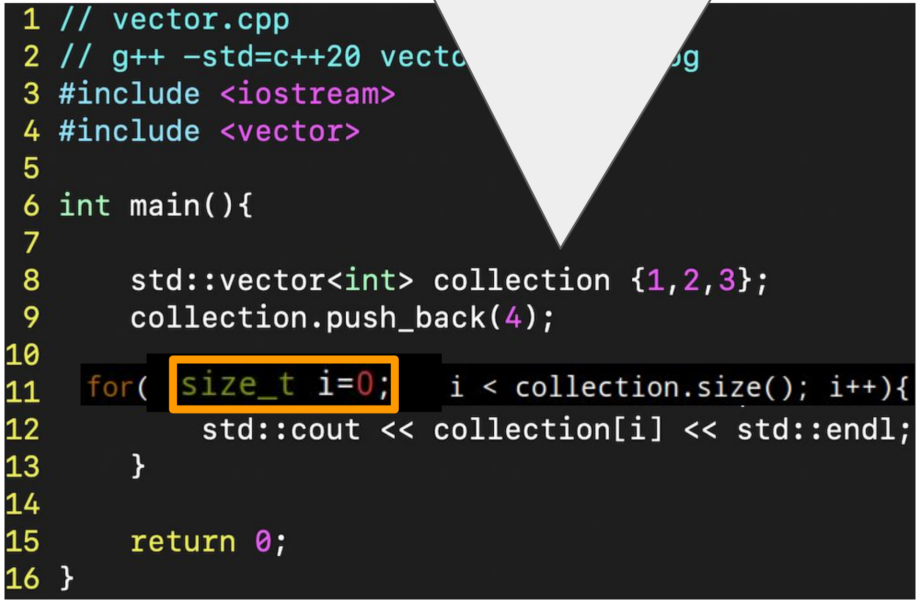


```
// vector.cpp
// g++ -std=c++20 vecto                og
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(unsigned int i=0; i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

- Use `size_t` -- even more precise code!
  - (By convention used for indexing)

Vector Part 2 - A b

- After introducing students to some small examples, we then want them to write “better C++” code
- “Better code” generally means:
  - **More precise**
  - More performant
  - More resilient to bugs
  - Easier to maintain

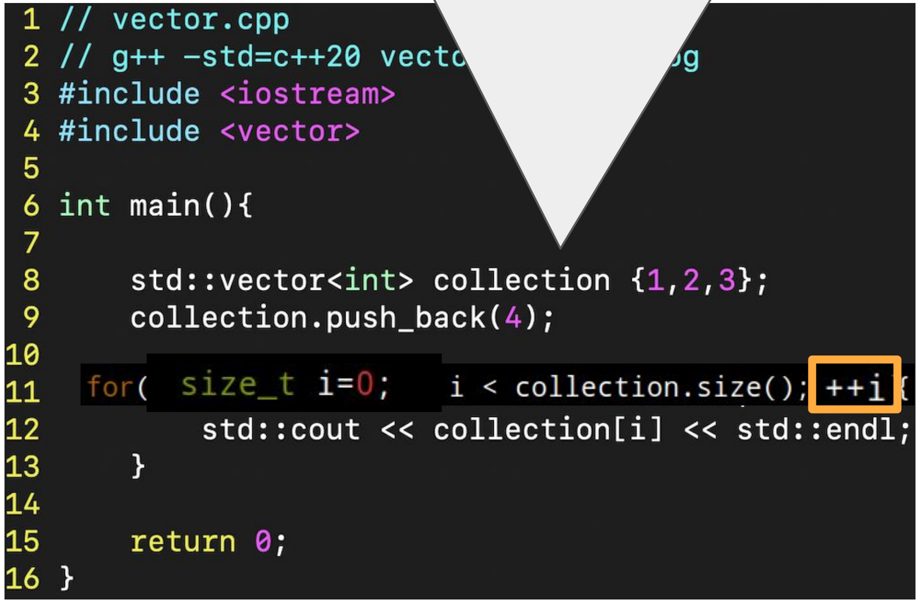


```
// vector.cpp
// g++ -std=c++20 vecto          g
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;    i < collection.size(); i++){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

Vector Part 2 - A b

- pre-increment (++i) because that'll be faster than post-increment (i++)
  - (the compiler *may* fix this for us--but let's do the right thing first)

- After introducing students to some small examples, we then want them to write "better C++" code
- "Better code" generally means:
  - More precise
  - **More performant**
  - More resilient to bugs
  - Easier to maintain

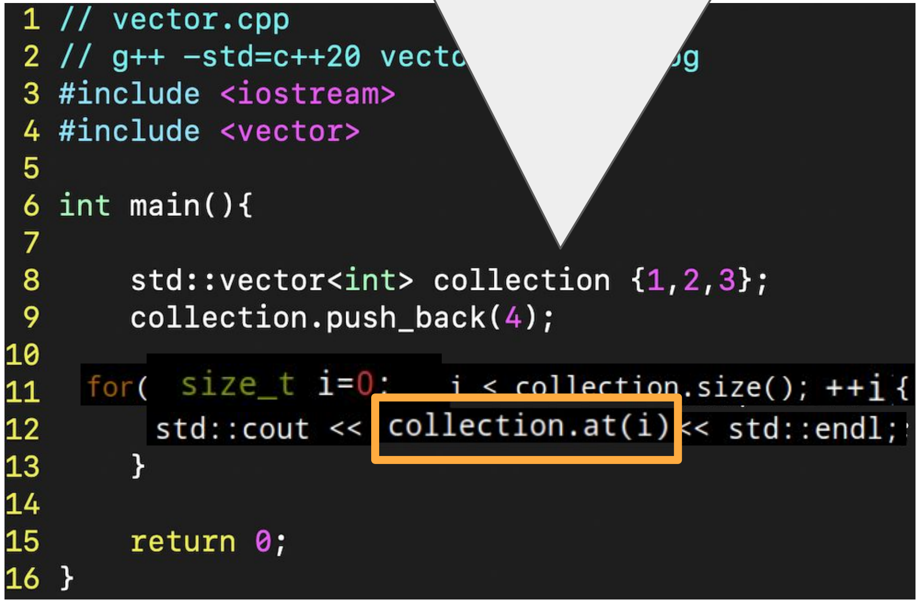


```
// vector.cpp
// g++ -std=c++20 vecto          g
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

  for( size_t i=0;   i < collection.size(); ++i){
        std::cout << collection[i] << std::endl;
    }

    return 0;
}
```

- Use `.at(i)` to do bounds checking -- even more bug resilient

Vector Part 2 - A b

- After introducing students to some small examples, we then want them to write “better C++” code
- “Better code” generally means:
  - More precise
  - More performant
  - **More resilient to bugs**
  - Easier to maintain

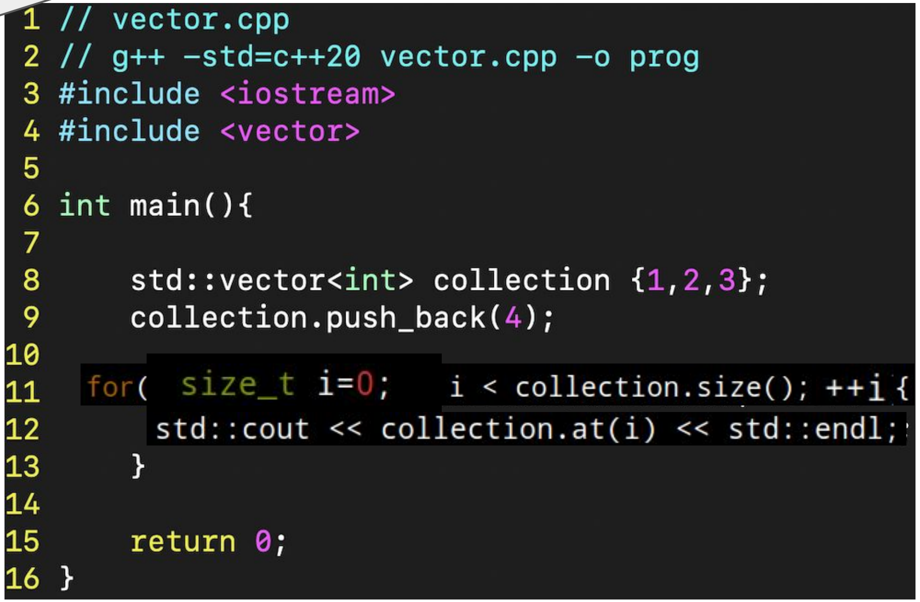


```cpp
// vector.cpp
// g++ -std=c++20 vecto          g
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;   i < collection.size(); ++i {
         std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

Vector Part 2 - Rev

- So here’s a summary of our improvements we have learned to become ‘better programmers’

- More precise
  - ~~Use an unsigned int for our index~~
  - Use size_t because that’s even better!
- More resilient to bugs
  - Use .at(i) to do bounds checking
- More performant
  - pre-increment because we think that’ll be faster (the compiler may fix for us)
- Easier to maintain/reason about
  - ??

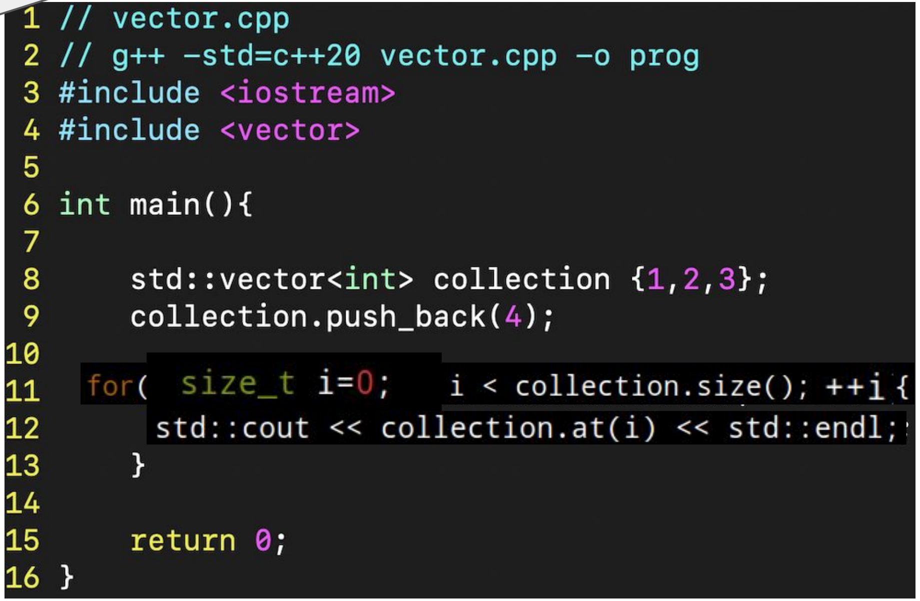


```
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

  for( size_t i=0;   i < collection.size(); ++i {
     std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

Vector Part 2 - Rev

- Let’s stop here, but there are more suggestions....
  - Get rid of the push_back(4) and add to our initializer list...
  - rename idx instead of i
  - `i< collection.size()` => `i != collection.size()-1`

- More precise
  - ~~Use an unsigned int for our index~~
  - Use size_t because that’s even better!
- More resilient to bugs
  - Use .at(i) to do bounds checking
- More performant
  - pre-increment because we think that’ll be faster (the compiler may fix for us)
- Easier to maintain/reason about
  - ??

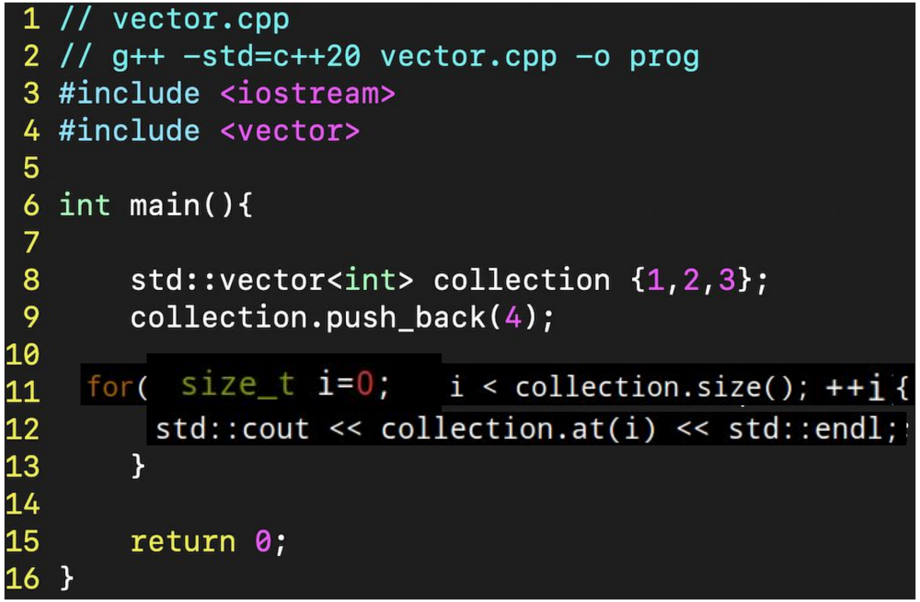


```
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

  for( size_t i=0;   i < collection.size(); ++i {
      std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

- I believe I have improved the code
- Someone would probably accept the new changes in a code review.
- But do you notice something missing?

Vector Part 2

- More precise
  - ~~Use an unsigned int for our index~~
  - Use size_t because that's even better!
- More resilient to bugs
  - Use .at(i) to do bounds checking
- More performant
  - pre-increment because we think that'll be faster (the compiler may fix for us)
- Easier to maintain/reason about
  - ??

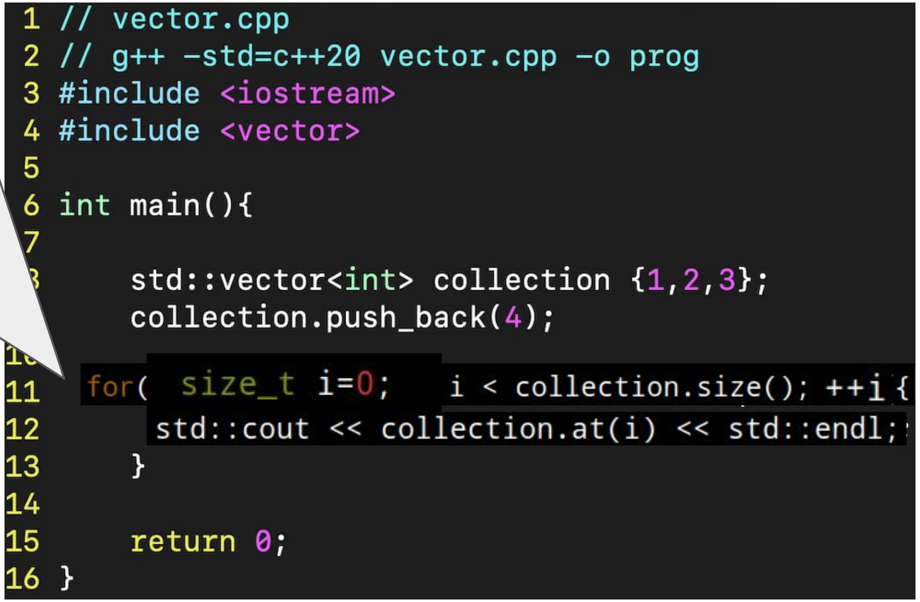


```
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;   i < collection.size(); ++i {
        std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

- Ooops! I didn't really improve the maintainability at all here
  - (maybe `.at(i)` counts?).
  - Maybe I could write some comments at the least?
- But I really have not taught anyone how to think or reason about their code.
- As a systems-y programmer I like the little details we added--but I believe our journey ends here.

be faster (the compile may fix for us)

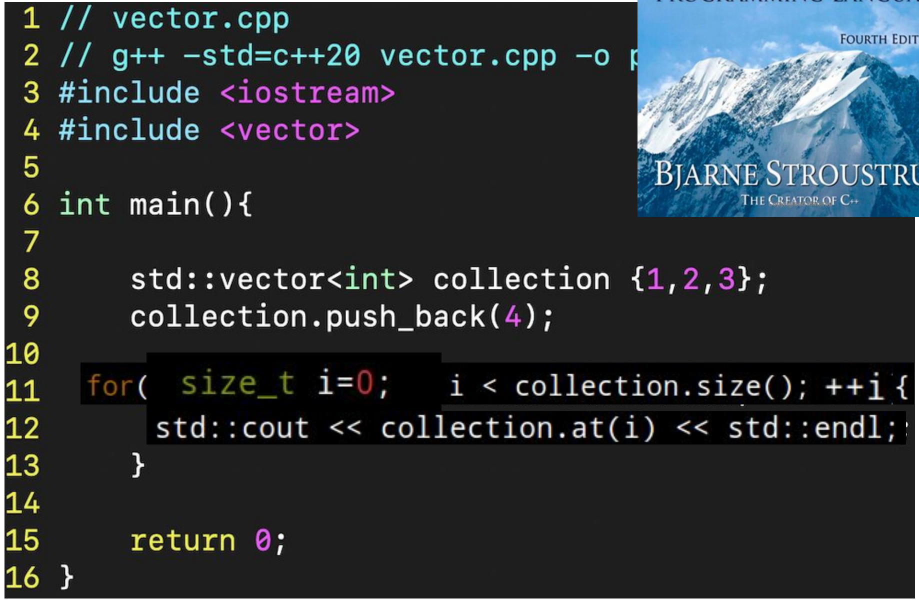
- **Easier to maintain/reason about**
  - **??**



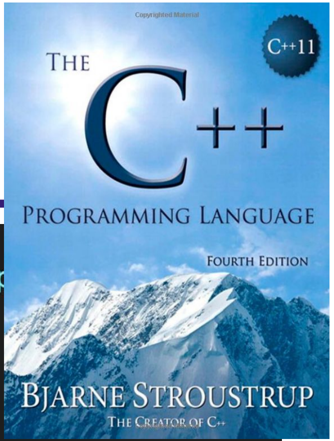
```
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

  for( size_t i=0;   i < collection.size(); ++i {
      std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

loops + classes == Programs?

- So I *could* stop the talk here
- I have written some C++ and incrementally improved a valid program.
    - The ‘loop’ is our *algorithm* describing how we sequentially do something a set number of times
    - The ‘class’ (std::vector) is our *data structure* to store information
- This ‘program’ achieves its job, and we have written software that works.



```cpp
// vector.cpp
// g++ -std=c++20 vector.cpp -o p
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;    i < collection.size(); ++i {
        std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```



Thank you!
The end! Lunch time!

2022

Beginners Guide to C++'s Best Kept Secret

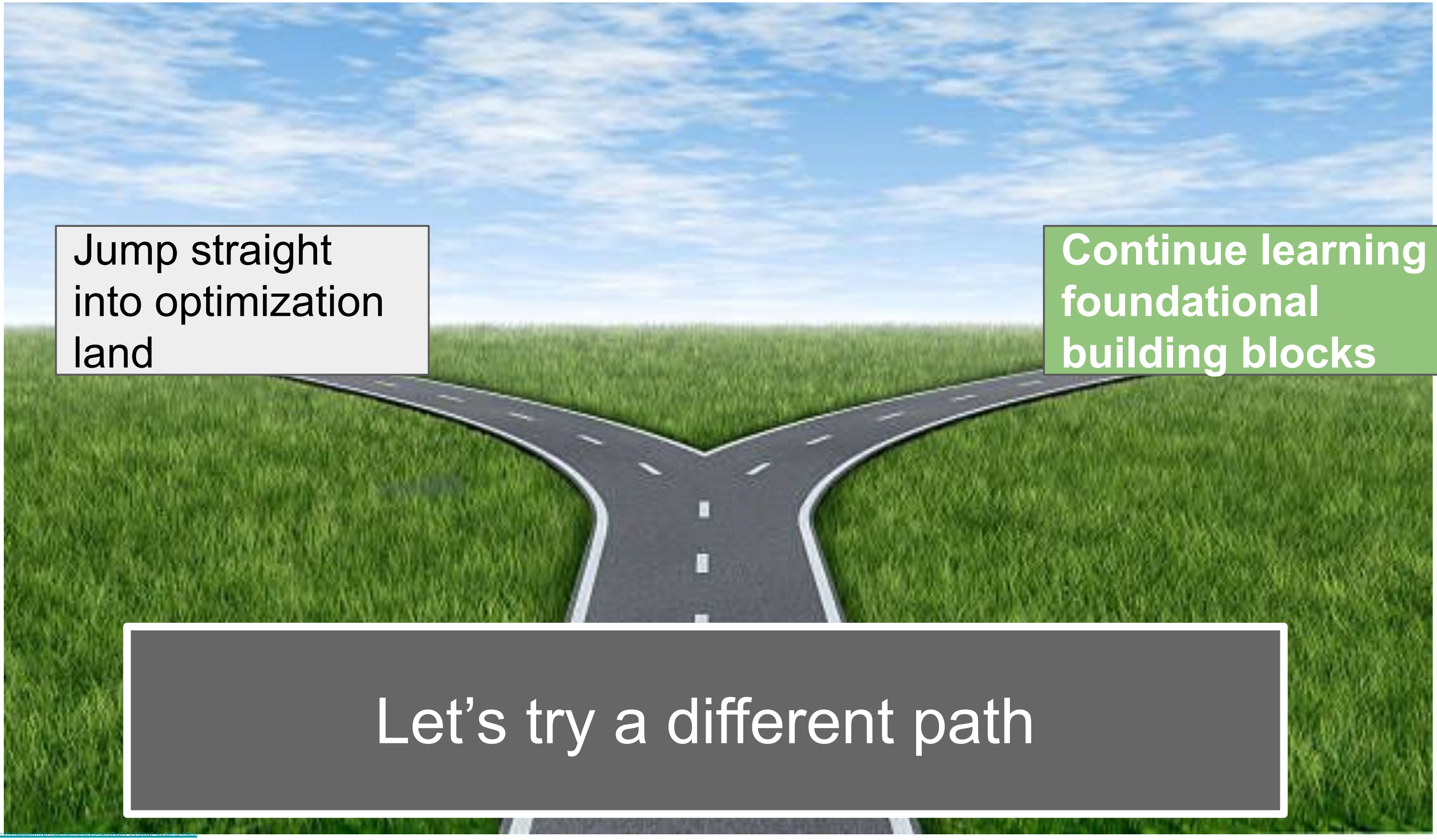
std::algorithm

Mike Shah

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube:
www.youtube.com/c/MikeShah

11:00-12:00, Wed, 6th July 2022

60 minutes | Introductory Audience



Jump straight into optimization land

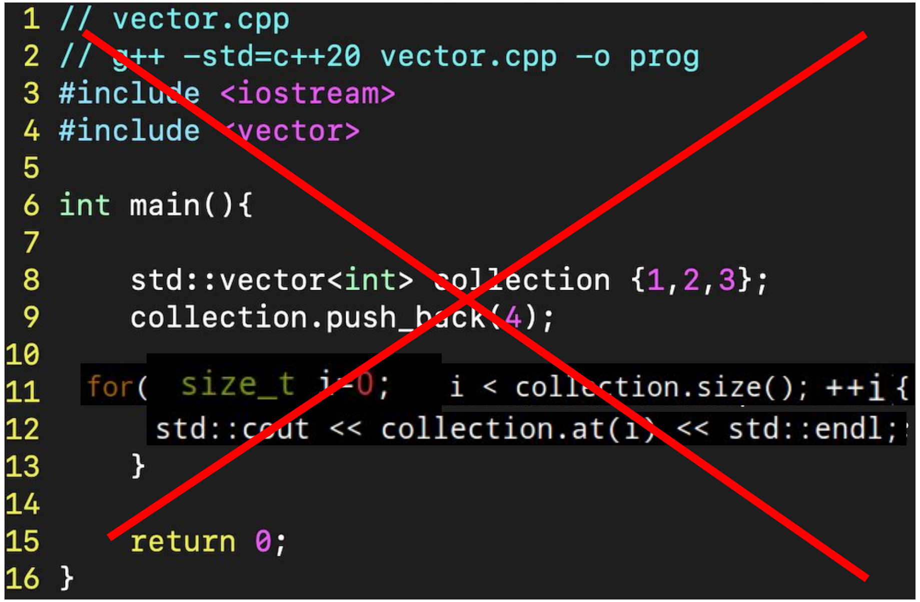
Continue learning foundational building blocks

Let’s try a different path

https://media.istockphoto.com/photos/cross-roads-horizon-picture-id1355264492?k=20&m=1355264492&s=612x612&w=0&h=[illegible]

Focusing on how to make our code **Easier to maintain and reason about**

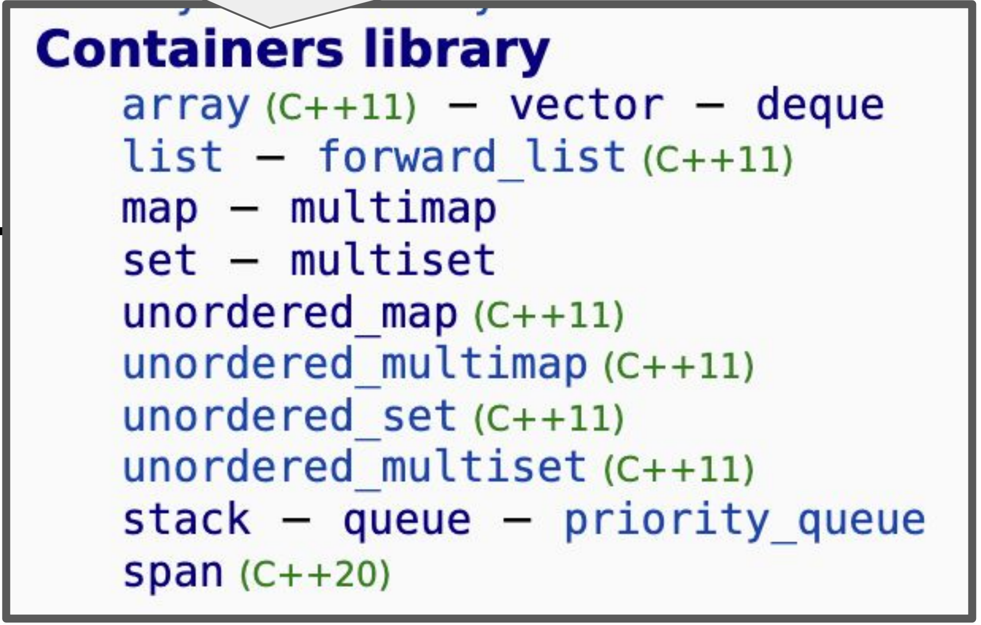
This is Programming - Part 3 std::vector



```
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;  i < collection.size(); ++i {
        std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

We do a good job as teachers teaching and students learning this part

This is Programming - Part 3 `std::vector`



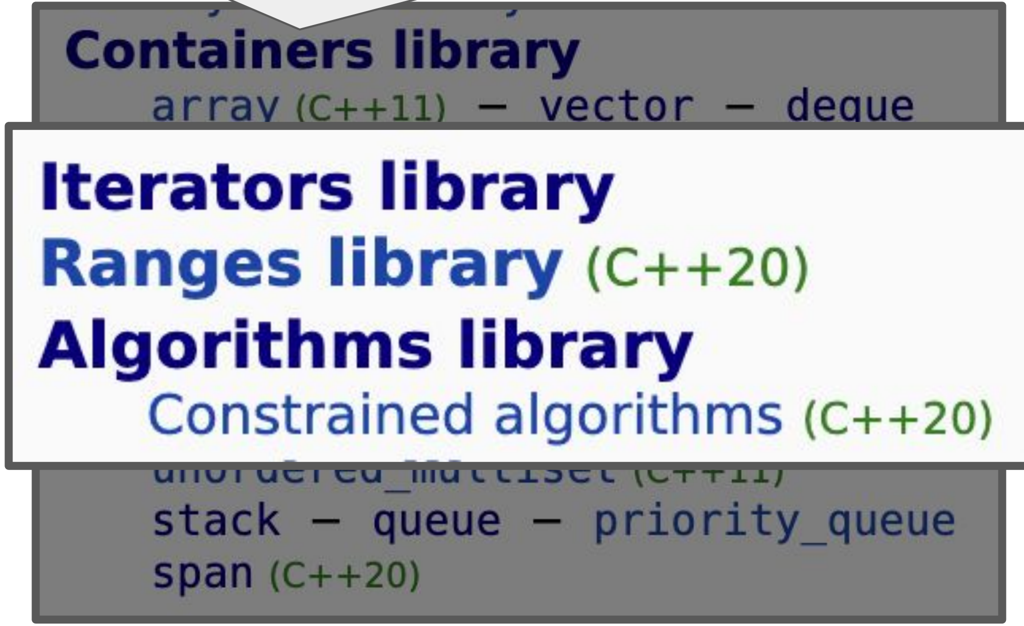
Containers library

array (C++11) – vector – deque
list – forward_list (C++11)
map – multimap
set – multiset
unordered_map (C++11)
unordered_multimap (C++11)
unordered_set (C++11)
unordered_multiset (C++11)
stack – queue – priority_queue
span (C++20)

[Containers cppref]

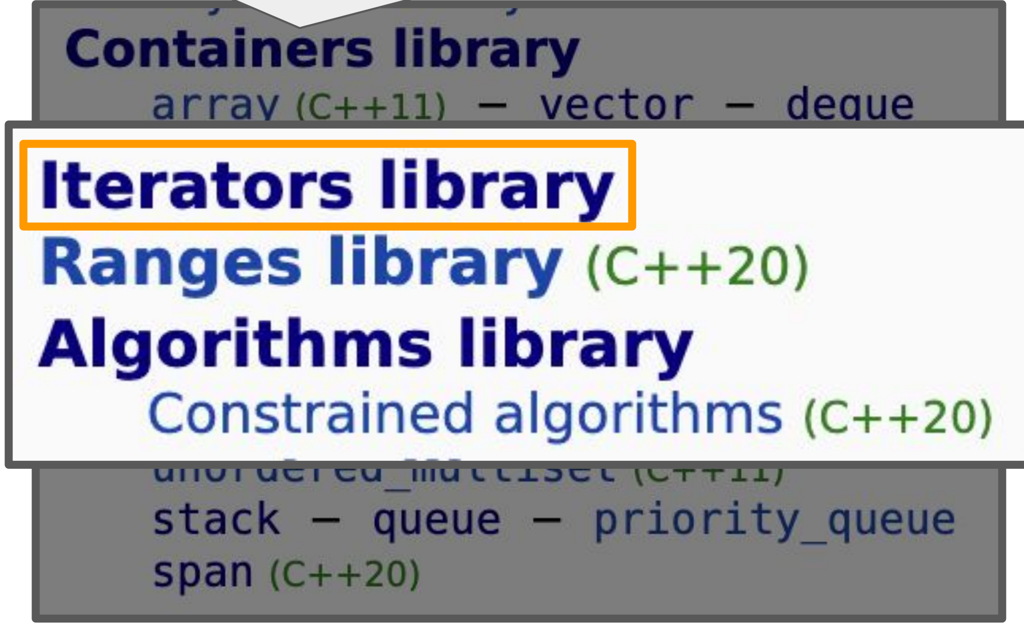
But we need to spend more time emphasizing this part on our journey to write better and more maintainable code.

This is Programming Part 3 `std::vector`


Containers library
array (C++11) – vector – deque
Iterators library
Ranges library (C++20)
Algorithms library
Constrained algorithms (C++20)
stack – queue – priority_queue
span (C++20)

Let's start here

This is Programming Part 3 `std::vector`



Containers library

array (C++11) — vector — deque

Iterators library

Ranges library (C++20)

Algorithms library

Constrained algorithms (C++20)

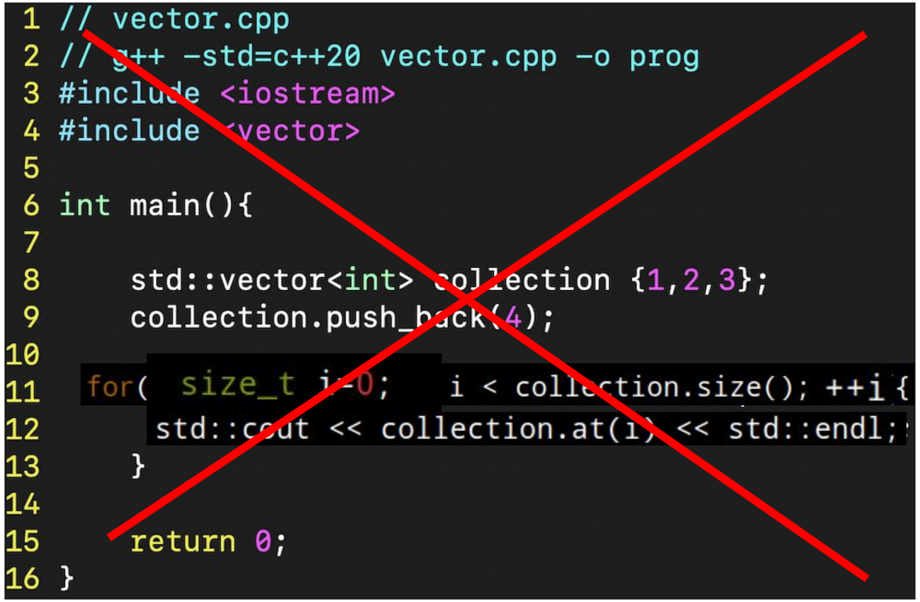
unordered_multiset (C++11)

stack — queue — priority_queue

span (C++20)

Vector Part 3 - Introducing Iterators

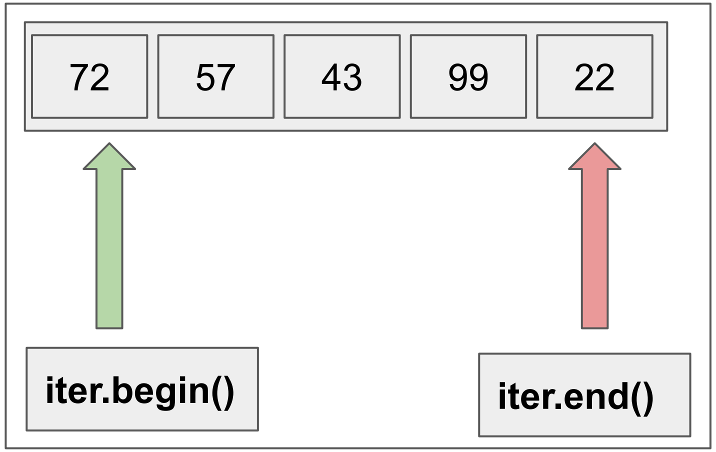
- So the new question is--**how can I make it more maintainable and easy to reason about?**
    - We need some building blocks
- We have actually missed a major part of the Standard Template Library (STL) that is helpful!
    - **Iterators**
    - (Next slide)



```
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;    i < collection.size(); ++i){
        std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

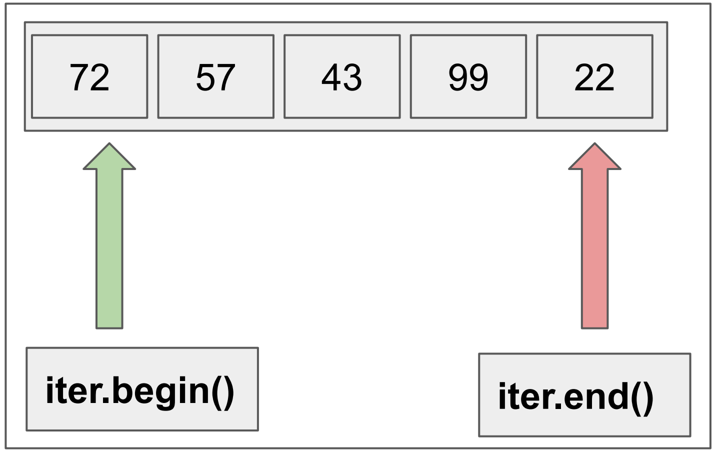
Iterators in C++

- An iterator in C++ is an object that allows you to access elements in a collection.
  - e.g. An iterator may be a pointer to a specific element.
    - `begin()` and `end()` give us convenient access to the beginning or end of a collection
    - (Note: It's probably more interesting for maps or graph data structures--but let's stay with std::vector)
  - This iterator can advance to the next element in some manner
    - Often advancing forward sequentially
    - (could be backwards, or perhaps random access [more on iterator library] as well)



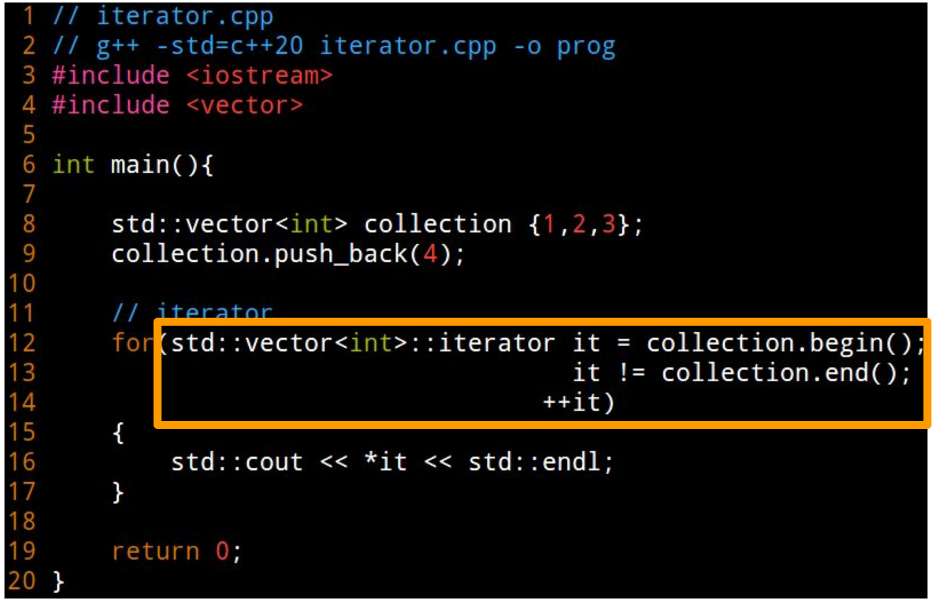
Iterators in C++

- We use a pair of iterators to move through the beginning and end of a collection and perform some computation.
  - We call the pair of iterators a **range.**
    - A 'range' is a pair of iterators designating the beginning and end of our computation.
    - [begin, end)
      - ^ Note the interval notation
- The **range** is where we want to do our algorithmic work



Vector Part 3 - Using Iterators

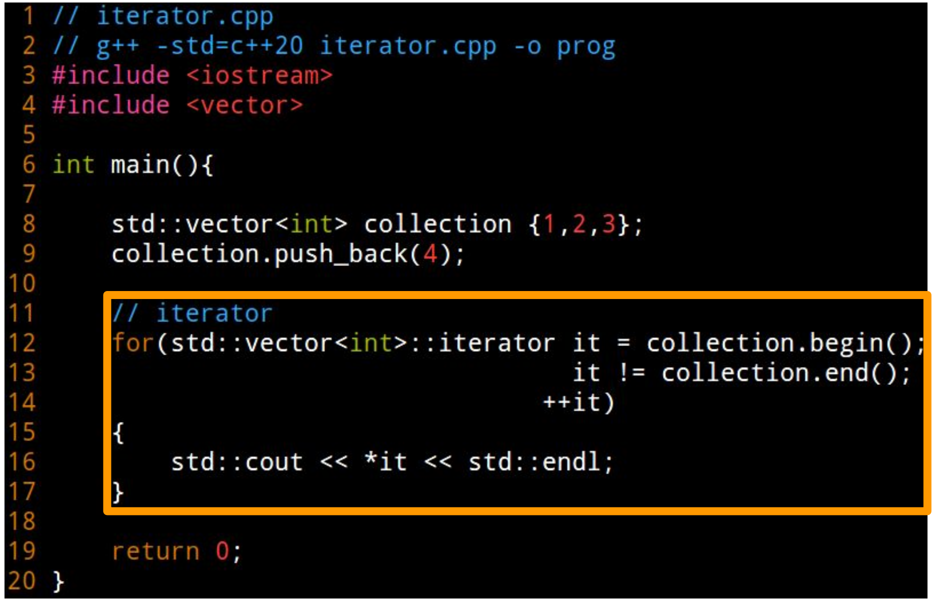
- Let's use a pair of iterators to sequentially move through our entire collection.
    - `collection.begin()` and `collection.end()`
- Using iterators is powerful because we decouple the 'algorithm' of how we iterate (in this example forward iteration) from our actual container data structure.



```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Vector Part 3 - Using Iterators

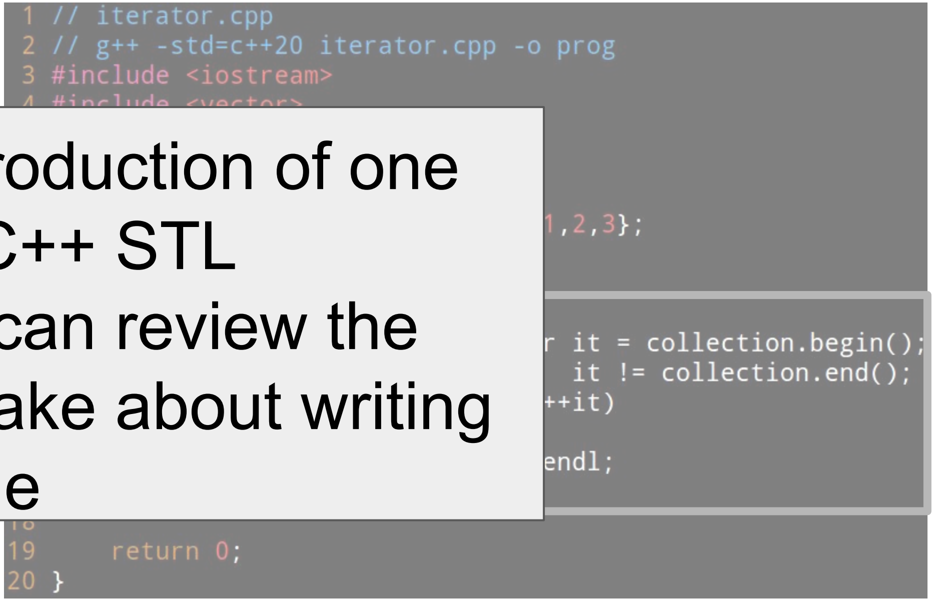
- **Iterators** are a behavioral design pattern:
  - Shows more clearly the intent to iterate through the entire collection from beginning to end
  - Decouples our traversal code from the container.
  - Would likely be easier to change if I decided to change the data structure (e.g. use a `std::list`)
    - (Consistent API in STL)



```
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Vector Part 3 - Using Iterators

- **Iterators** are a behavioral design pattern:
  - Shows more
  - iterate throu
  - from beginni
  - Decouples o
  - container.
  - Would likely
  - decided to c
  - (e.g. use a std::list)
    - (Consistent API in STL)

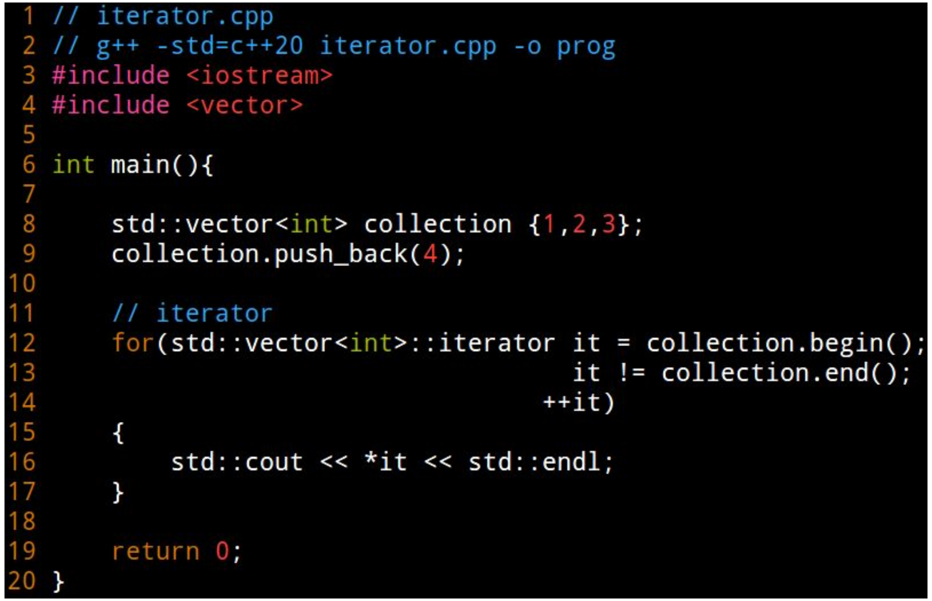


```
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>
1,2,3};
r it = collection.begin();
  it != collection.end();
++it)
endl;

    return 0;
}
```

So with the introduction of one feature in the C++ STL (iterators), we can review the claims I can make about writing better C++ code

Vector Part 3 - Code Review

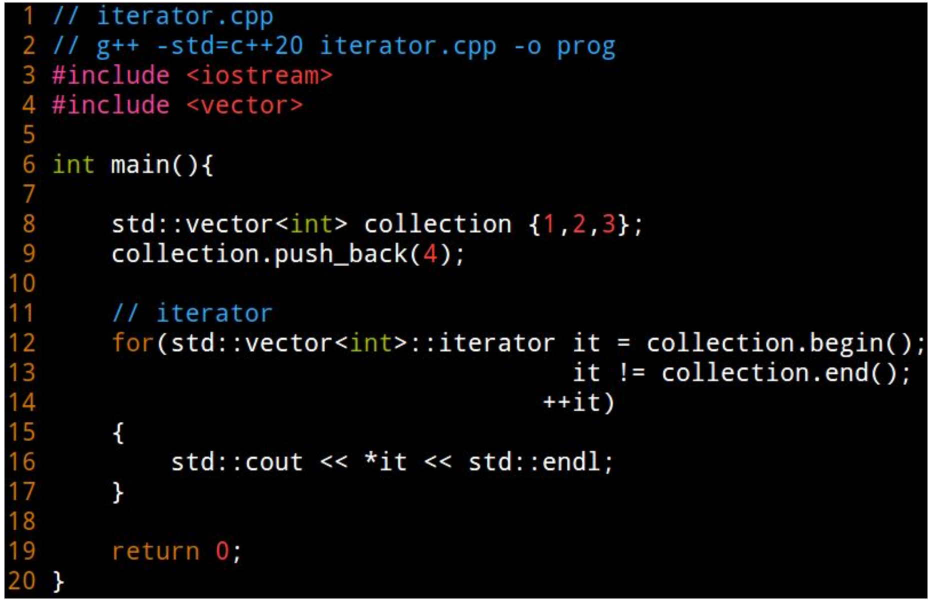
- More precise
  - ??
- More resilient to bugs
  - ??
- More performant
  - ??
- Easier to maintain/reason about
  - ??



```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Vector Part 3 - Code Review

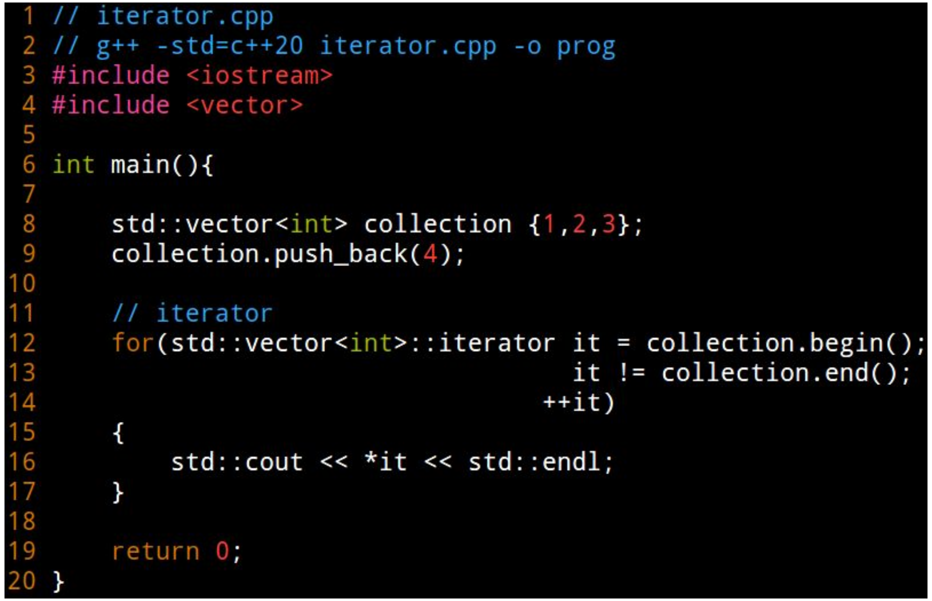
- More precise
  - The intent is clear -- sequentially access one element at a time
- More resilient to bugs
  - ??
- More performant
  - ??
- Easier to maintain/reason about
  - ??



```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                  it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Vector Part 3 - Code Review

- More precise
  - The intent is clear -- sequentially access one element at a time
- More resilient to bugs*
  - The bounds checking invariants hold--I only can look at elements from the start to the finish
- More performant
  - ??
- Easier to maintain/reason about
  - ??

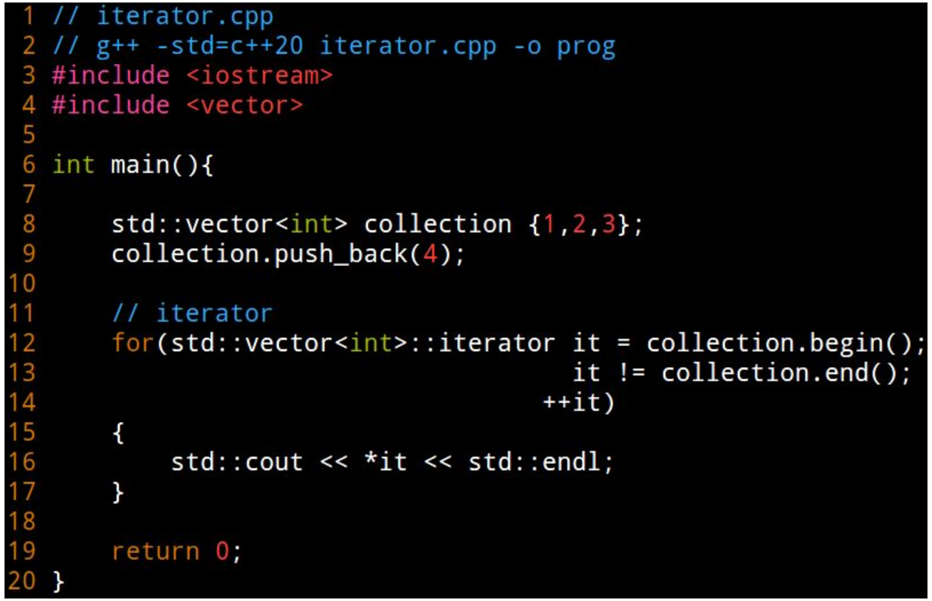


```
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

*Careful when erasing elements, or otherwise operations in multithreaded code

Vector Part 3 - Code Review

- More precise
  - The intent is clear -- sequentially access one element at a time
- More resilient to bugs
  - The bounds checking invariants hold--I only can look at elements from the start to the finish
- More performant*
  - Need to officially measure, but should be equivalent or negligible difference
- Easier to maintain/reason about
  - ??

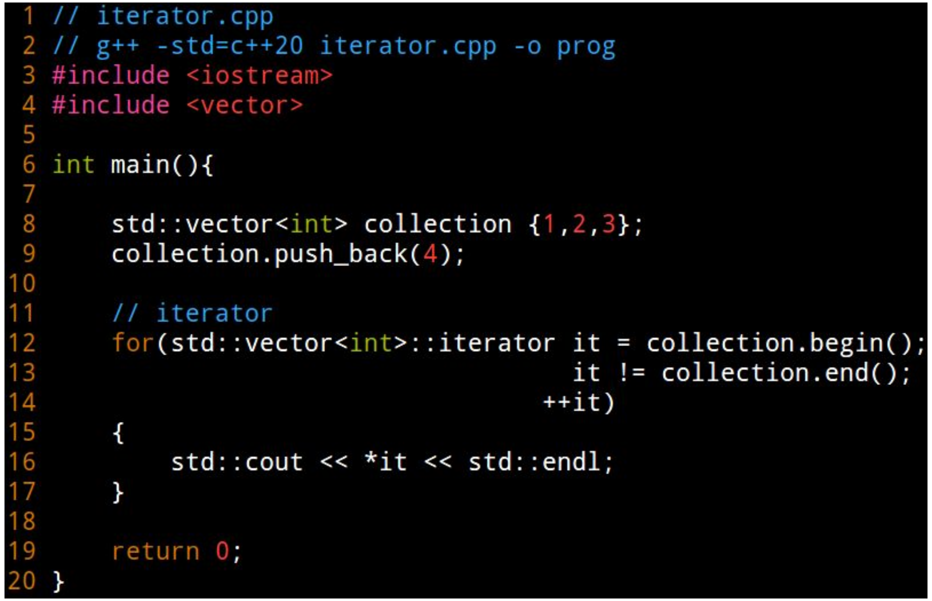


```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                  ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

*Anecdotally iterators have not been a major performance bottleneck in my code versus raw loops.
Okay fine...If you're using SIMD instructions on a vector adding numbers a raw loop can be more automatically or hand-tuned for optimal performance.

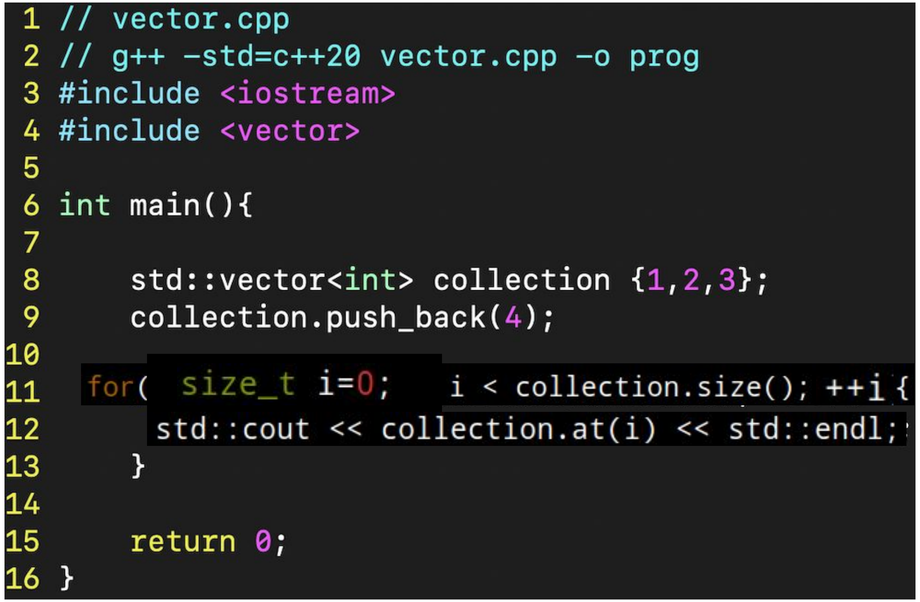
Vector Part 3 - Code Review

- More precise
    - The intent is clear -- sequentially access one element at a time
- More resilient to bugs
    - The bounds checking invariants hold--I only can look at elements from the start to the finish
- More performant
    - Need to officially measure, but should be equivalent or negligible difference
- Easier to maintain/reason about
    - Yes--I'd argue I have less 'things' that I can toggle or have to worry about.
    - I can change the iterator implementation, and the client would not need to do anything (i.e. iterators decouple our traversal from the container)

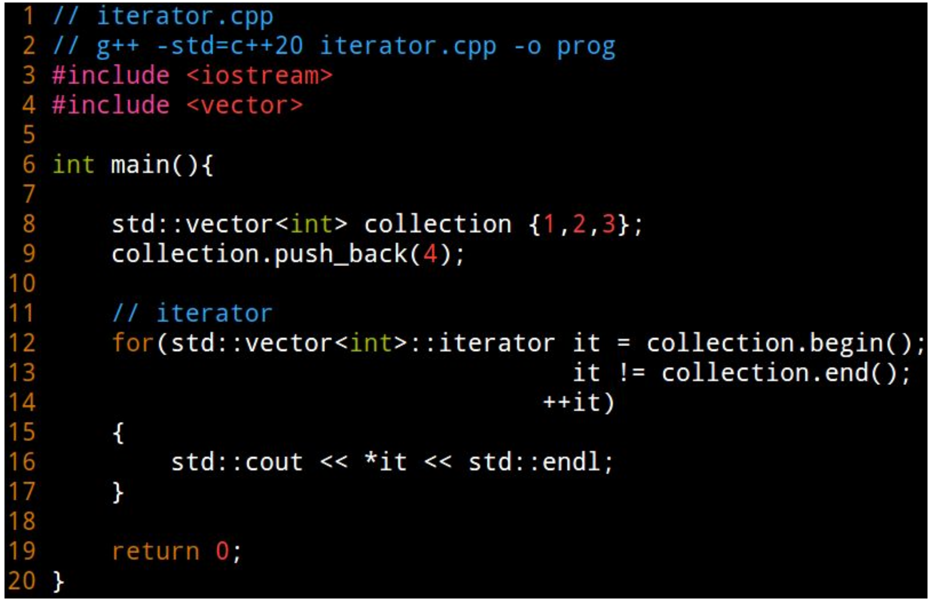


```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Vector Part 2 - Raw Loop | Vector Part 3 - Using Iterators



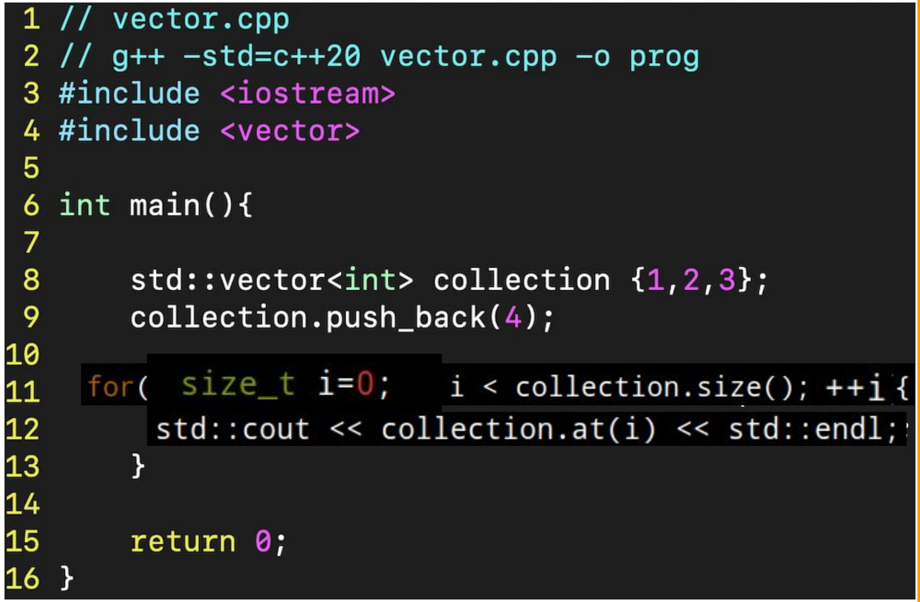
```cpp
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;   i < collection.size(); ++i {
        std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```



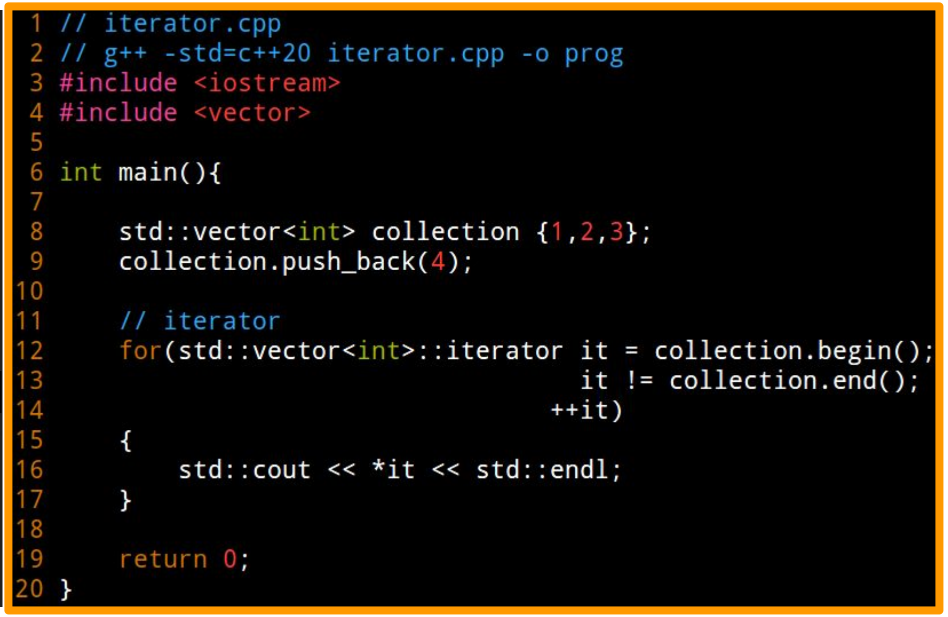
```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

- So the latest example (*Vector Part 3 on the right*) I'm arguing has more qualities of being 'good code' as Vector Part 2 (on the left).

Vector Part 2 - Raw Loop | Vector Part 3 - Using Iterators



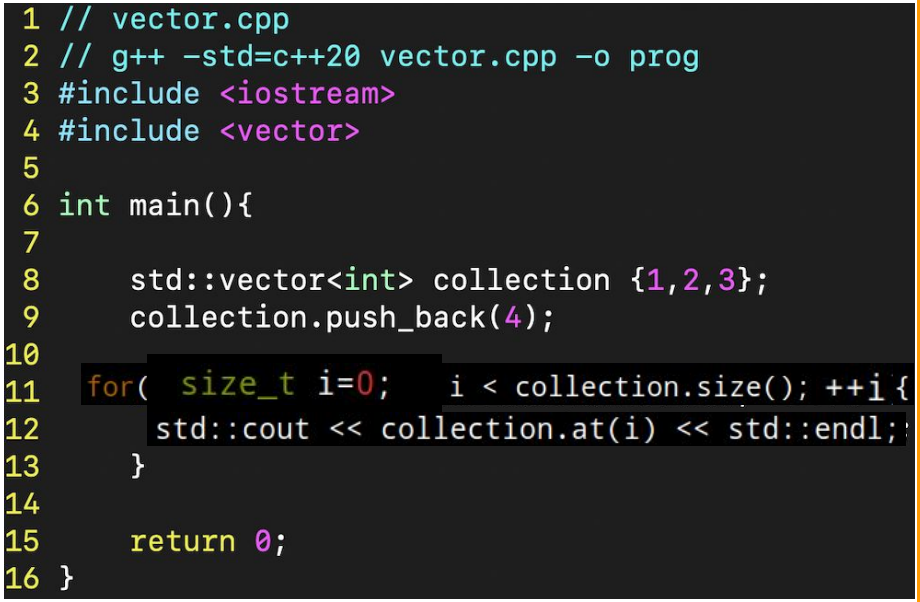
```cpp
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for( size_t i=0;   i < collection.size(); ++i {
        std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```



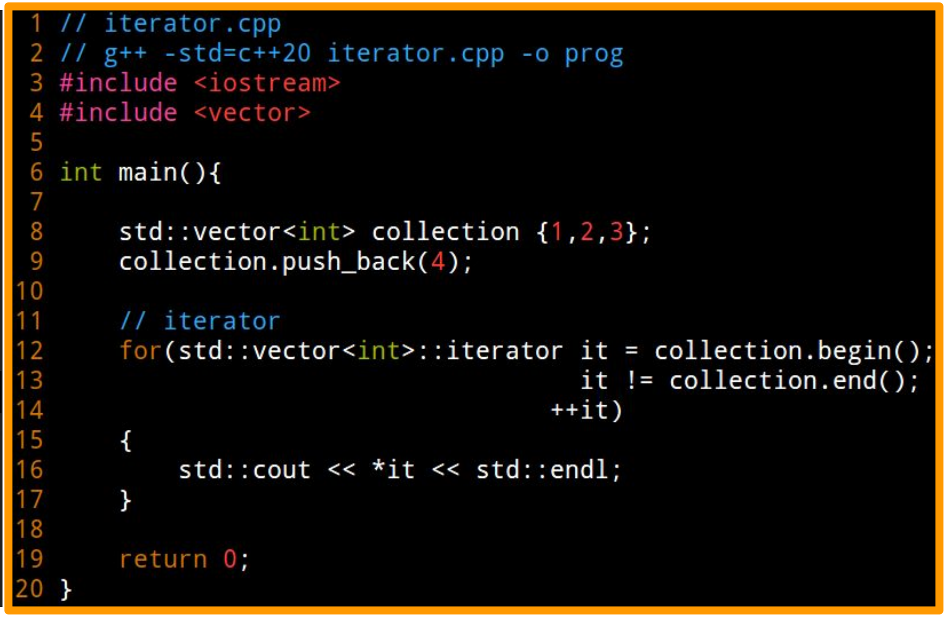
```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                 ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

- For someone new to learning C++, iterators are perhaps intimidating to simply type.
  - (They could use ‘auto’ as well)
- Or perhaps they sound scary and learners avoid learning them when they can simply write code with raw loops like on the left.

Vector Part 2 - Raw Loop | Vector Part 3 - Using Iterators



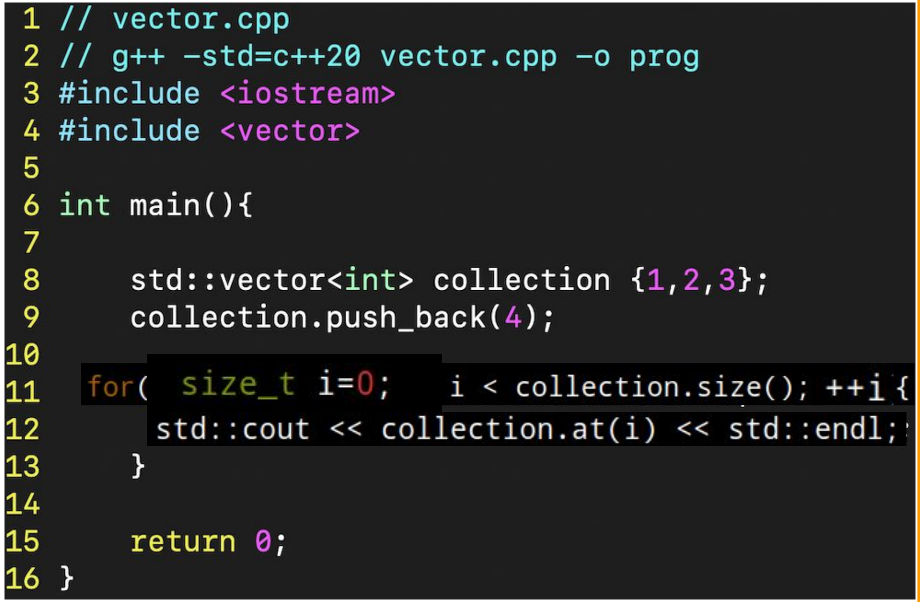
```cpp
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

  for( size_t i=0;   i < collection.size(); ++i {
       std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```



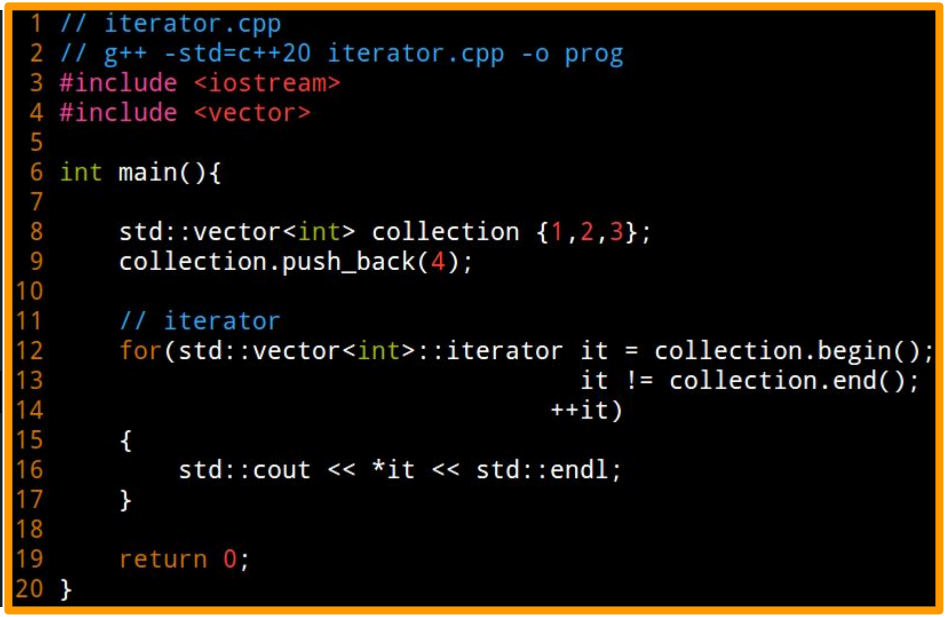
```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                  ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

- But iterators are an important part of the C++ STL, and using them to traverse containers is important for code clarity.
- You can reason about the start and end of a computation more easily.

Vector Part 2 - Raw Loop | Vector Part 3 - Using Iterators



```cpp
// vector.cpp
// g++ -std=c++20 vector.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

  for( size_t i=0;   i < collection.size(); ++i {
       std::cout << collection.at(i) << std::endl;
    }

    return 0;
}
```

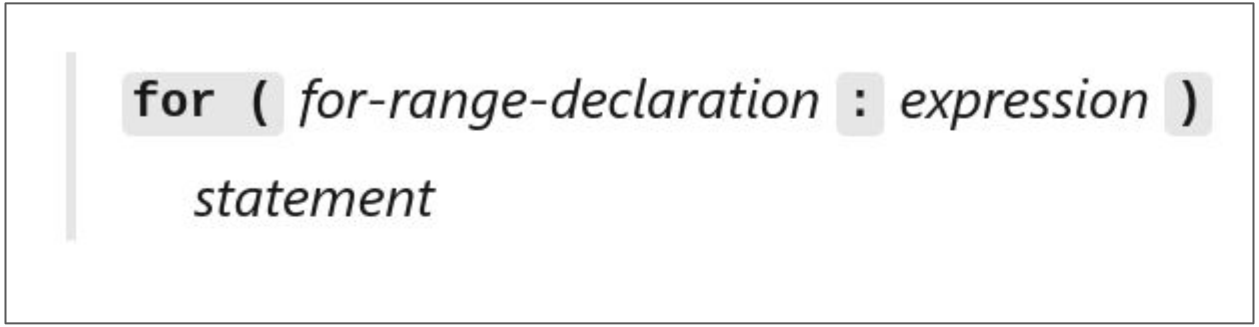


```cpp
// iterator.cpp
// g++ -std=c++20 iterator.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    // iterator
    for(std::vector<int>::iterator it = collection.begin();
                                   it != collection.end();
                                  ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

- I also have some good news for those who do not like typing....

Introducing Ranged-Based For-loop -- **more concise syntax**

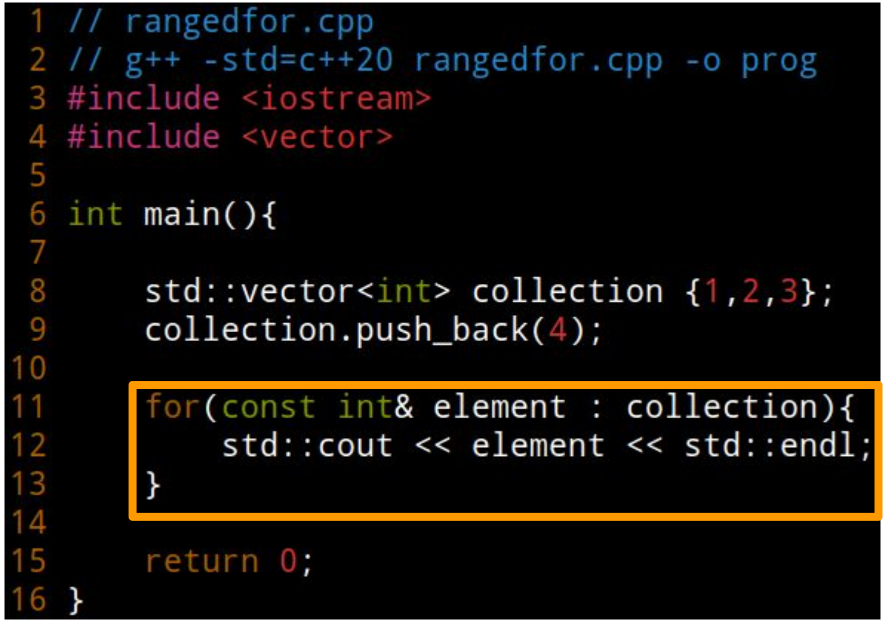
This is Programming - Part 4 `std::vector`



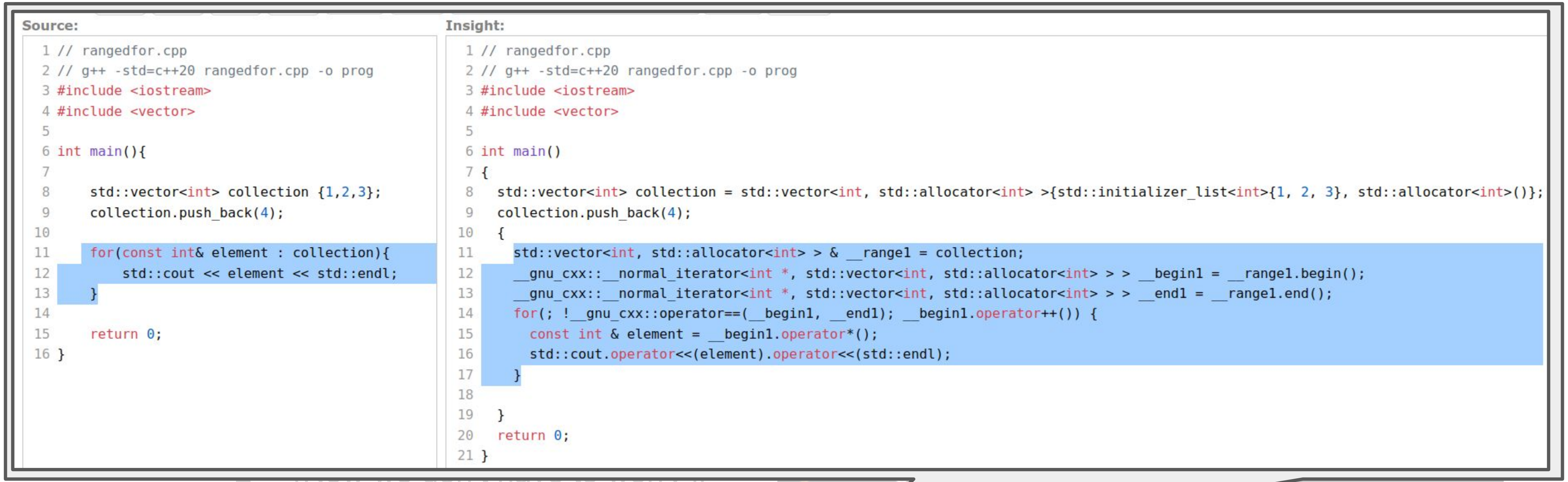
```
for ( for-range-declaration : expression )
    statement
```

Vector Part 4

- C++11 introduced a ranged-based for loop [cppref].
  - The syntax is more concise
  - The loop itself signals our intent to iterate through every element.
    - (Now we don’t have to worry if we get the bounds on the loop condition correct either)



```cpp
// rangedfor.cpp
// g++ -std=c++20 rangedfor.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(const int& element : collection){
        std::cout << element << std::endl;
    }

    return 0;
}
```



Source:

```
// rangedfor.cpp
// g++ -std=c++20 rangedfor.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(const int& element : collection){
        std::cout << element << std::endl;
    }

    return 0;
}
```

Insight:

```
// rangedfor.cpp
// g++ -std=c++20 rangedfor.cpp -o prog
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> collection = std::vector<int, std::allocator<int> >{std::initializer_list<int>{1, 2, 3}, std::allocator<int>()};
  collection.push_back(4);
  {
    std::vector<int, std::allocator<int> > & __range1 = collection;
    __gnu_cxx::__normal_iterator<int *, std::vector<int, std::allocator<int> > > __begin1 = __range1.begin();
    __gnu_cxx::__normal_iterator<int *, std::vector<int, std::allocator<int> > > __end1 = __range1.end();
    for(; !__gnu_cxx::operator==(__begin1, __end1); __begin1.operator++()) {
      const int & element = __begin1.operator*();
      std::cout.operator<<(element).operator<<(std::endl);
    }

  }
  return 0;
}
```

```
(const int& element : collection){
    std::cout << element << std::endl;
}

return 0;
```

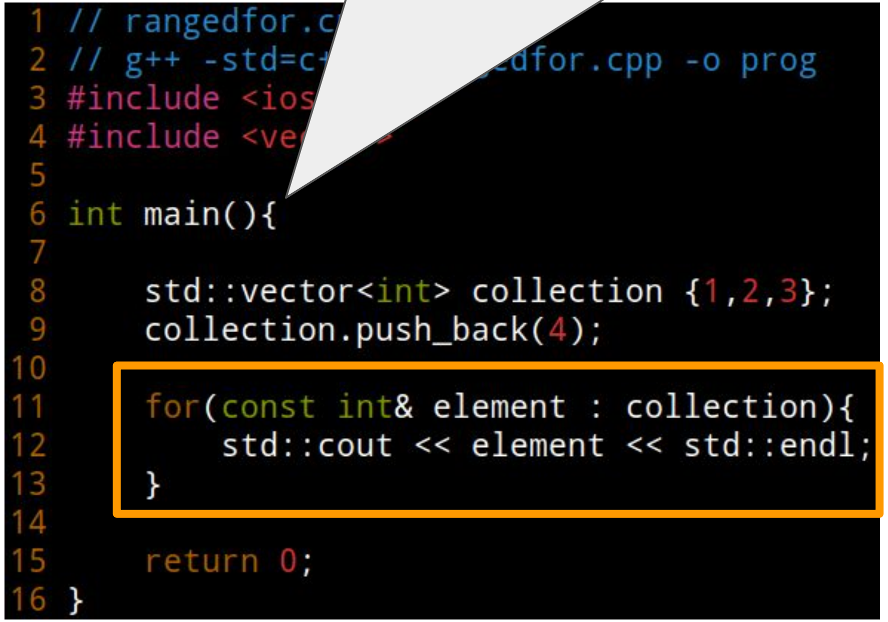
(Aside) How is a Range implemented?

- Using the tool 'cppinsights' notice a 'ranged-based for loop' is translated into code using a forward iterator.
  - (On the left I have the ranged-loop, and on the right the insight in the code).
  - https://cppinsights.io/s/1200df99

Vector Part 4

Now to me, this is a bit of a revelation, and one of the first things that made me actually say "C++ is an elegant language"

- Now, some of you students who were lucky learned C++11
- C++11 introduced a ranged-based for loop [cppref].
  - This shows our intent, to operate on each element in a collection
    - (Now don't have to worry if we get the conditions correct on the for-loop)



```
// rangedfor.c
// g++ -std=c+            dfor.cpp -o prog
#include <ios
#include <ve

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    for(const int& element : collection){
        std::cout << element << std::endl;
    }

    return 0;
}
```

Jump straight into optimization land

Continue learning foundational building blocks

Continue further

Iterators provide us a great building block for defining ranges of computation!

Our **exciting** journey is starting!

https://media.istockphoto.com/photos/cross-roads-horizon-picture-id1355264492k=20&m=135526449&s=612x612&w=0&h=[illegible]

And we'll take a look at some more interesting problems -- and get to `std::algorithm`

This is Programming - Part 5 Algorithmic Thinking

Iterators library
Ranges library (C++20)
Algorithms library
Constrained algorithms (C++20)

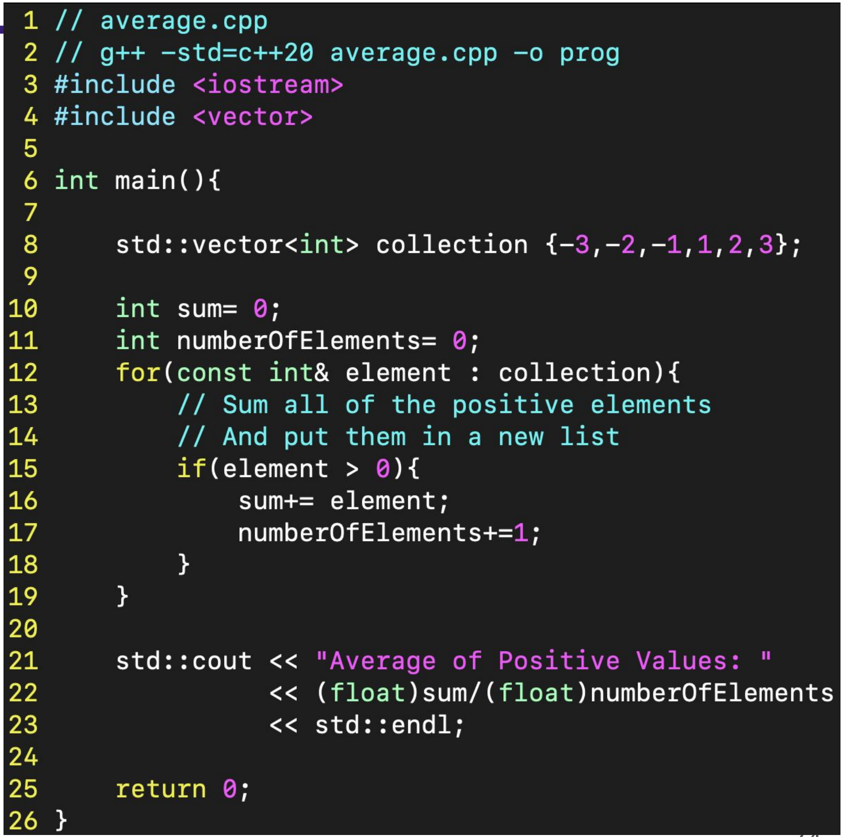
Compute the Average of Positive Numbers in `std::vector`

- So here's a potential problem
    - **Compute the average of all of the integers greater than zero in a collection.**
- Probably fairly trivial for us--but let's look at some possible solutions.

```cpp
// average.cpp
// g++ -std=c++20 average.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};

    int sum= 0;
    int numberOfElements= 0;
    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;
            numberOfElements+=1;
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)numberOfElements
              << std::endl;

    return 0;
}
```

Compute the Average of Positive Numbers in `std::vector`

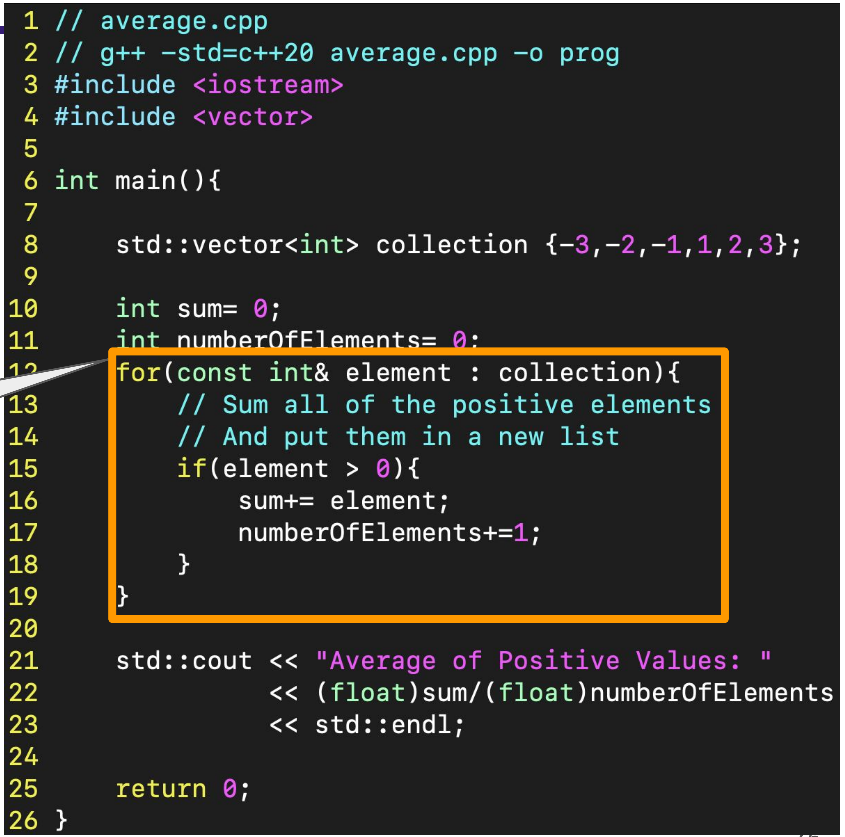
- So here's a potential problem
  - **Compute the average of all of the integers greater than zero in a collection.**
- Probably fairly trivial for us--but let's look at some possible solutions.



```cpp
// average.cpp
// g++ -std=c++20 average.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};

    int sum= 0;
    int numberOfElements= 0;
    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;
            numberOfElements+=1;
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)numberOfElements
              << std::endl;

    return 0;
}
```

Compute the Average of Positive Numbers in `std::vector`

- So here's a potential problem
  - **Compute the average of all of the integers greater than zero in a collection.**
- Probably fairly trivial for us--but let's look at some possible solutions.



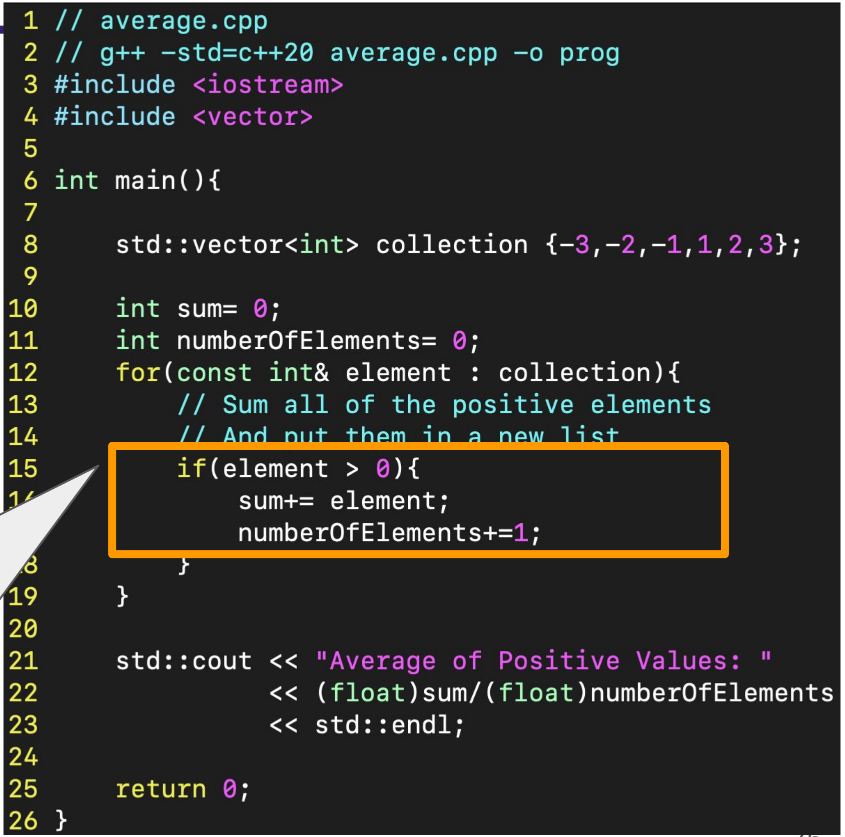
```
// average.cpp
// g++ -std=c++20 average.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};

    int sum= 0;
    int numberOfElements= 0;
    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;
            numberOfElements+=1;
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)numberOfElements
              << std::endl;

    return 0;
}
```

- **Simple solution:**
  - **Loop through all of our elements**

Compute the Average of Positive Numbers in `std::vector`

- So here's a potential problem
  - **Compute the average of all of the integers greater than zero in a collection.**
- Probably fairly trivial for us--but let's look at some possible solutions.

- Simple solution:
  - Loop through all of our elements
  - **Test values that are greater than 0**
  - **Accumulate the total elements that satisfy this condition**

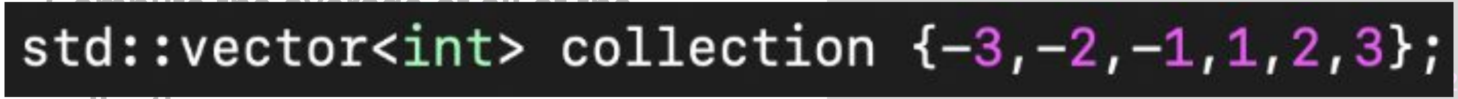


```cpp
// average.cpp
// g++ -std=c++20 average.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};

    int sum= 0;
    int numberOfElements= 0;
    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;
            numberOfElements+=1;
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)numberOfElements
              << std::endl;

    return 0;
}
```

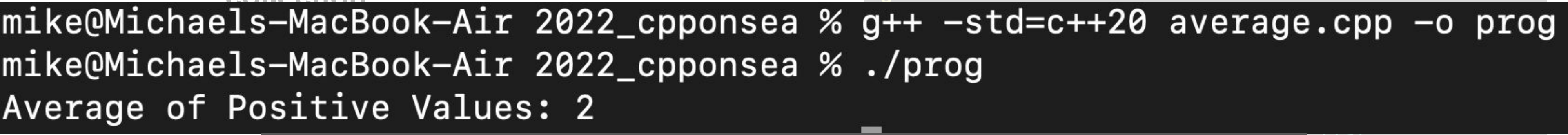
Compute the Average of Positive Numbers in `std::vector`

- So here's a potential problem
  - Compute the average of all of the
    collection
- Simple
  - Lo
  - Test values that are greater than 0
  - Accumulate the total elements that satisfy this condition

```
// average.cpp
// g++ -std=c++20 average.cpp -o prog
#include <iostream>
#include <vector>

,-1,1,2,3};

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)numberOfElements
              << std::endl;

    return 0;
}
```



```
std::vector<int> collection {-3,-2,-1,1,2,3};
```



```
mike@Michaels-MacBook-Air 2022_cpponsea % g++ -std=c++20 average.cpp -o prog
mike@Michaels-MacBook-Air 2022_cpponsea % ./prog
Average of Positive Values: 2
```

I can confirm this solution works with our sample data

Average of Positive Numbers - Round 2

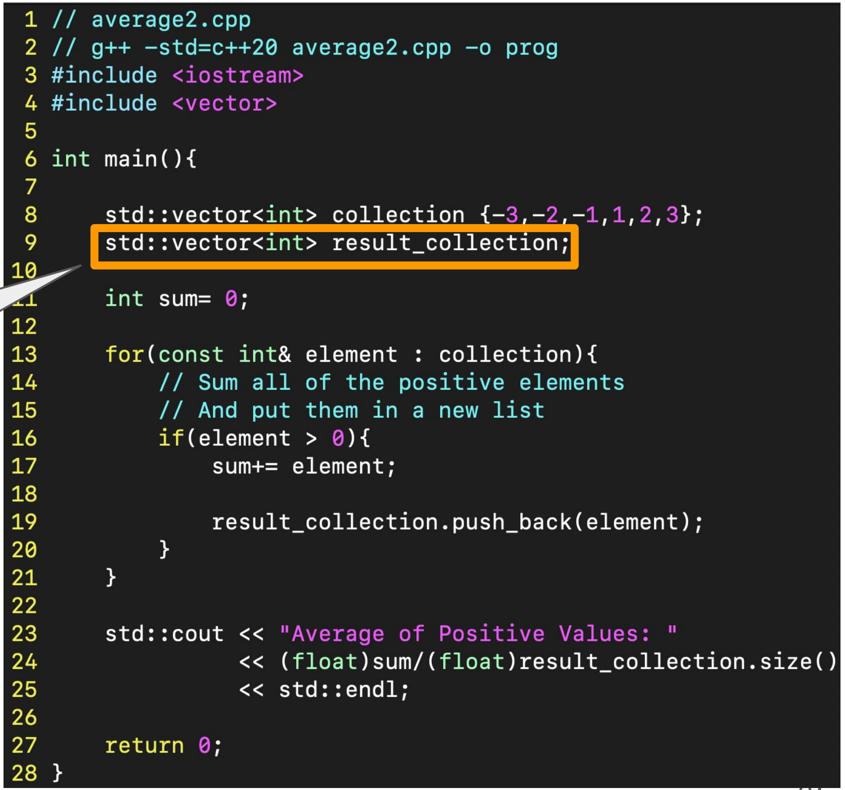
- So let’s extend the program:
    - Compute the average of all of the integers greater than zero in a collection.
    - ****And we also want to keep the resulting set of values.****

```cpp
// average2.cpp
// g++ -std=c++20 average2.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)result_collection.size()
              << std::endl;

    return 0;
}
```

Average of Positive Numbers - Round 2

- So let's extend the program
  - Compute the average of all of the integers greater than zero in a collection.
  - ****And we also want to keep the resulting set of values.****
- Probably fairly trivial for u s

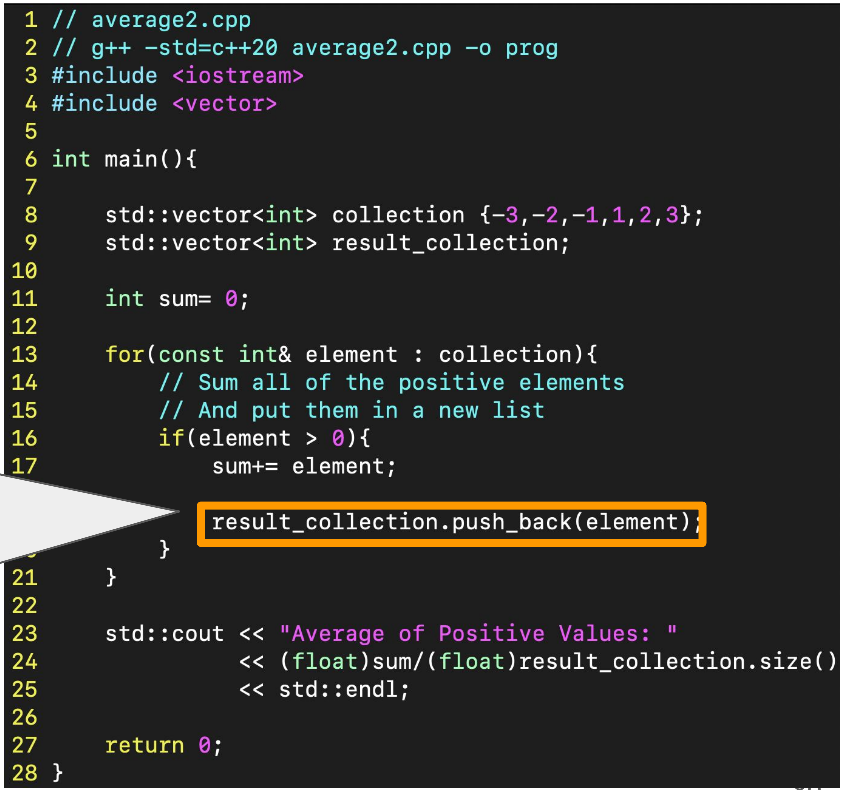
- **Add in a collection to store our results**



```
// average2.cpp
// g++ -std=c++20 average2.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)result_collection.size()
              << std::endl;

    return 0;
}
```

Average of Positive Numbers - Round 2

- So let's extend the program
  - Compute the average of all of the integers greater than zero in a collection.
  - ****And we also want to keep the resulting set of values.****
- Probably fairly trivial for us--but let's



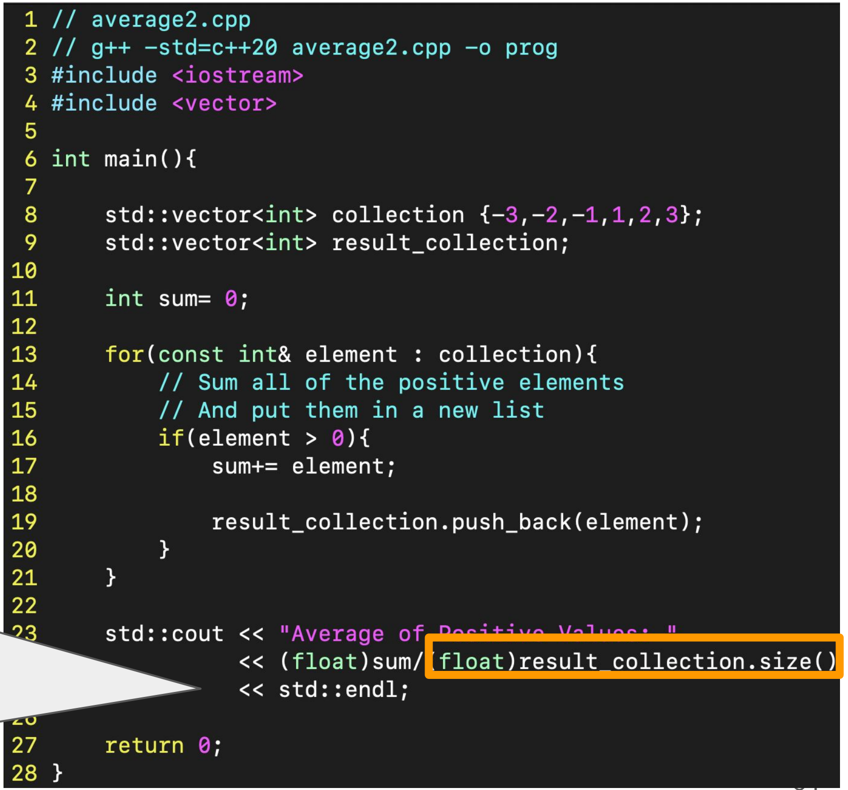
```
// average2.cpp
// g++ -std=c++20 average2.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)result_collection.size()
              << std::endl;

    return 0;
}
```

- Add in a collection to store our results
- **Store each element in the new collection**

Average of Positive Numbers - Round 2

- So let's extend the program
  - Compute the average of all of the integers greater than zero in a collection.
  - ****And we also want to keep the resulting set of values.****
- Probably fairly trivial for us--but let's

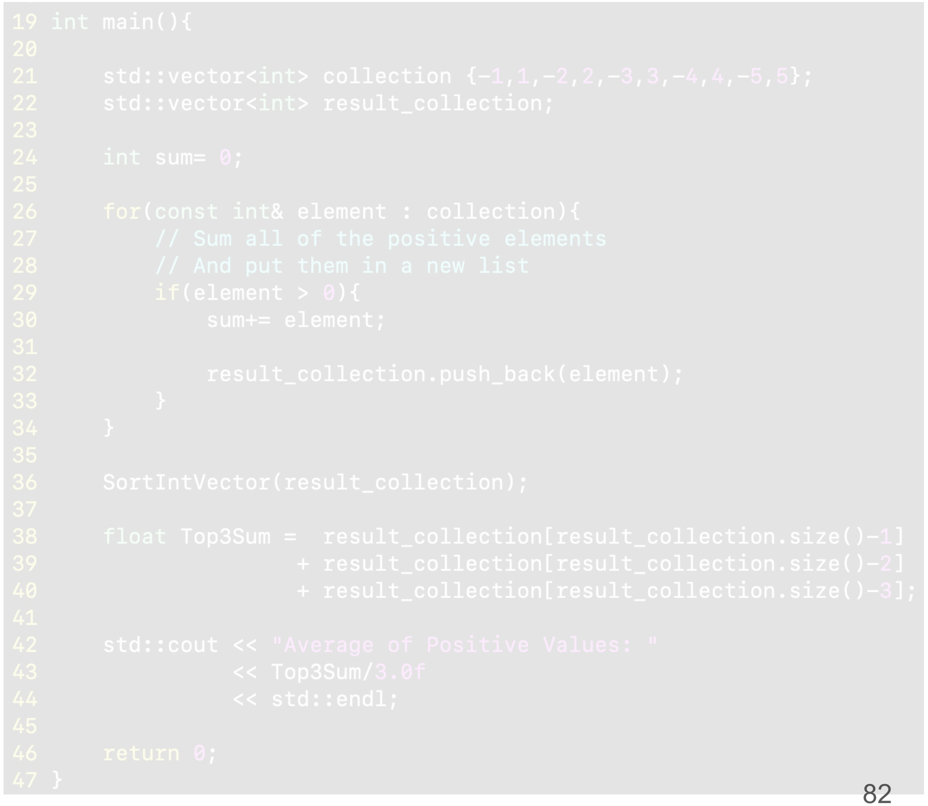
- Add in a collection to store our results
- Store each element in the new collection
- **Can take advantage of the collection size now.**



```cpp
// average2.cpp
// g++ -std=c++20 average2.cpp -o prog
#include <iostream>
#include <vector>

int main(){

    std::vector<int> collection {-3,-2,-1,1,2,3};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    std::cout << "Average of Positive Values: "
              << (float)sum/(float)result_collection.size()
              << std::endl;

    return 0;
}
```

Average of Positive Numbers - Round 3 **(Top 3 numbers average)**

- Let’s extend our program further
    - Compute the average of all of the integers greater than zero in a collection.
    - And we also want to keep the resulting set of values.
    - ****Now I want you to only take the average of the top 3 values****

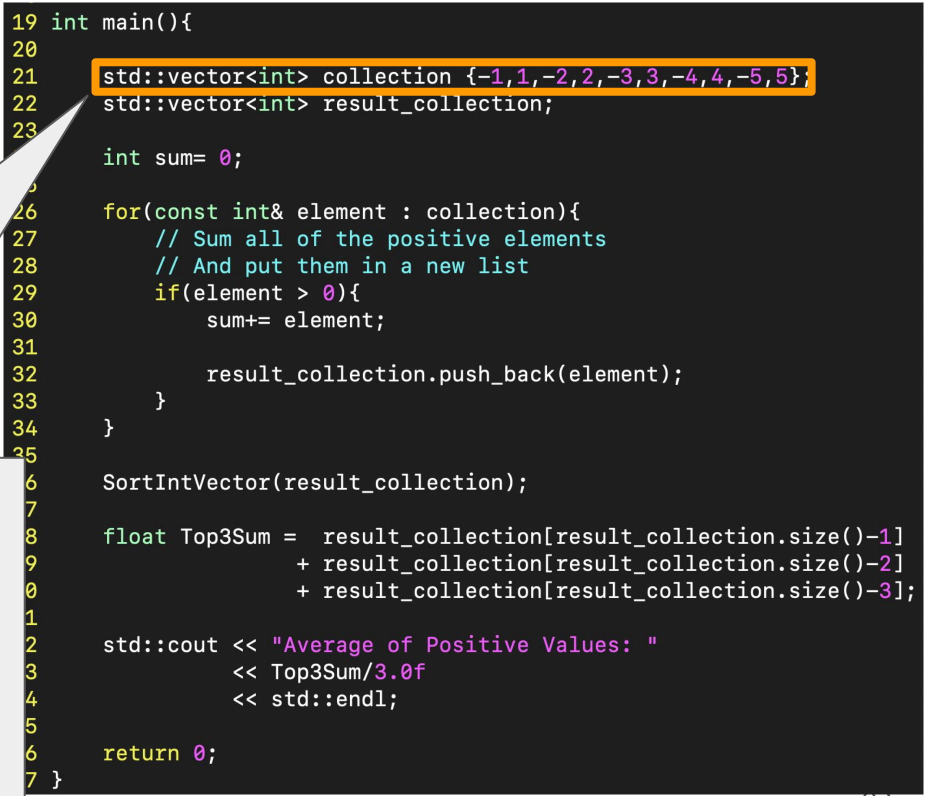


```cpp
int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

Average of Positive Numbers - Round 3 **(Top 3 numbers average)**

- Let’s extend our program further
  - Compute the average of all of the integers greater than zero in a collection.
  - And we also want to keep the resu
    set of values.
  - ****Now I want you to only**
    average of the top 3

- **Added some more test data**

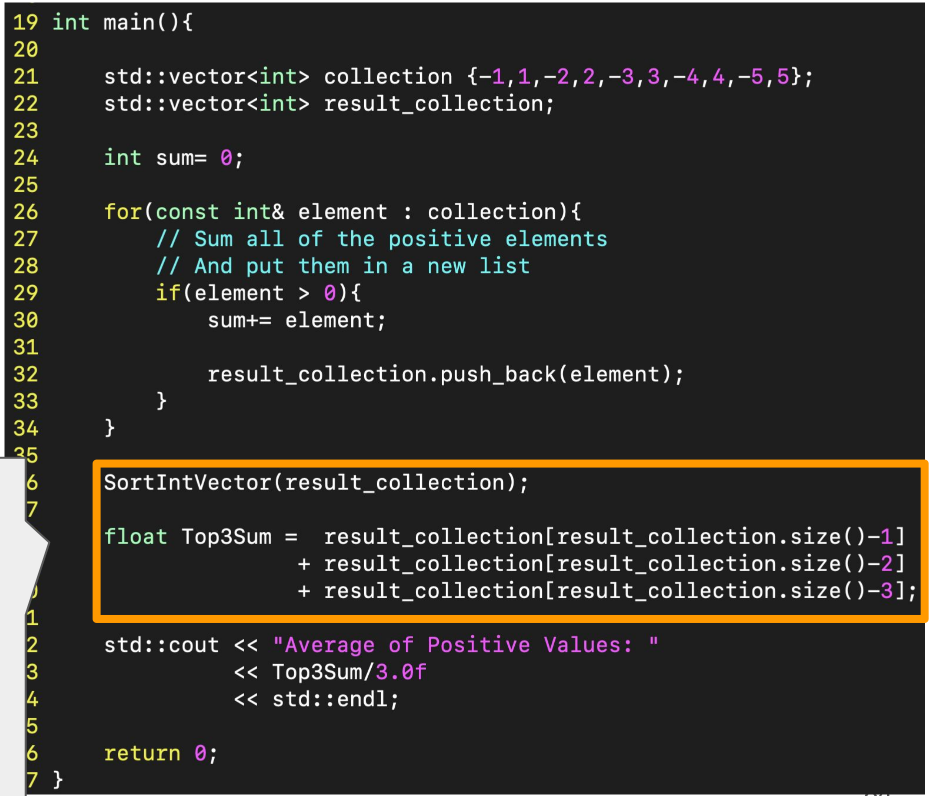


```
int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

Average of Positive Numbers - Round 3 **(Top 3 numbers average)**

- Let’s extend our program further
  - Compute the average of all of the integers greater than zero in a collection.
  - And we also want to keep the resulting set of values.
  - ****Now I want you to only take the average of the top 3 values****

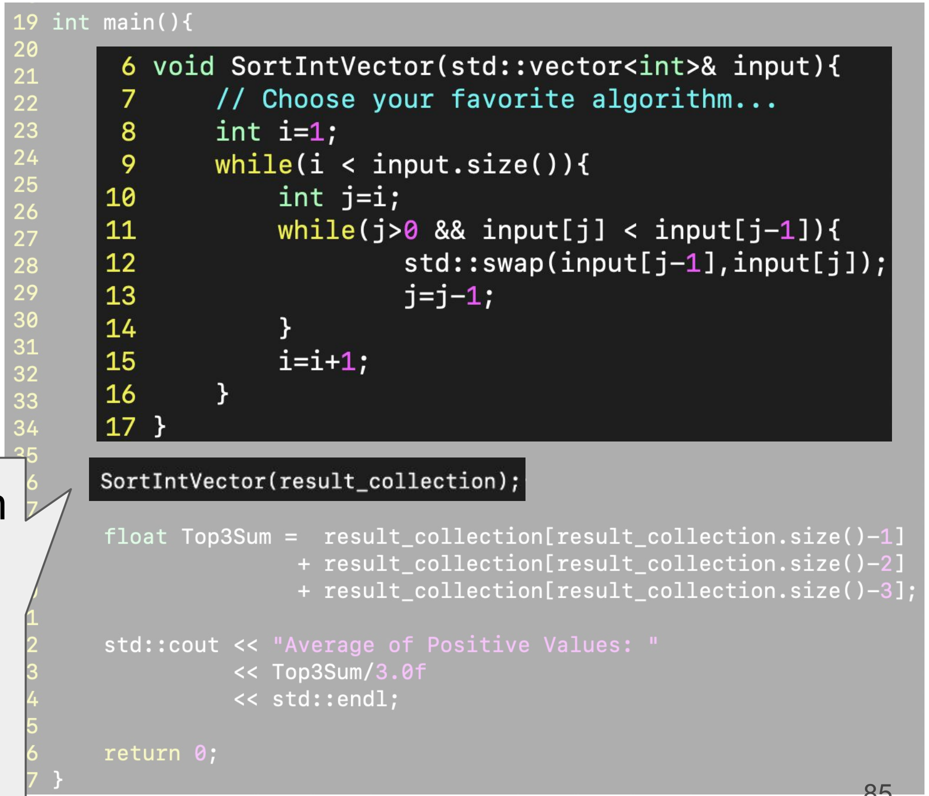
- **I have to think a bit here, but my algorithm is to ‘sort the values’, then I’ll just take the top 3 values.**



```cpp
int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

Average of Positive Numbers - Round 3 **(Top 3 numbers average)**

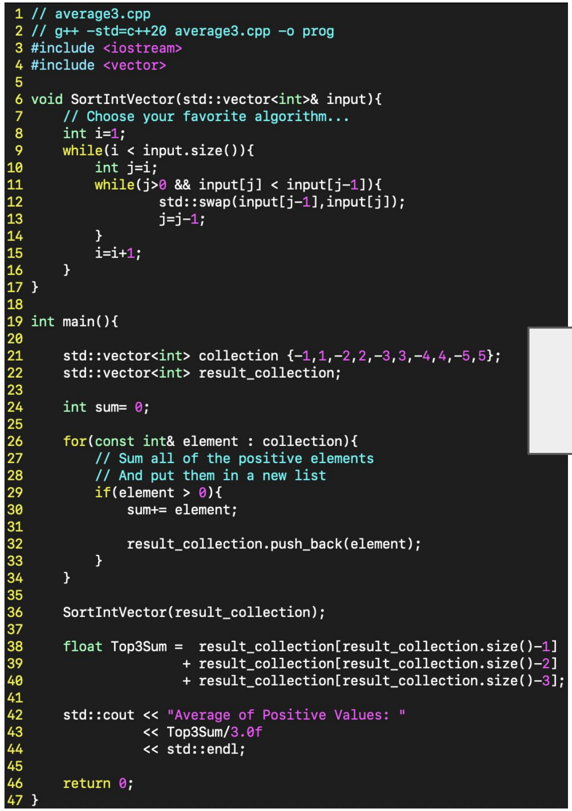
- Let’s extend our program further
  - Compute the average of all of the integers greater than zero in a collection.
  - And we also want to keep the resulting set of values.
  - ****Now I want you to only take the average of the top 3 values****



```
int main(){

    void SortIntVector(std::vector<int>& input){
        // Choose your favorite algorithm...
        int i=1;
        while(i < input.size()){
            int j=i;
            while(j>0 && input[j] < input[j-1]){
                    std::swap(input[j-1],input[j]);
                    j=j-1;
            }
            i=i+1;
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

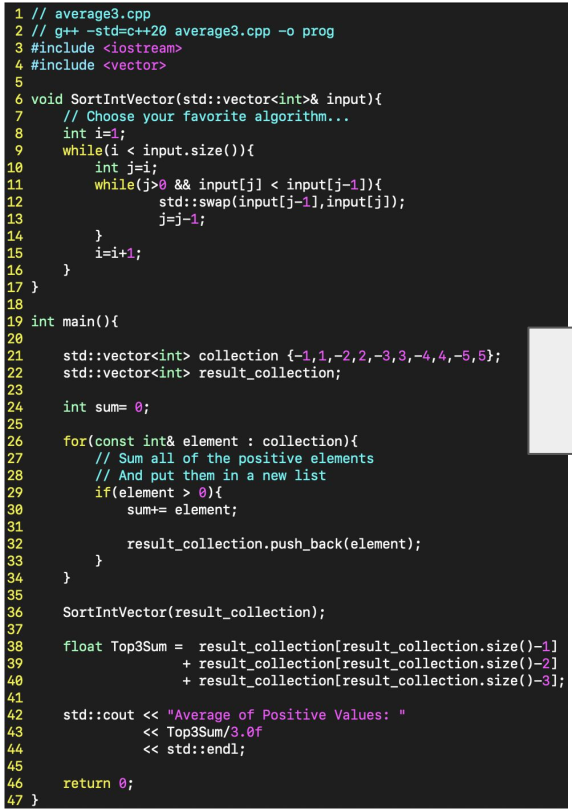
- I have to think a bit here, but my algorithm is to ‘sort the values’, then I’ll just take the top 3 values.
- **And my sort function--I had to think a bit more about that...**

So here we are, I’ve solved a non-trivial problem on our journey, adding conditionals and writing algorithms along the way.

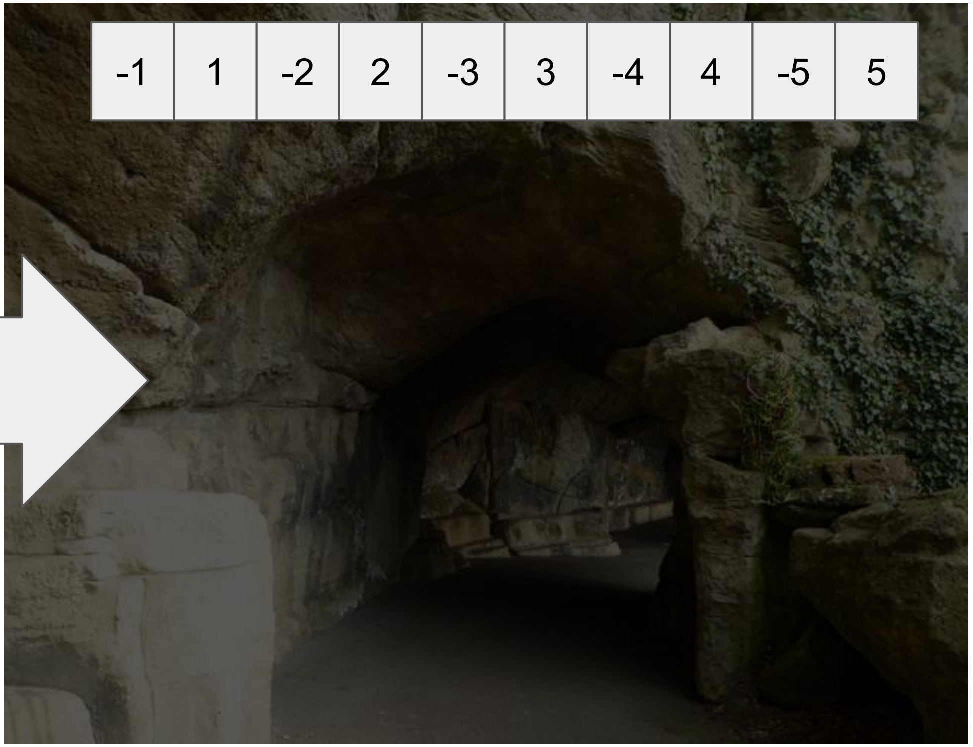


```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

We acquired some data

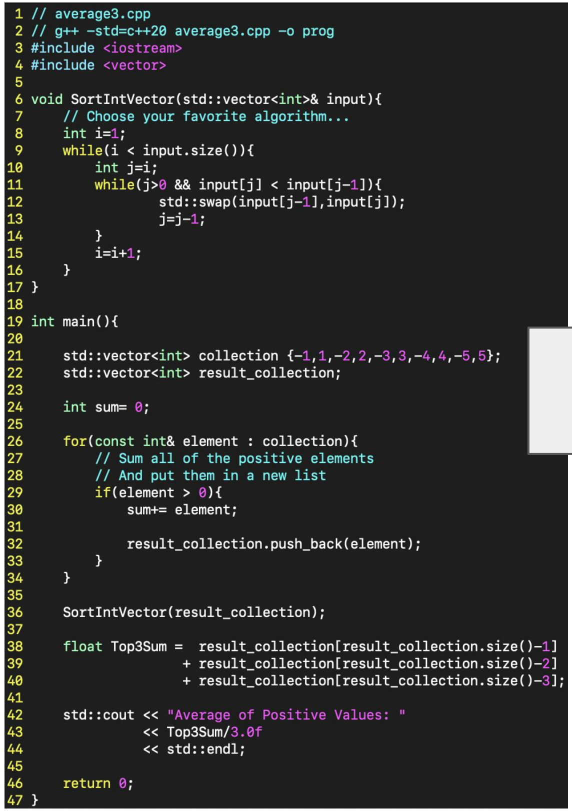


```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

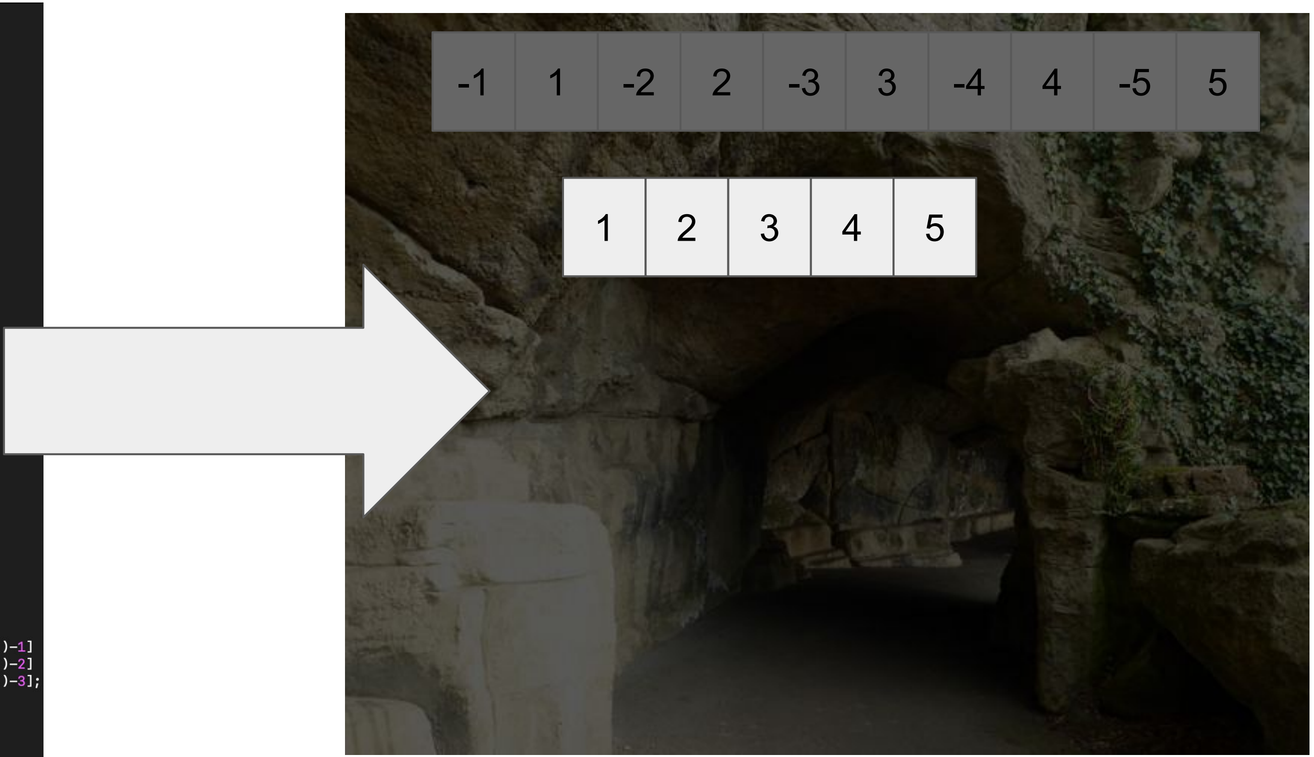


-1	1	-2	2	-3	3	-4	4	-5	5

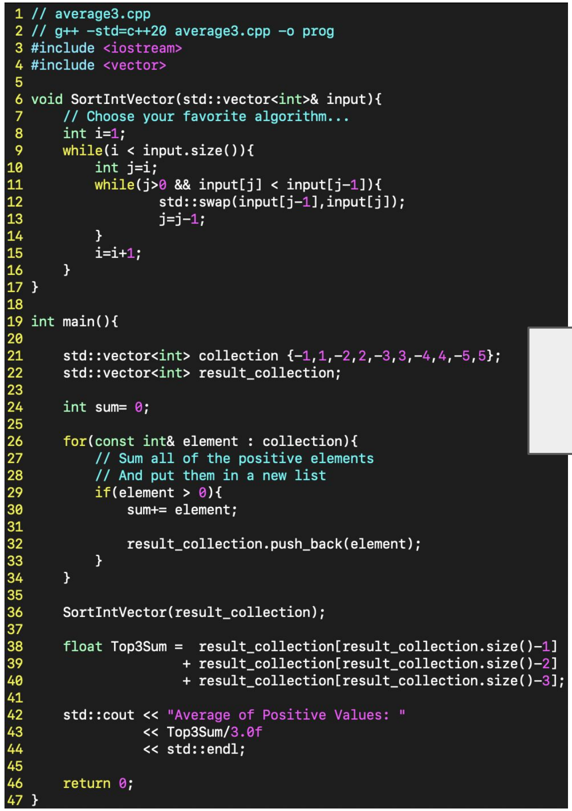
‘filtered’ it to a new collection



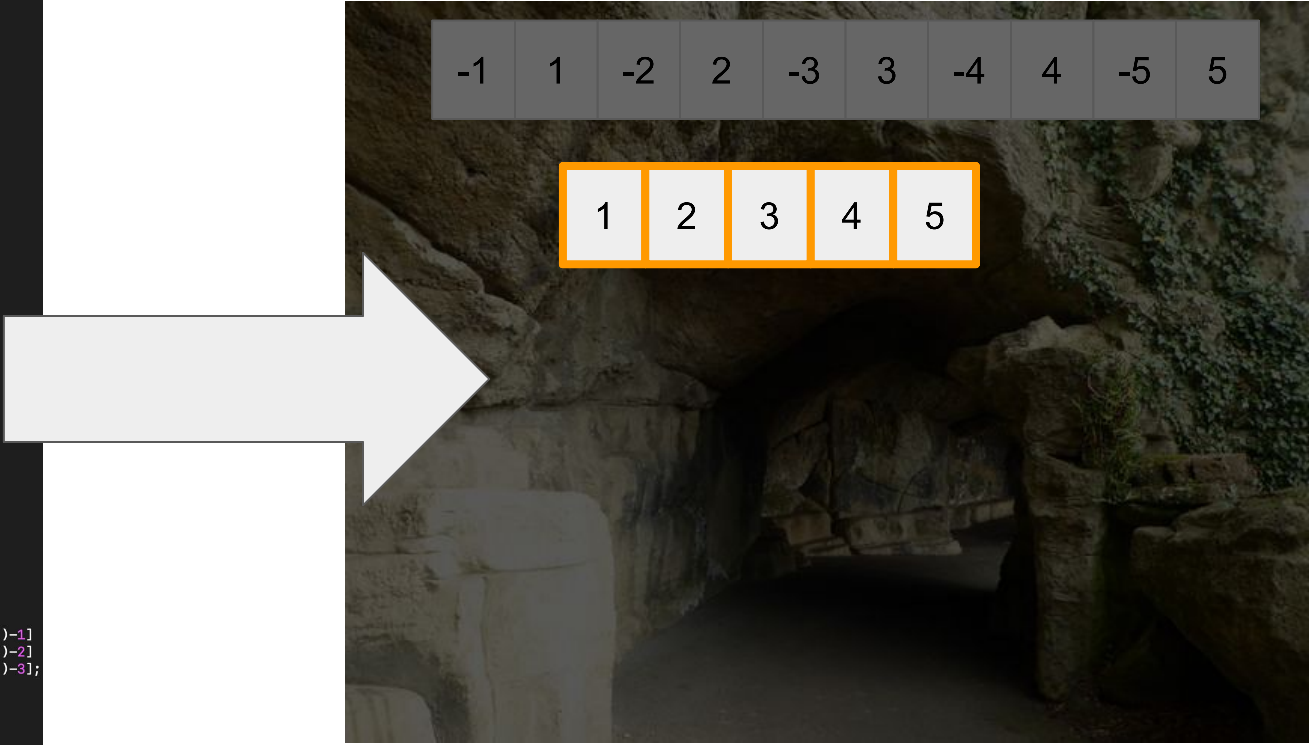
```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```



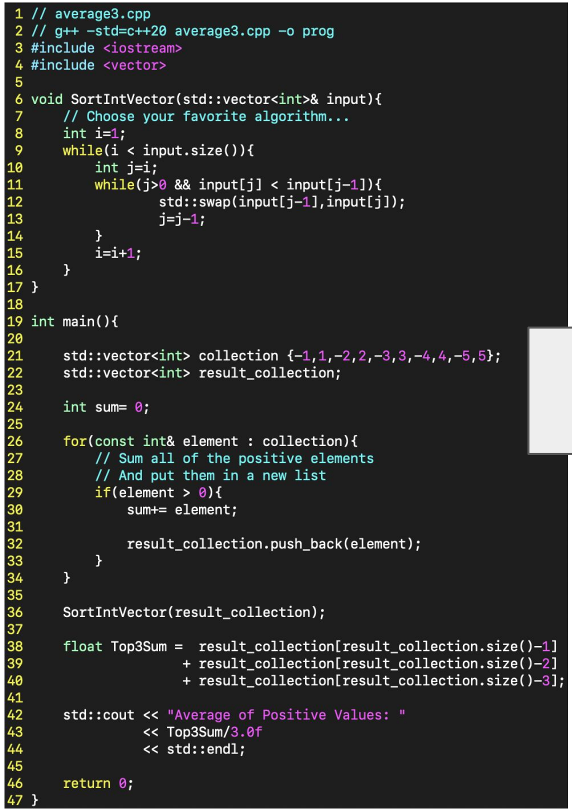
Sorted the data (it was already sorted)



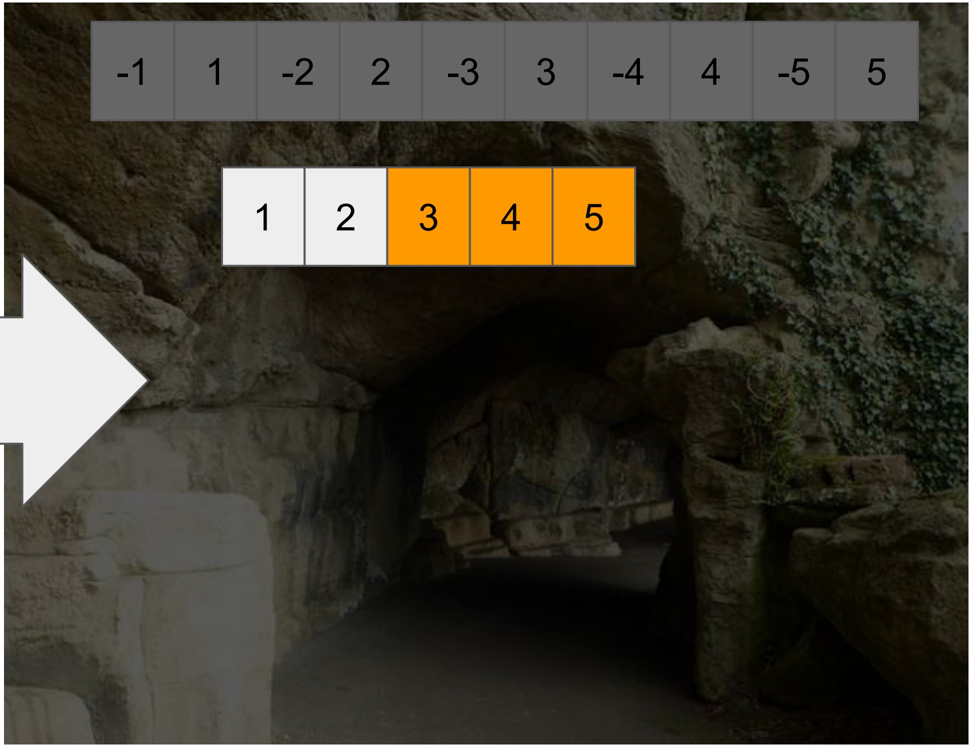
```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```



Then computed a result taking the average of the top 3 values

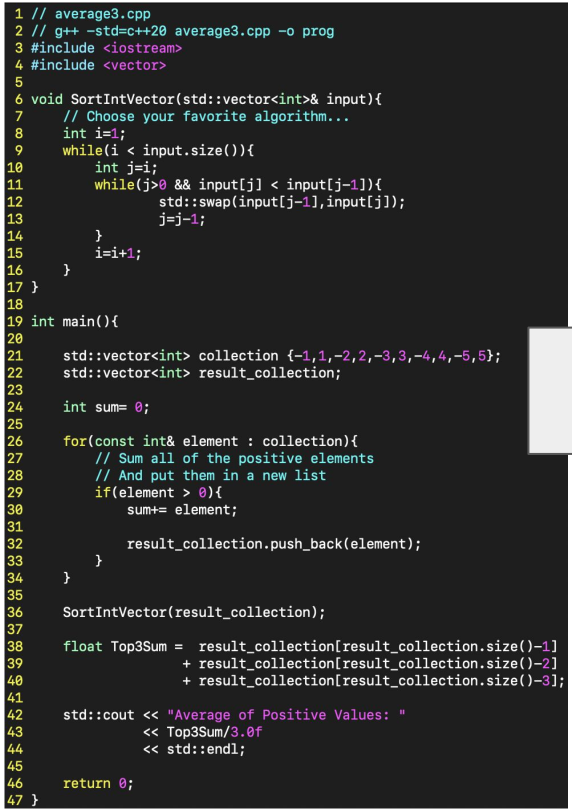


```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

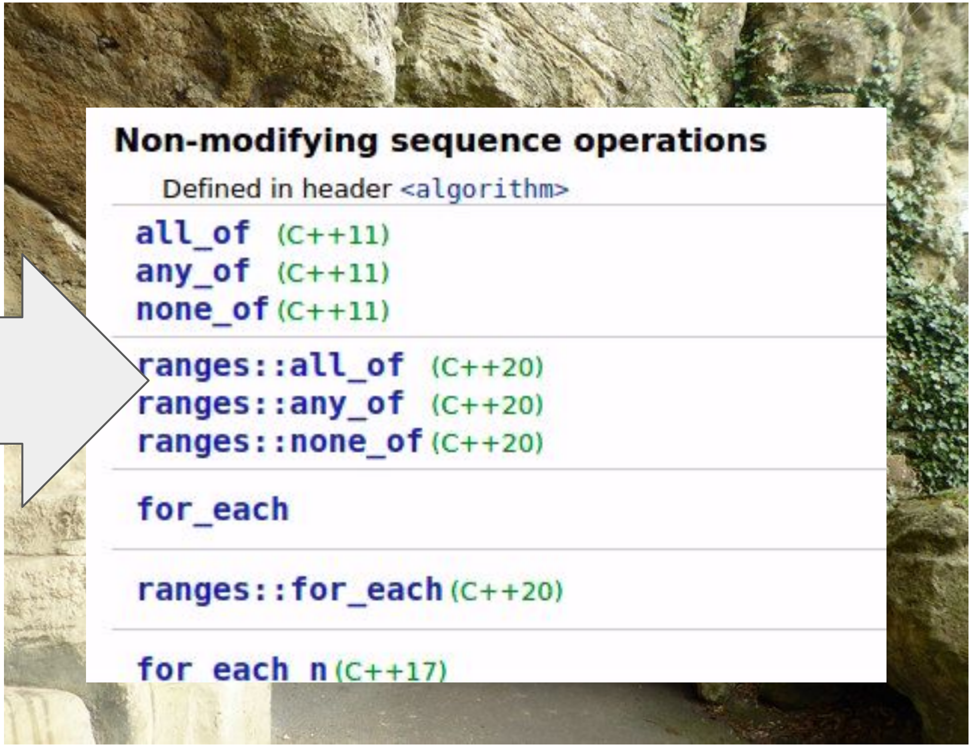


Now let me show you using `std::algorithm`

-- when writing this I spent my time thinking as opposed to writing code.



```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```



Non-modifying sequence operations

Defined in header `<algorithm>`

`all_of` (C++11)
`any_of` (C++11)
`none_of` (C++11)

`ranges::all_of` (C++20)
`ranges::any_of` (C++20)
`ranges::none_of` (C++20)

`for_each`

`ranges::for_each` (C++20)

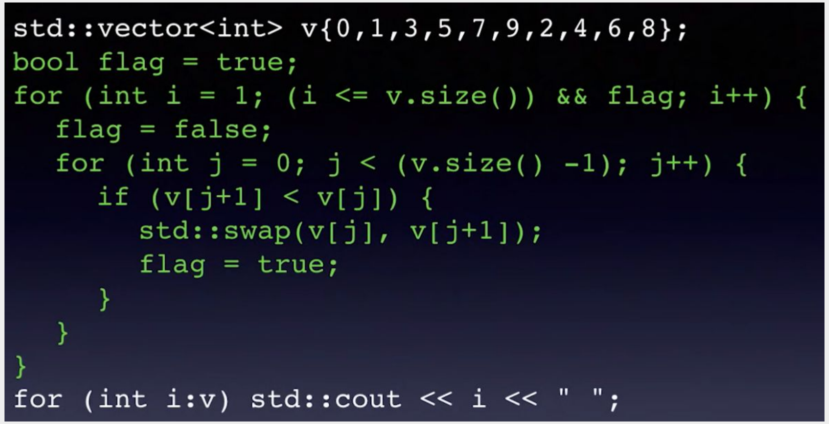
`for_each_n` (C++17)

Introducing `std::algorithm`

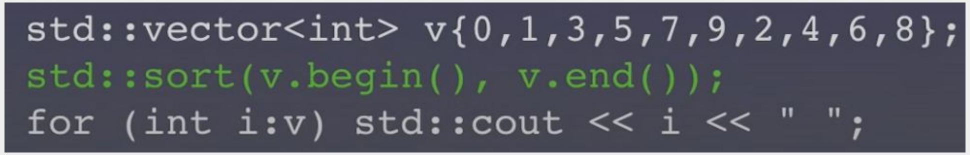
Spend more time thinking about which ‘building block to choose’ rather than what code to write from scratch.

Why use STL Algorithms

- Serve as general purpose building blocks for solving difficult problems
- Help make it easier to reason about and maintain your code
  - (i.e. avoiding raw loops)
- They're well tested and debugged
- Generally reduces your code size
  - See example below



```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
bool flag = true;
for (int i = 1; (i <= v.size()) && flag; i++) {
   flag = false;
   for (int j = 0; j < (v.size() -1); j++) {
      if (v[j+1] < v[j]) {
         std::swap(v[j], v[j+1]);
         flag = true;
      }
   }
}
for (int i:v) std::cout << i << " ";
```

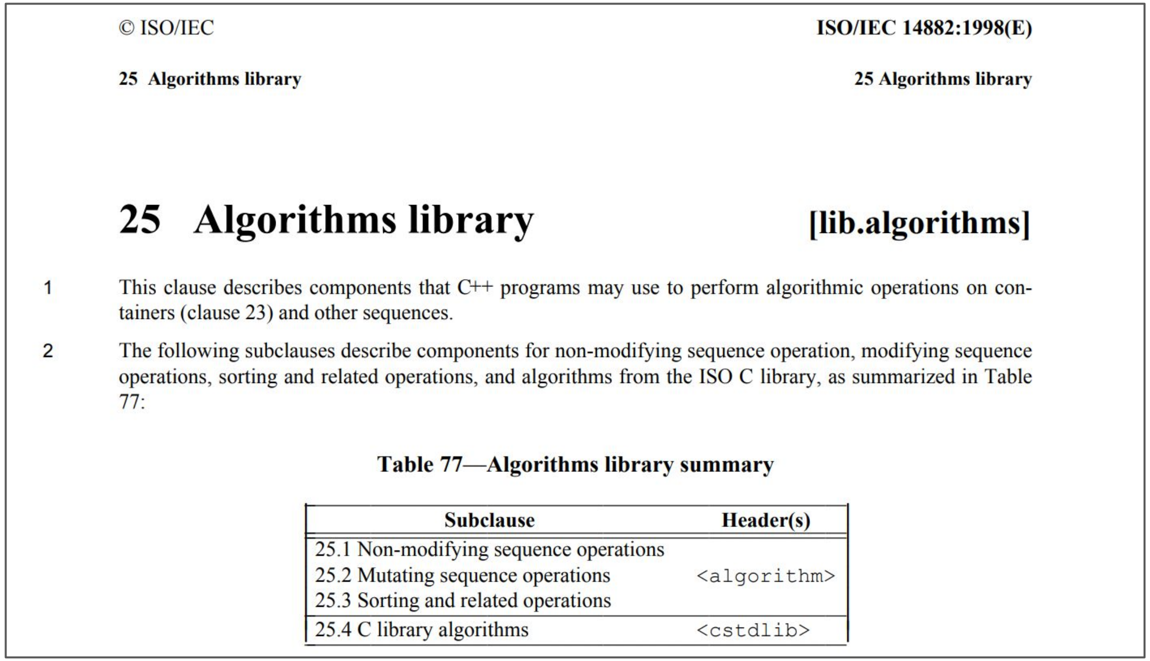


```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
std::sort(v.begin(), v.end());
for (int i:v) std::cout << i << " ";
```

Example from: CppCon 2016: Marshall Clow "STL Algorithms - why you should use them, and how to write your own"

Is `std::algorithm` something new in Modern C++?

- Not at all!
  - (Even more odd that I never really heard of or started using it until modern C++ era.)
- To the right is an image of the 1998 C++ standard with the Algorithms Library defined.



© ISO/IEC ISO/IEC 14882:1998(E)

25 Algorithms library 25 Algorithms library

25 Algorithms library [lib.algorithms]

1 This clause describes components that C++ programs may use to perform algorithmic operations on containers (clause 23) and other sequences.

2 The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 77:

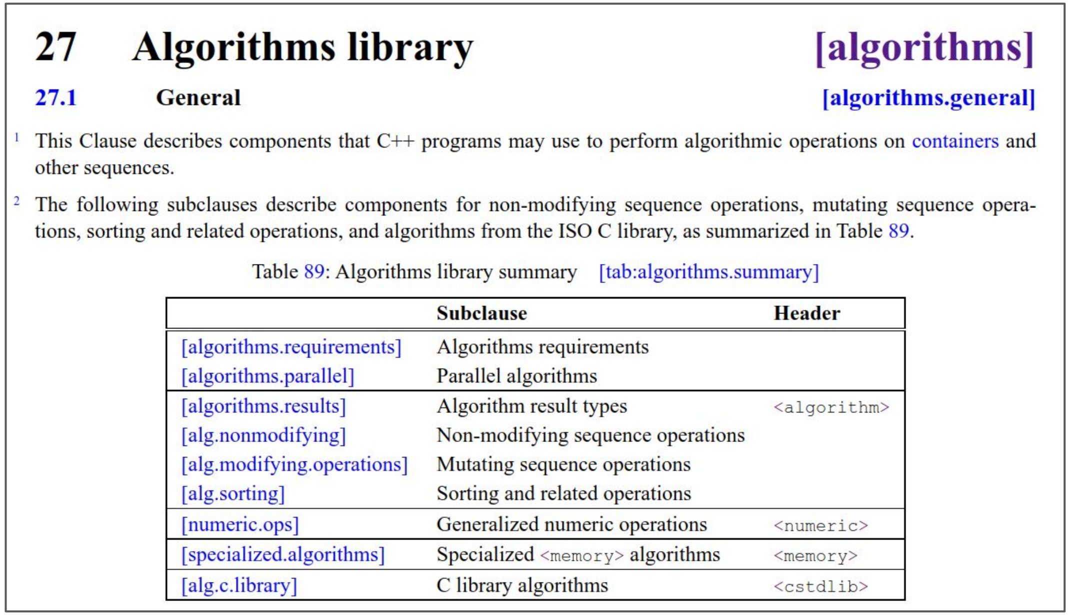
Table 77—Algorithms library summary

Subclause	Header(s)
25.1 Non-modifying sequence operations	
25.2 Mutating sequence operations	`<algorithm>`
25.3 Sorting and related operations	
25.4 C library algorithms	`<cstdlib>`

https://www.lirmm.fr/~ducour/Doc-objets/ISO+IEC+14882-1998.pdf

(Aside: Here's a draft of a recent C++ standard in 2022)

- The algorithms library still exists and is remains an increasingly important part of C++!



27 Algorithms library **[algorithms]**

27.1 General **[algorithms.general]**

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers and other sequences.

2 The following subclauses describe components for non-modifying sequence operations, mutating sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 89.

Table 89: Algorithms library summary [tab:algorithms.summary]

	Subclause	Header
[algorithms.requirements]	Algorithms requirements	
[algorithms.parallel]	Parallel algorithms	
[algorithms.results]	Algorithm result types	`<algorithm>`
[alg.nonmodifying]	Non-modifying sequence operations	
[alg.modifying.operations]	Mutating sequence operations	
[alg.sorting]	Sorting and related operations	
[numeric.ops]	Generalized numeric operations	`<numeric>`
[specialized.algorithms]	Specialized `<memory>` algorithms	`<memory>`
[alg.c.library]	C library algorithms	`<cstdlib>`

https://eel.is/c++draft/algorithms

(Aside: Here’s a draft of a recent C++ standard in 2022)

- The algorithms library still exists and is remains an important pa

27 Algorithms library [algorithms]

27.1 General [algorithms.general]

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers and

ations, mutating sequence opera-
narized in Table 89.

ummary]

		Header
		<algorithm>
[alg.modifying.operations]	Mutating sequence operations	
[alg.sorting]	Sorting and related operations	
[numeric.ops]	Generalized numeric operations	<numeric>
[specialized.algorithms]	Specialized <memory> algorithms	<memory>
[alg.c.library]	C library algorithms	<cstdlib>

Let’s dive in!

My goal being to show you some neat algorithms that are available.

https://eel.is/c++draft/algorithms

Average Top 3 Positive Numbers - Algorithm Version

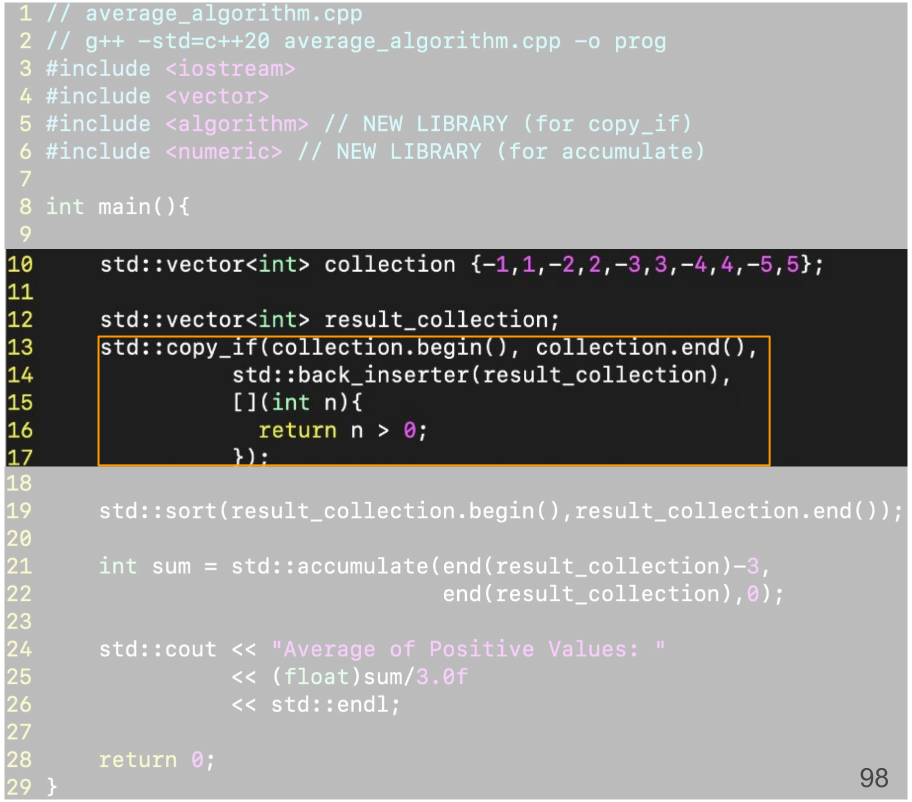
- Same problem--this time using the STL algorithm and numerics library

```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
              std::back_inserter(result_collection),
              [](int n){
                return n > 0;
              });

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

    int sum = std::accumulate(end(result_collection)-3,
                              end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

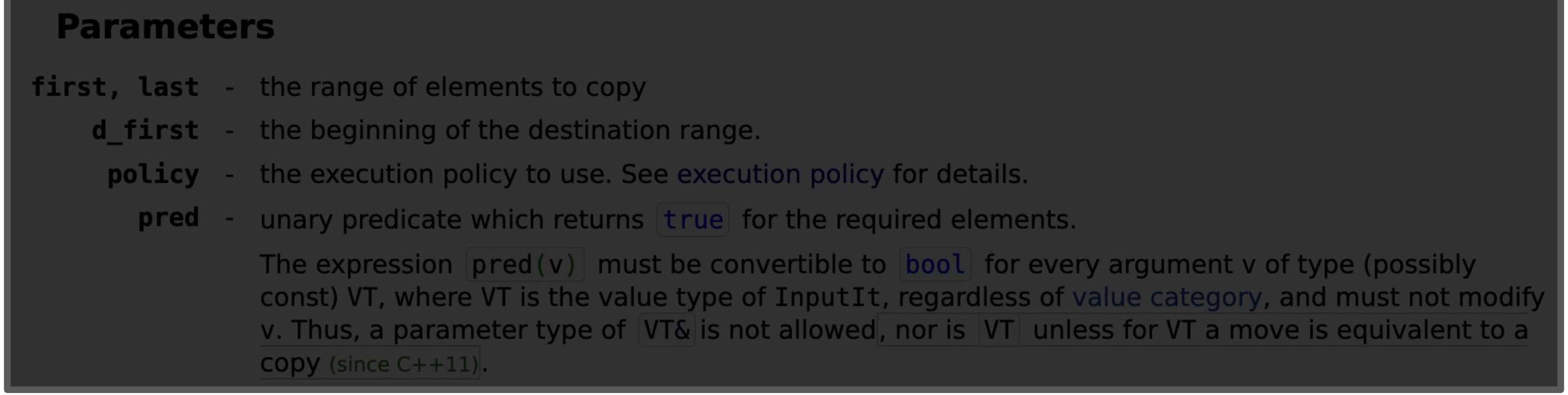
Average Top 3 Positive Numbers - Algorithm Version

- Same problem--this time using the STL algorithm and numerics library

- We use ‘`copy_if`’ which will copy elements from [start,end) into another collection if some predicate is true.
    - Let’s look closer at `copy_if` for a moment...



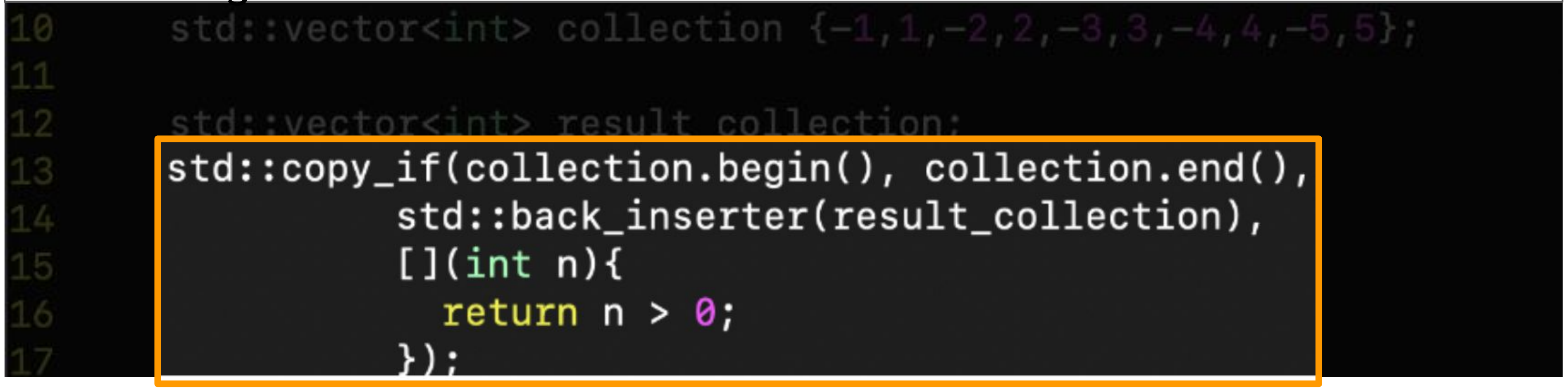
```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
              std::back_inserter(result_collection),
              [](int n){
                return n > 0;
              });

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

    int sum = std::accumulate(end(result_collection)-3,
                                  end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```



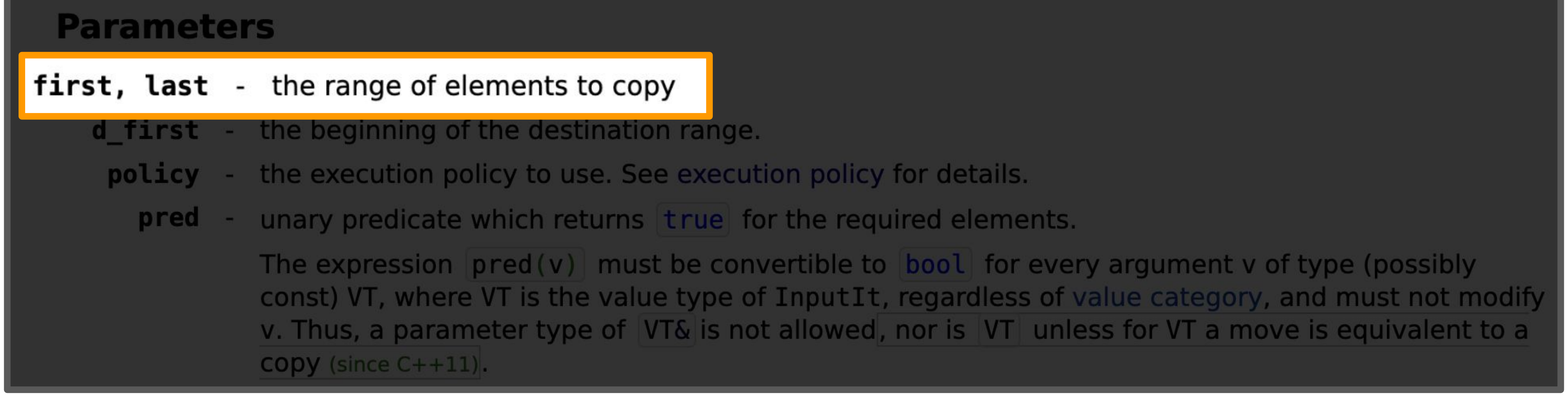
Parameters

`first, last`	-	the range of elements to copy
`d_first`	-	the beginning of the destination range.
`policy`	-	the execution policy to use. See execution policy for details.
`pred`	-	unary predicate which returns `true` for the required elements. The expression `pred(v)` must be convertible to `bool` for every argument v of type (possibly const) VT, where VT is the value type of InputIt, regardless of value category, and must not modify v. Thus, a parameter type of `VT&` is not allowed, nor is `VT` unless for VT a move is equivalent to a copy (since C++11).

Let's take a moment to understand the 'general form' of `std::algorithm` functions



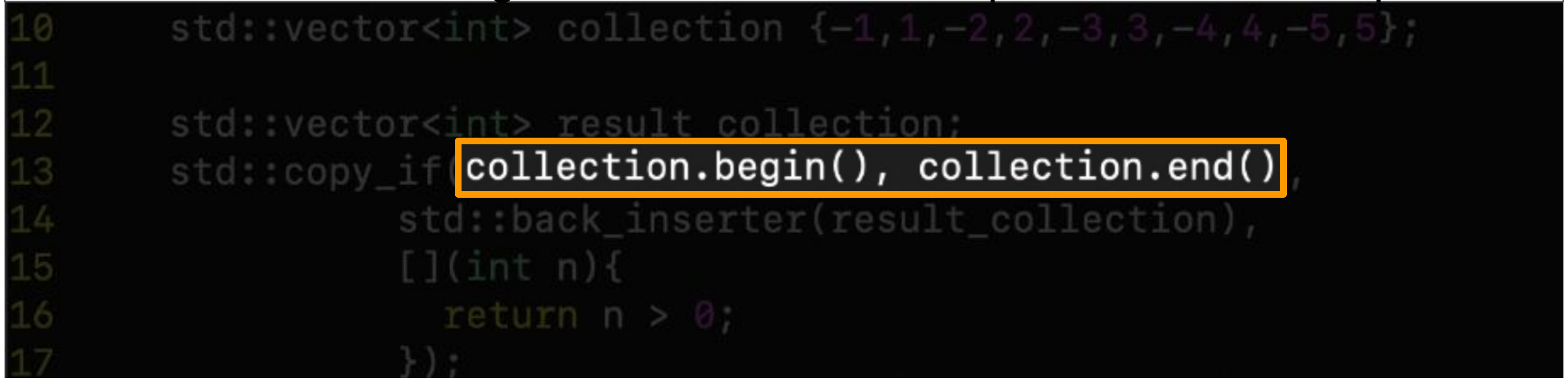
```cpp
    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });
```



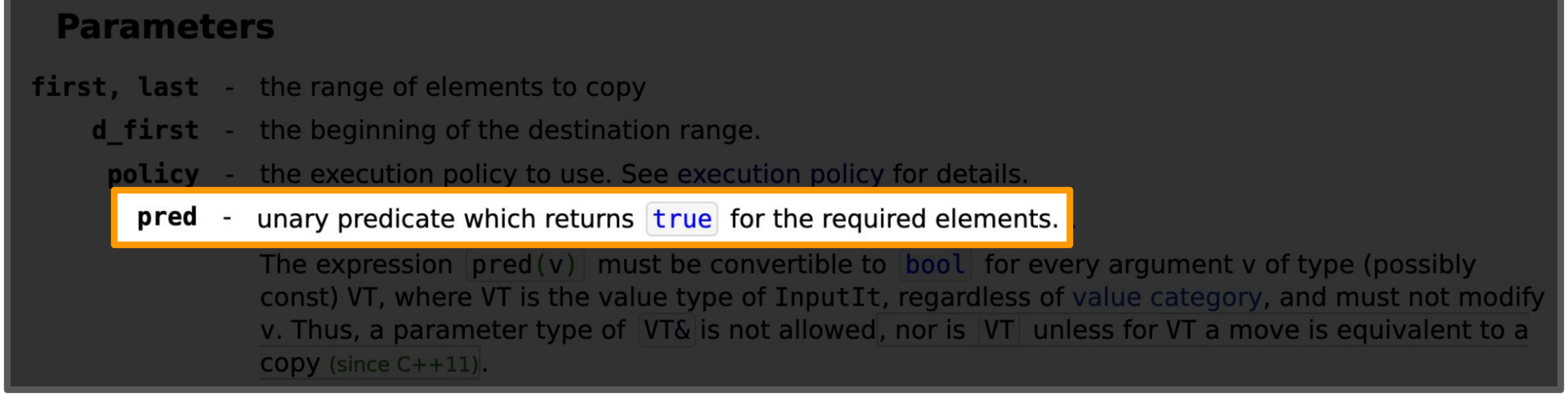
Parameters

first, last	-	the range of elements to copy
d_first	-	the beginning of the destination range.
policy	-	the execution policy to use. See execution policy for details.
pred	-	unary predicate which returns `true` for the required elements. The expression `pred(v)` must be convertible to `bool` for every argument v of type (possibly const) VT, where VT is the value type of InputIt, regardless of value category, and must not modify v. Thus, a parameter type of `VT&` is not allowed, nor is `VT` unless for VT a move is equivalent to a copy (since C++11).

First we have the range of elements we want to copy -- remember iterators tell us our 'range' of where we want to perform some computation



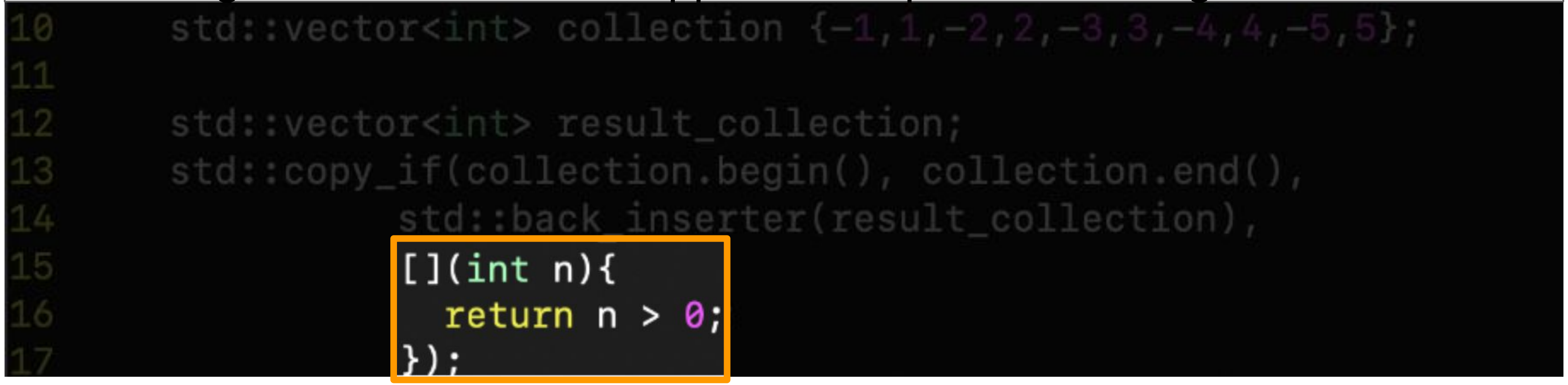
```
    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
            std::back_inserter(result_collection),
            [](int n){
              return n > 0;
            });
```



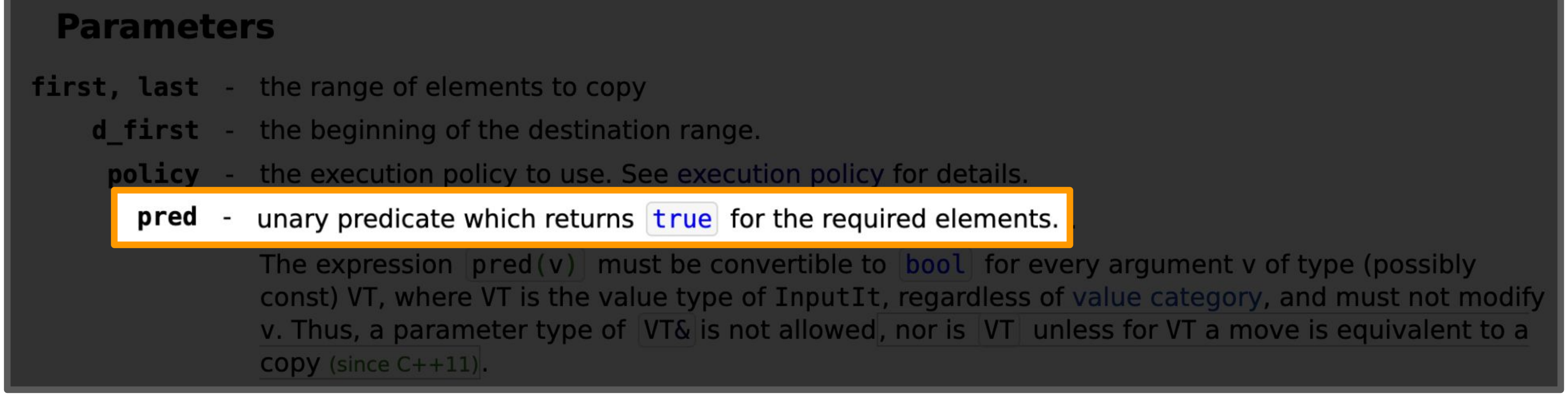
Parameters

`first, last`	-	the range of elements to copy
`d_first`	-	the beginning of the destination range.
`policy`	-	the execution policy to use. See execution policy for details.
`pred`	-	unary predicate which returns `true` for the required elements. The expression `pred(v)` must be convertible to `bool` for every argument v of type (possibly const) VT, where VT is the value type of InputIt, regardless of value category, and must not modify v. Thus, a parameter type of `VT&` is not allowed, nor is `VT` unless for VT a move is equivalent to a copy (since C++11).

The next most common part is our ‘predicate’.
Something that if returns true, applies the operation on a given element.



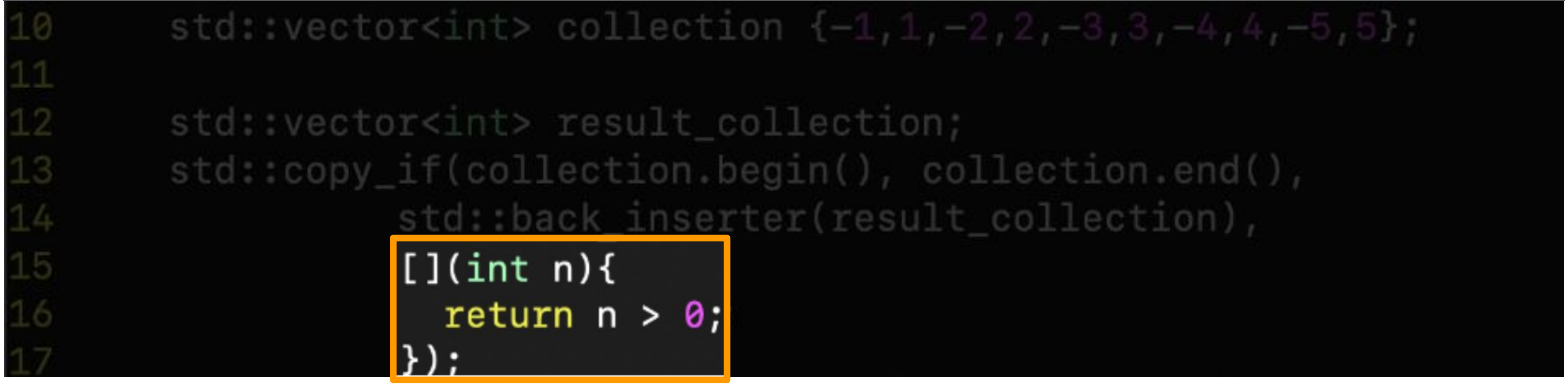
```
    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
            std::back_inserter(result_collection),
            [](int n){
              return n > 0;
            });
```



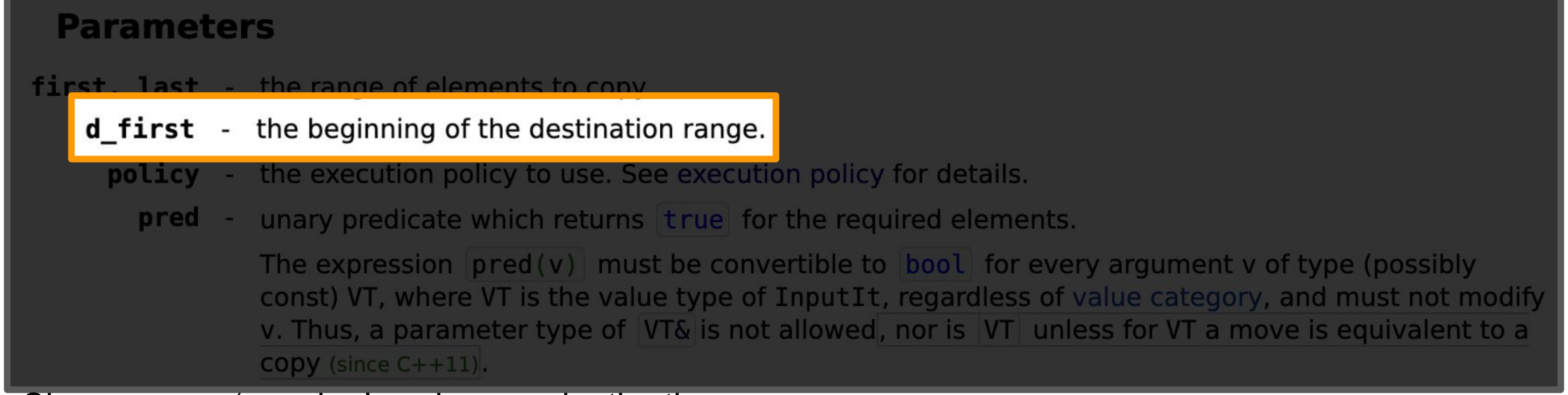
Parameters

`first, last`	-	the range of elements to copy
`d_first`	-	the beginning of the destination range.
`policy`	-	the execution policy to use. See execution policy for details.
`pred`	-	unary predicate which returns `true` for the required elements. The expression `pred(v)` must be convertible to `bool` for every argument v of type (possibly const) VT, where VT is the value type of InputIt, regardless of value category, and must not modify v. Thus, a parameter type of `VT&` is not allowed, nor is `VT` unless for VT a move is equivalent to a copy (since C++11).

Frequently our predicates will be lambda functions.



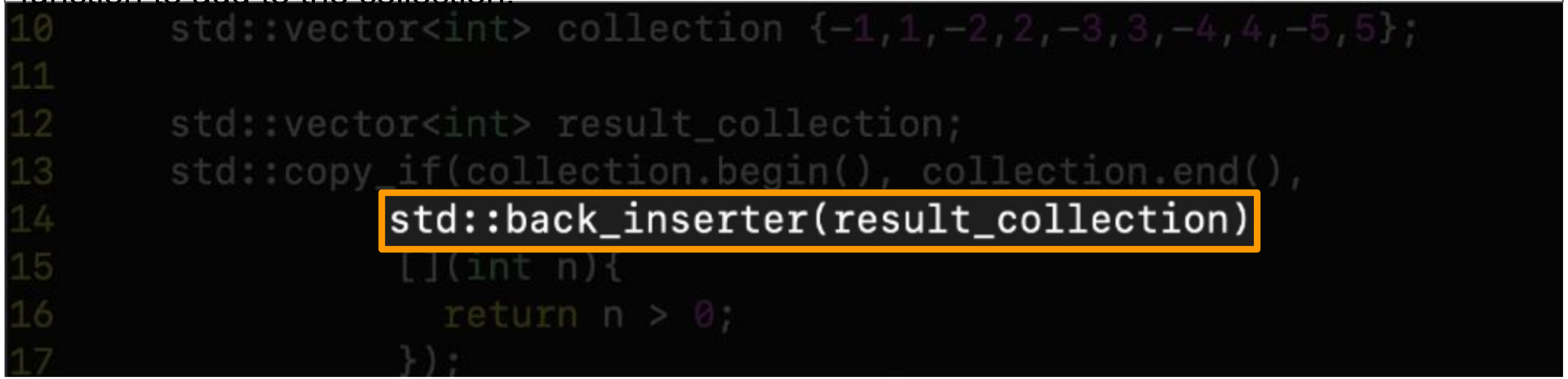
```
std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

std::vector<int> result_collection;
std::copy_if(collection.begin(), collection.end(),
             std::back_inserter(result_collection),
             [](int n){
               return n > 0;
             });
```



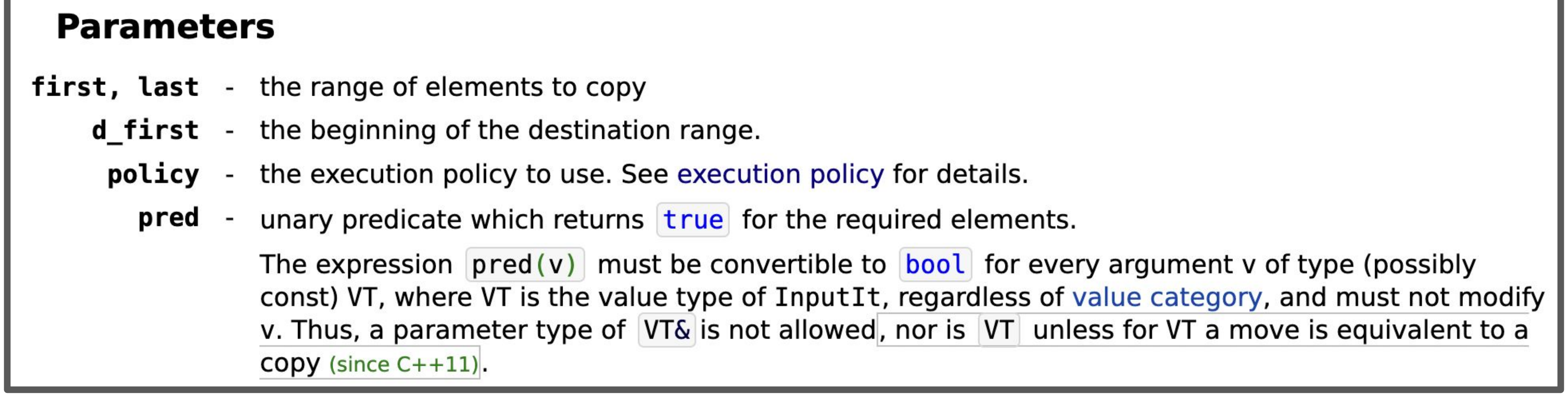
Parameters

`first, last`	-	the range of elements to copy
`d_first`	-	the beginning of the destination range.
`policy`	-	the execution policy to use. See execution policy for details.
`pred`	-	unary predicate which returns `true` for the required elements. The expression `pred(v)` must be convertible to `bool` for every argument v of type (possibly const) VT, where VT is the value type of InputIt, regardless of value category, and must not modify v. Thus, a parameter type of `VT&` is not allowed, nor is `VT` unless for VT a move is equivalent to a copy (since C++11).

Since we are 'copying' we have a destination range.
back_inserter is an iterator adaptor that can be used with containers that have a push_back function to add to the collection.



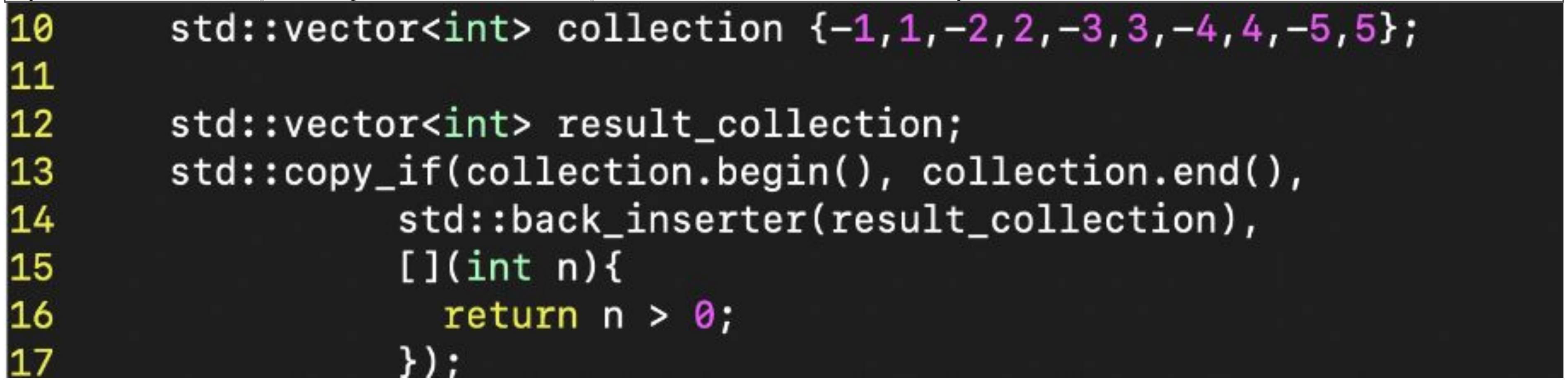
```
    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection)
                 [](int n){
                   return n > 0;
                 });
```



Parameters

first, last	-	the range of elements to copy
d_first	-	the beginning of the destination range.
policy	-	the execution policy to use. See execution policy for details.
pred	-	unary predicate which returns `true` for the required elements. The expression `pred(v)` must be convertible to `bool` for every argument v of type (possibly const) VT, where VT is the value type of `InputIt`, regardless of value category, and must not modify v. Thus, a parameter type of `VT&` is not allowed, nor is `VT` unless for VT a move is equivalent to a copy (since C++11).

So here's the `copy_if` documentation and code in one place.
(I'll mention policy later -- it's part of an overload)

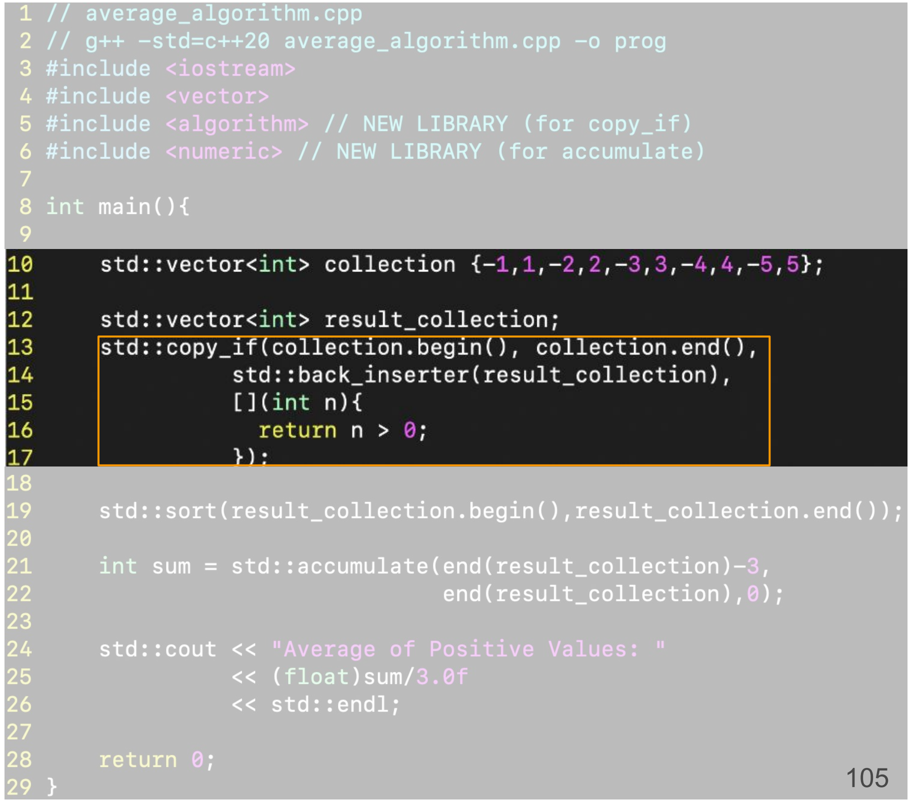


```
    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });
```

Average Top 3 Positive Numbers - Algorithm Version

- Same problem--this time using the STL algorithm and numerics library

- We now have copied all integers greater than zero into a new collection
  - Look--no raw for-loops needed!
- (Next part...)

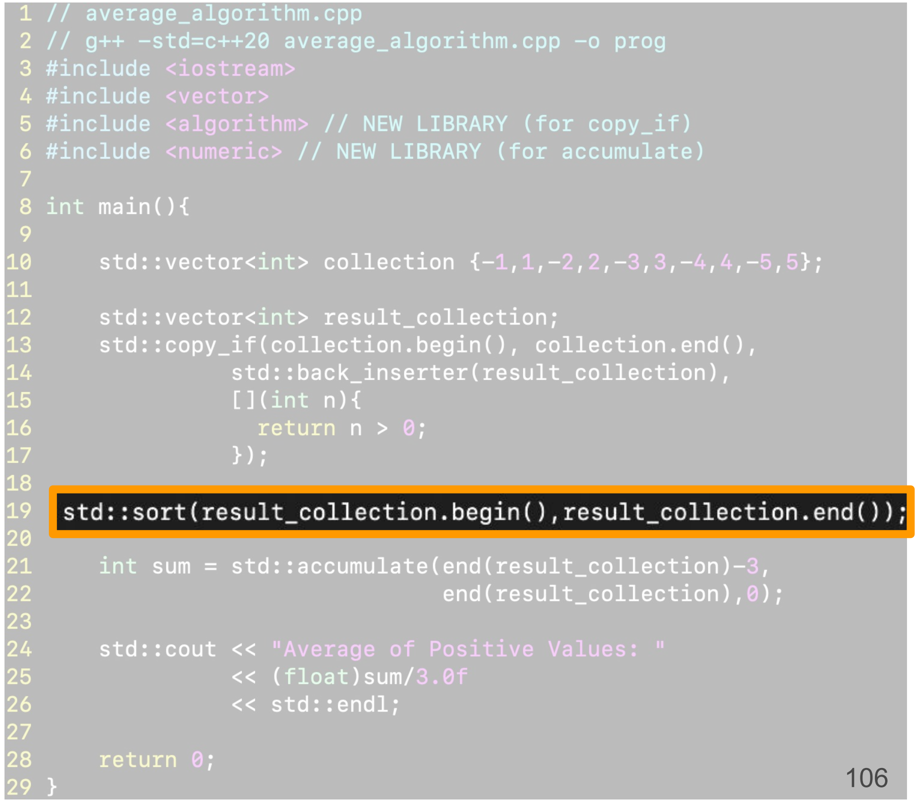


```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
              std::back_inserter(result_collection),
              [](int n){
                return n > 0;
              });

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

    int sum = std::accumulate(end(result_collection)-3,
                              end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Average Top 3 Positive Numbers - Algorithm Version

- Same problem--this time using the STL algorithm and numerics library

- std::sort is somewhat explanatory
  - We sort from the start to the end of a range
  - And we actually get an O(nlog$_2$n) sorting algorithm
    - (Better than my ad-hoc insertion sort!)

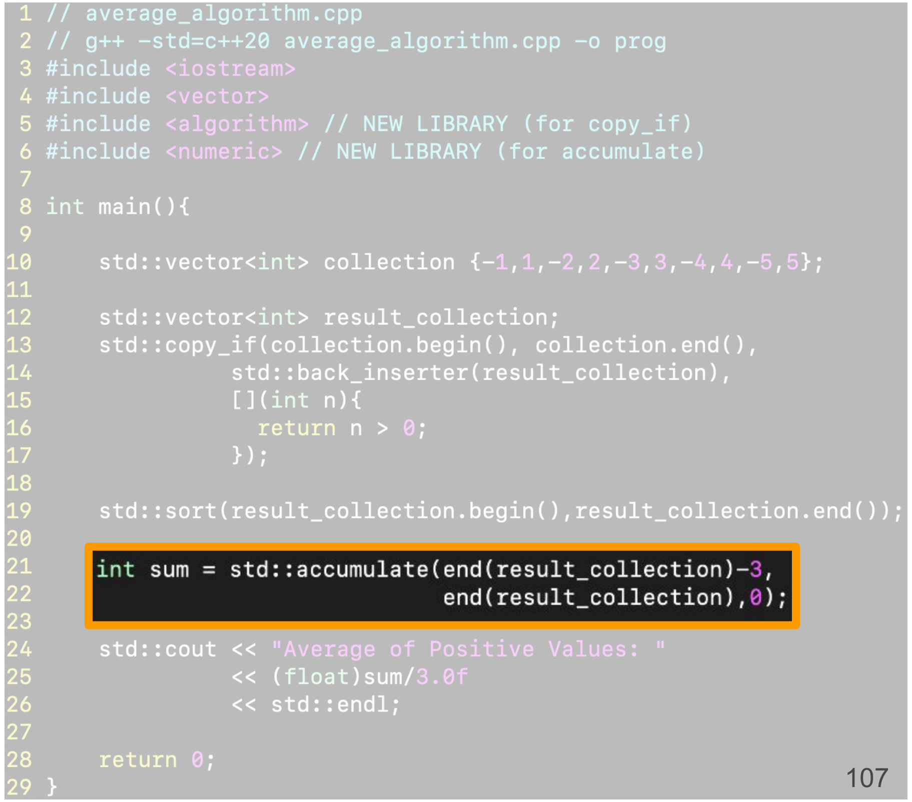


```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

  std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

  std::vector<int> result_collection;
  std::copy_if(collection.begin(), collection.end(),
       std::back_inserter(result_collection),
       [](int n){
        return n > 0;
       });

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

  int sum = std::accumulate(end(result_collection)-3,
               end(result_collection),0);

  std::cout << "Average of Positive Values: "
       << (float)sum/3.0f
       << std::endl;

  return 0;
}
```

Average Top 3 Positive Numbers - Algorithm Version

- Same problem--this time using the STL algorithm and numerics library

- std::accumulate takes a range (start and end iterator) and sums up their values
  - Note: our starting iterator is -3 from the end of our sorted collection
    - Thus, the last three elements are added.
    - (Kind of neat to play with iterators!)

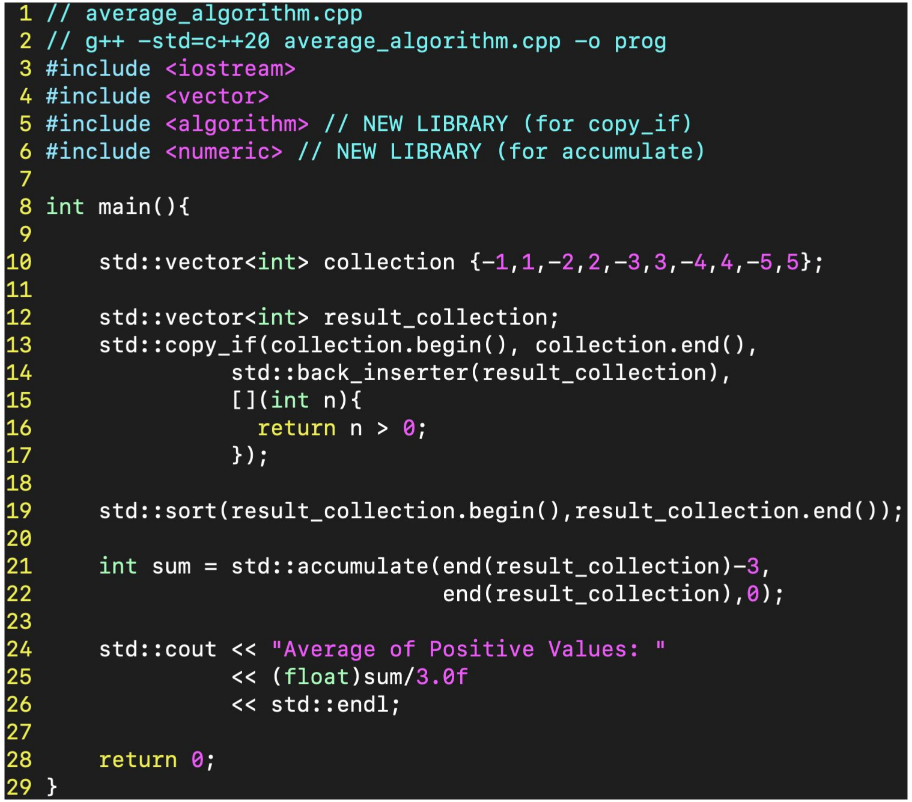


```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });

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

    int sum = std::accumulate(end(result_collection)-3,
                               end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Average Top 3 Positive Numbers - Algorithm Version

- Same problem--this time using the STL algorithm and numerics library

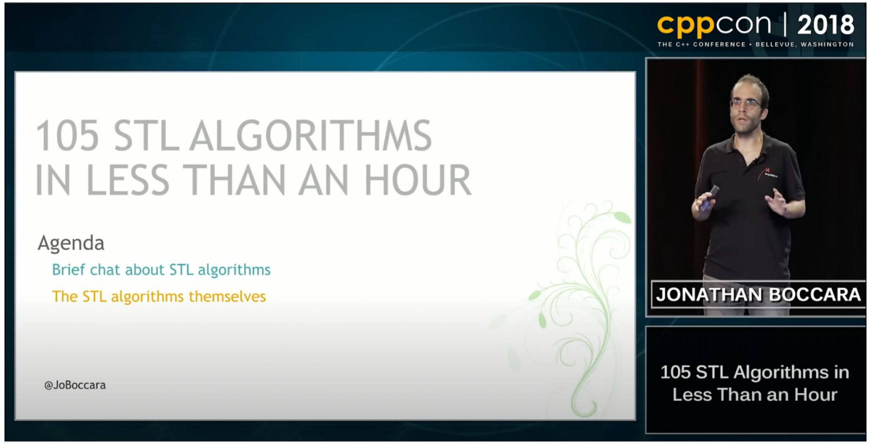
- Here’s the full program
  - (And it works!)
- And what’s important, is how we thought about our operations:
  - What range to copy
  - What range to sort
  - What range to accumulate
- Less thought on small details like our first two examples (vector part 1 & 2)



```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });

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

    int sum = std::accumulate(end(result_collection)-3,
                                    end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Building Blocks and Rapid Fire of Small Examples

My Goal at this point is to just show you what is available -- Hopefully you're motivated now!


cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
105 STL ALGORITHMS
IN LESS THAN AN HOUR
Agenda
Brief chat about STL algorithms
The STL algorithms themselves
@JoBoccara
JONATHAN BOCCARA
105 STL Algorithms in Less Than an Hour

https://www.youtube.com/watch?v=2olsGf6JIkU - For a more full coverage on nearly every STL algorithm start here!

High Level Overview of std::algorithm Building Blocks [cppref]

- Non-modifying sequence operations
- Modifying sequence operations
- Partitioning operations
- Sorting operations
- Binary search operations (on sorted ranges)
- Other operations on sorted ranges
- Set operations (on sorted ranges)
- Heap operations
- Minimum/maximum operations
- Comparison operations
- Permutation operations
- Numeric operations
- Operations on uninitialized memory
- C library

Algorithms library

The algorithms library defines functions for a va
operate on ranges of elements. Note that a rang
the last element to inspect or modify.

High Level Overview of std::algorithm Building Blocks [cppref]

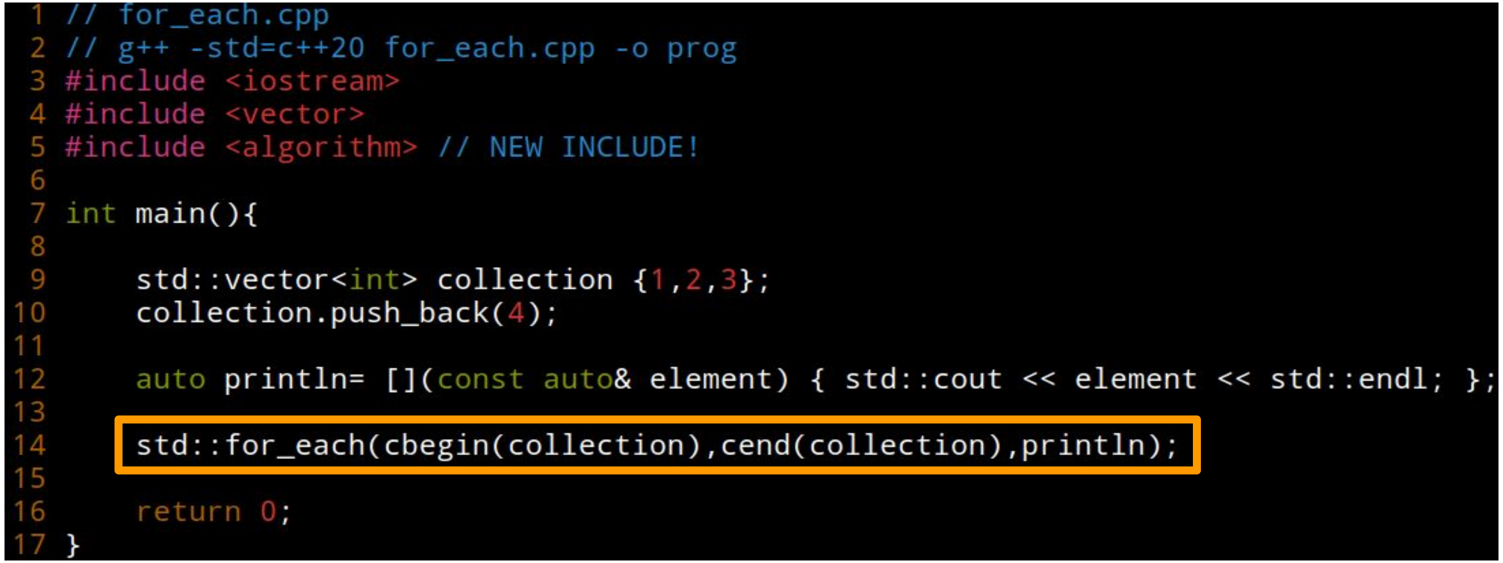
- **Non-modifying sequence operations**
- **Modifying sequence operations**
- **Partitioning operations**
- **Sorting operations**
- Binary search operations (on sorted ranges)
- Other operations on sorted ranges
- Set operations (on sorted ranges)
- Heap operations
- Minimum/maximum operations
- Comparison operations
- Permutation operations
- **Numeric operations**
- Operations on uninitialized memory
- C library

Algorithms library

The algorithms library defines functions for a va
operate on ranges of elements. Note that a rang
the last element to inspect or modify.

for_each applies a function to a range of elements (function template)

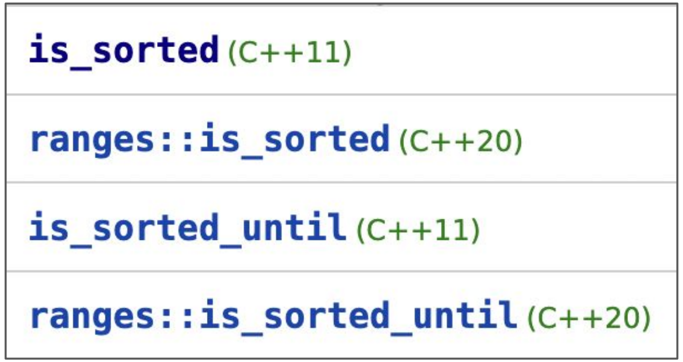
- Pretty much identical to our ranged-based loop, but for_each (or for_each_n) pushes the level of abstraction one layer further.
- This time using iterators and applies a lambda function (‘println’) to each element in the `std::vector`.
  - Note: This time using const_iterators to enforce const correctness.



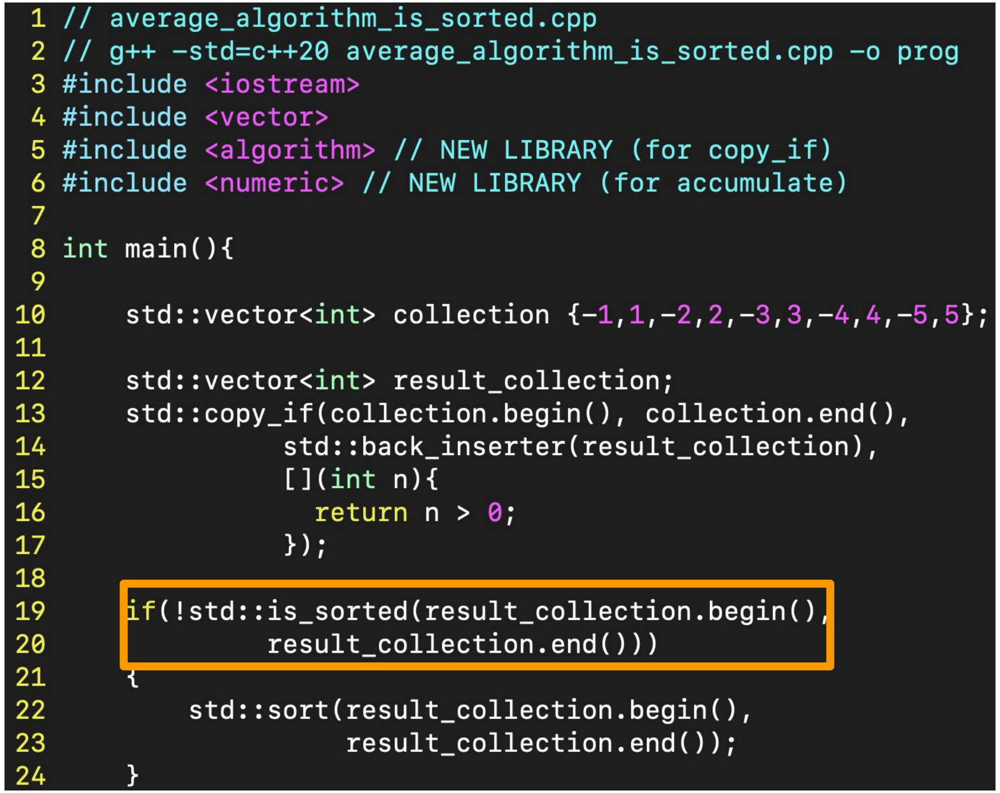
```cpp
// for_each.cpp
// g++ -std=c++20 for_each.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW INCLUDE!

int main(){

    std::vector<int> collection {1,2,3};
    collection.push_back(4);

    auto println= [](const auto& element) { std::cout << element << std::endl; };

    std::for_each(cbegin(collection),cend(collection),println);

    return 0;
}
```

Sorting Operations -- `is_sorted`

- Check if a collection is sorted prior to performing a sort routine.
  - (Small improvement on our previous example of computing averages)
  - Other variations exist



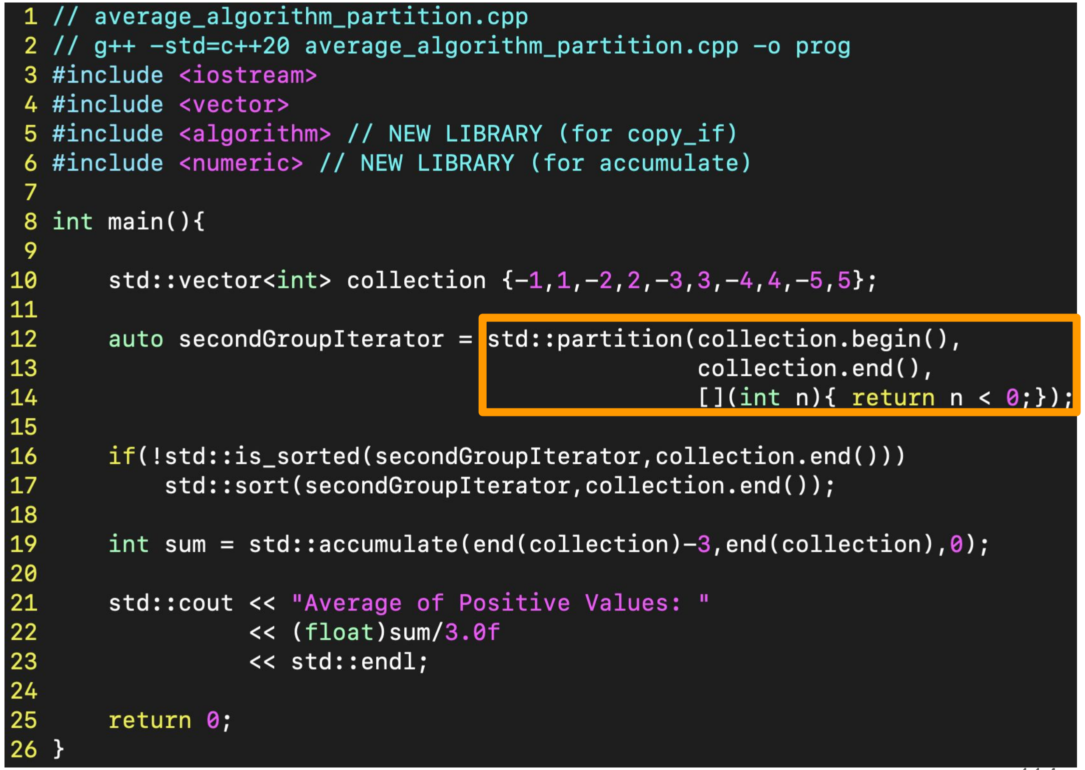
is_sorted (C++11)
ranges::is_sorted (C++20)
is_sorted_until (C++11)
ranges::is_sorted_until (C++20)



```
// average_algorithm_is_sorted.cpp
// g++ -std=c++20 average_algorithm_is_sorted.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });

    if(!std::is_sorted(result_collection.begin(),
                result_collection.end()))
    {
        std::sort(result_collection.begin(),
                  result_collection.end());
    }
```

Partitioning Operations -- `partition` (or `stable_partition`)

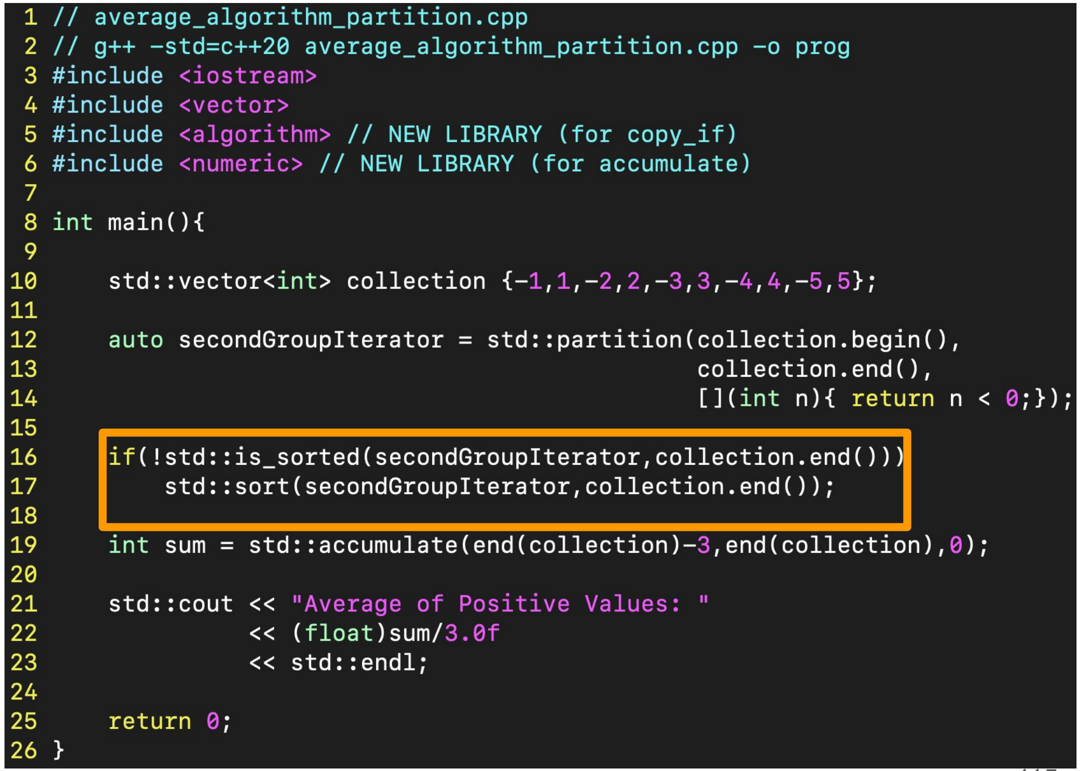
- This time partition all of the negative numbers in a first group, and positive numbers in a second group
  - std::partition will return an iterator to the second group
- (next slide)



```cpp
// average_algorithm_partition.cpp
// g++ -std=c++20 average_algorithm_partition.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    auto secondGroupIterator = std::partition(collection.begin(),
                                              collection.end(),
                                              [](int n){ return n < 0;});

    if(!std::is_sorted(secondGroupIterator,collection.end()))
        std::sort(secondGroupIterator,collection.end());

    int sum = std::accumulate(end(collection)-3,end(collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Partitioning Operations -- `partition` (or `stable_partition`)

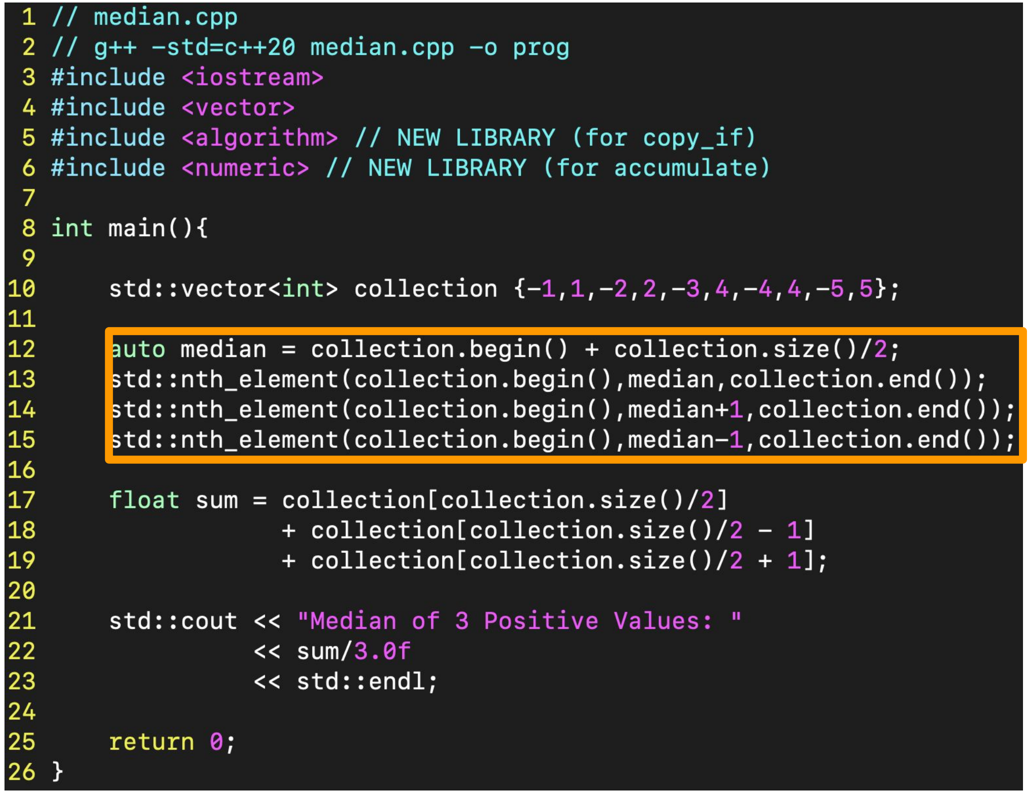
- This time partition all of the negative numbers in a first group, and positive numbers in a second group
  - std::partition will return an iterator to the second group
- Then sort only the positive numbers in our second group
  - (and proceed to accumulate and take average of top 3 values)



```cpp
// average_algorithm_partition.cpp
// g++ -std=c++20 average_algorithm_partition.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    auto secondGroupIterator = std::partition(collection.begin(),
                                              collection.end(),
                                              [](int n){ return n < 0;});

    if(!std::is_sorted(secondGroupIterator,collection.end()))
        std::sort(secondGroupIterator,collection.end());

    int sum = std::accumulate(end(collection)-3,end(collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Partitioning Operations -- `nth_element`

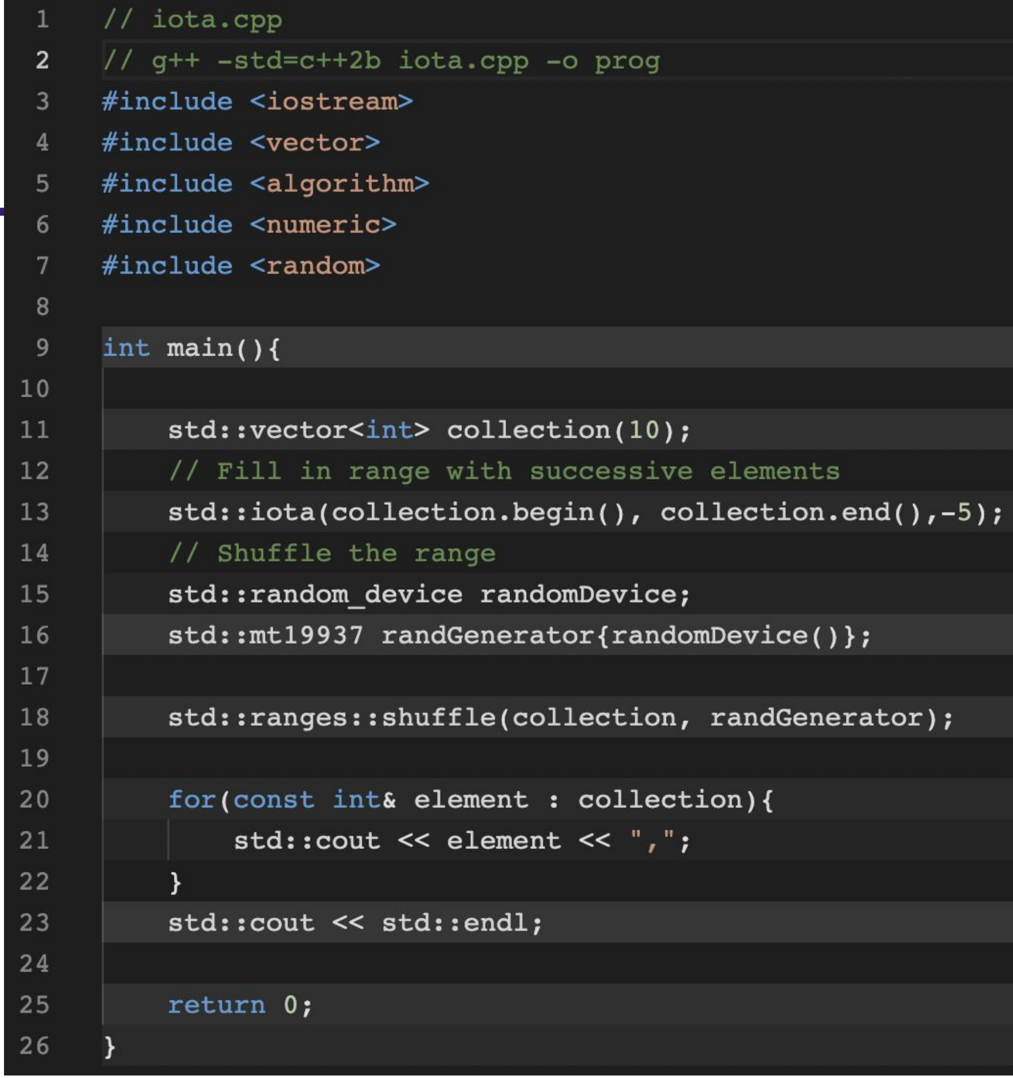
- This time we find the average of 3 median values.
    - e.g. Median filter for noise reduction in image processing
- `nth_element` partitions at the nth element putting smaller values in front of the value.



```cpp
// median.cpp
// g++ -std=c++20 median.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,4,-4,4,-5,5};

    auto median = collection.begin() + collection.size()/2;
    std::nth_element(collection.begin(),median,collection.end());
    std::nth_element(collection.begin(),median+1,collection.end());
    std::nth_element(collection.begin(),median-1,collection.end());

    float sum = collection[collection.size()/2]
            + collection[collection.size()/2 - 1]
            + collection[collection.size()/2 + 1];

    std::cout << "Median of 3 Positive Values: "
              << sum/3.0f
              << std::endl;

    return 0;
}
```

Numeric Operation -- `iota`

- Fill a range with successive elements (line 13)
  - (Could also do something similar with `generate`)
- Then we ‘shuffle’ the collection to get a set of random numbers (line 15-18)
- Note:
  - This uses something new called ‘ranges’ in C++20
  - Try here: https://godbolt.org/z/cbrsx35j5



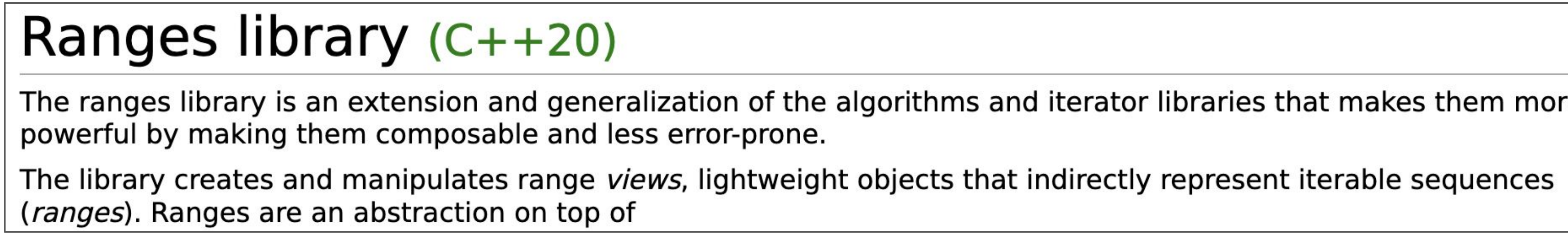
```
 1  // iota.cpp
 2  // g++ -std=c++2b iota.cpp -o prog
 3  #include <iostream>
 4  #include <vector>
 5  #include <algorithm>
 6  #include <numeric>
 7  #include <random>
 8
 9  int main(){
10
11      std::vector<int> collection(10);
12      // Fill in range with successive elements
13      std::iota(collection.begin(), collection.end(),-5);
14      // Shuffle the range
15      std::random_device randomDevice;
16      std::mt19937 randGenerator{randomDevice()};
17
18      std::ranges::shuffle(collection, randGenerator);
19
20      for(const int& element : collection){
21          std::cout << element << ",";
22      }
23      std::cout << std::endl;
24
25      return 0;
26  }
```

C++ 20 ranges and views

Brief introduction to Ranges and Views

C++ 20 Ranges [cppref]

- In short, ranges in C++ build off (most all) the std::algorithm functions.
  - Algorithms operate directly on the container
  - Composition with the ‘|’ operator
  - Lazy evaluation



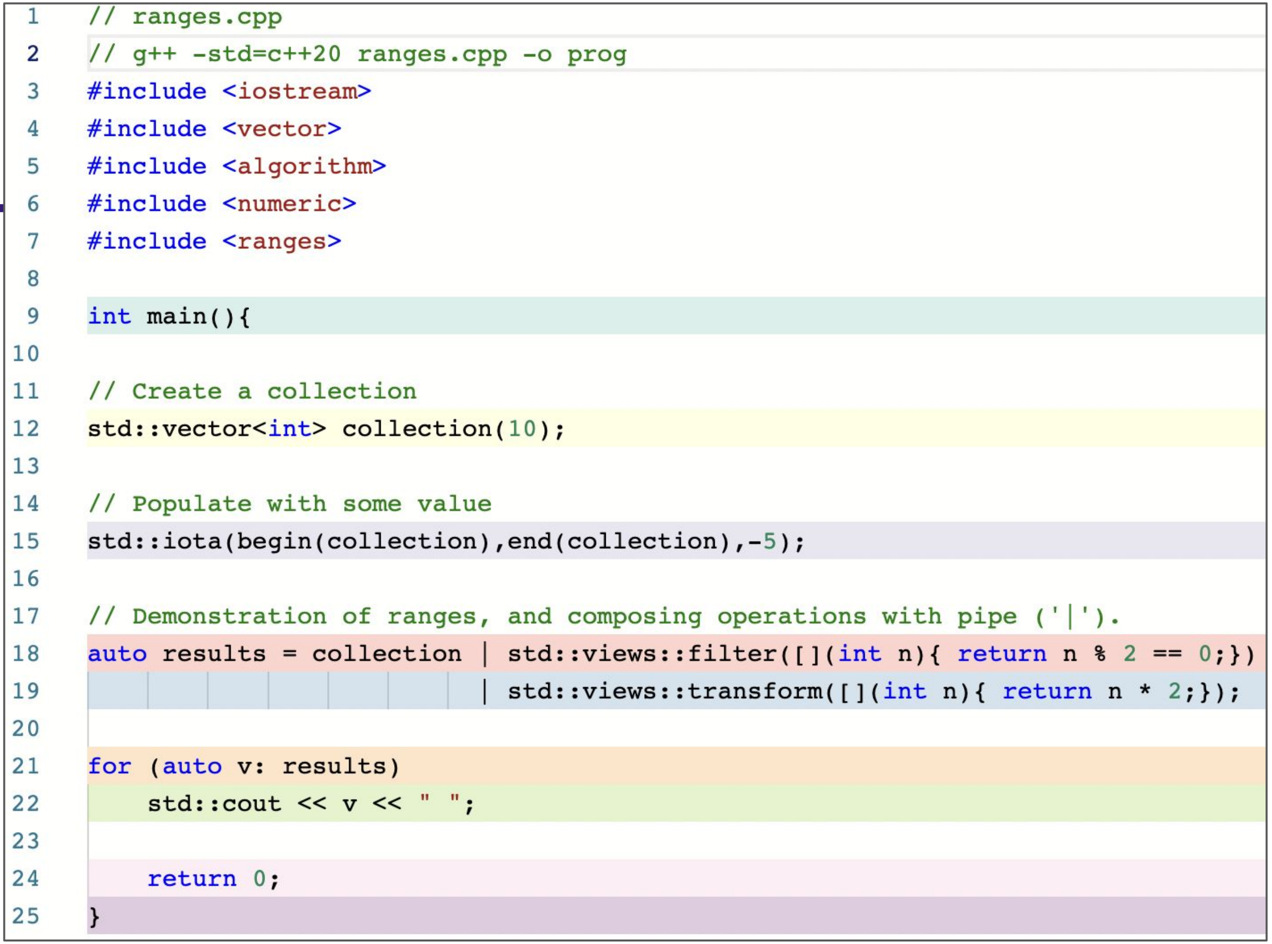
Ranges library (C++20)

The ranges library is an extension and generalization of the algorithms and iterator libraries that makes them mor
powerful by making them composable and less error-prone.

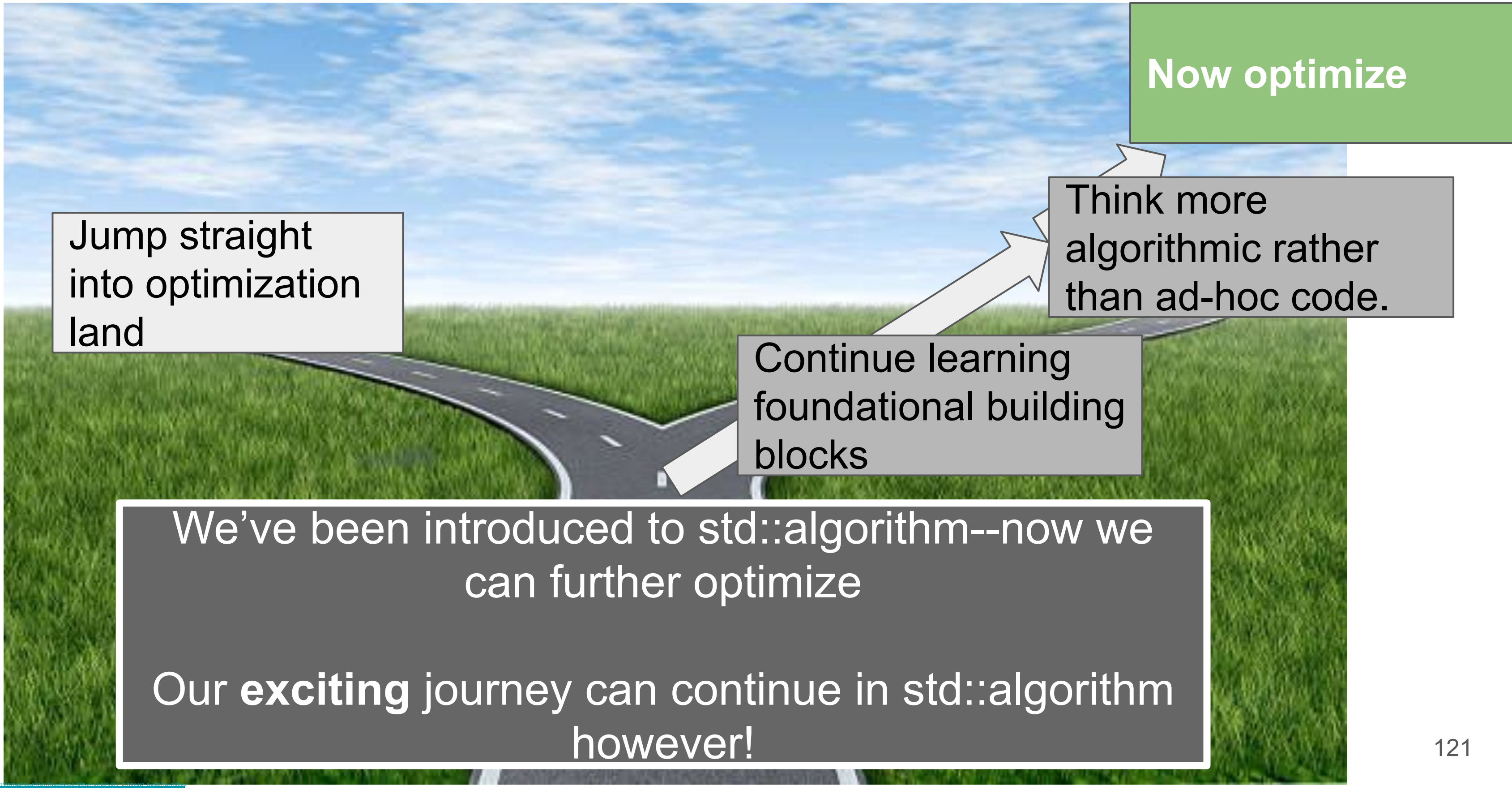
The library creates and manipulates range *views*, lightweight objects that indirectly represent iterable sequences (*ranges*). Ranges are an abstraction on top of

Ranges - Example

- Draw your attention to lines 18-20
- Try it!
  - https://godbolt.org/z/fn5e38f7b



```
 1  // ranges.cpp
 2  // g++ -std=c++20 ranges.cpp -o prog
 3  #include <iostream>
 4  #include <vector>
 5  #include <algorithm>
 6  #include <numeric>
 7  #include <ranges>
 8
 9  int main(){
10
11  // Create a collection
12  std::vector<int> collection(10);
13
14  // Populate with some value
15  std::iota(begin(collection),end(collection),-5);
16
17  // Demonstration of ranges, and composing operations with pipe ('|').
18  auto results = collection | std::views::filter([](int n){ return n % 2 == 0;})
19                            | std::views::transform([](int n){ return n * 2;});
20
21  for (auto v: results)
22      std::cout << v << " ";
23
24      return 0;
25  }
```



Now optimize

Think more algorithmic rather than ad-hoc code.

Jump straight into optimization land

Continue learning foundational building blocks

We've been introduced to std::algorithm--now we can further optimize

Our **exciting** journey can continue in std::algorithm however!

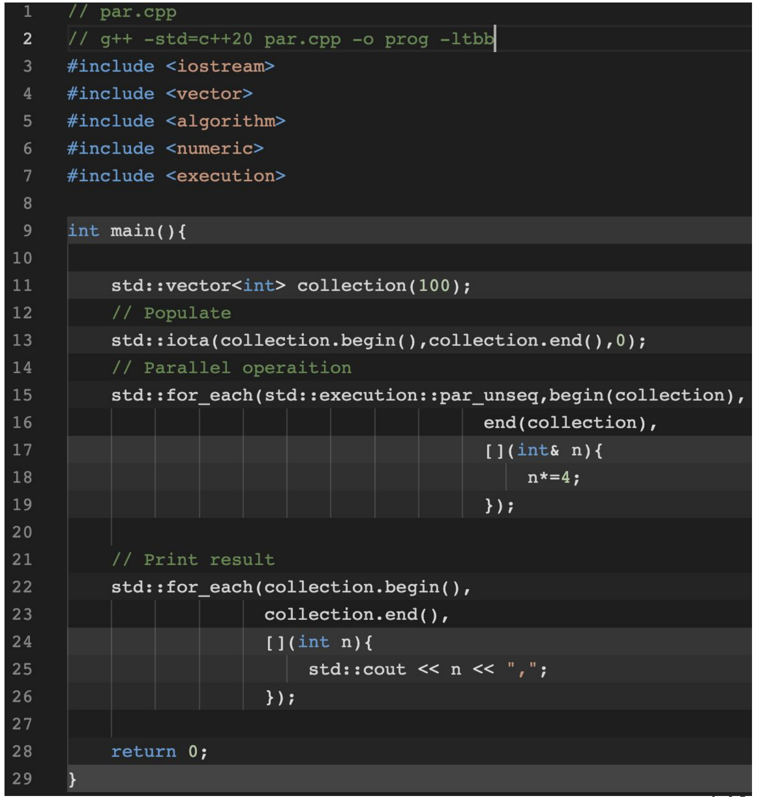
https://media.istockphoto.com/photos/cross-roads-horizon-picture-id1355264492?k=20&m=13552644[illegible]

Performance with std::algorithm

(And opening your code up for parallelism)

Measuring Performance of `std::algorithm`

- The reality is this isn't the right talk to talk about measuring performance.
- That said -- most `std::algorithm` have an overload for 'execution policy'
  - This execution policy can be sequential, or parallel for instance
  - This means there are opportunities to more easily parallelize your code using `std::algorithm`
    - A more complete introduction by Bryce [cppcon 2021]
  - More opportunities may be spotted for asynchronous programming as well in my experience using std::algorithm
- Example: https://godbolt.org/z/TeW9T8jMs



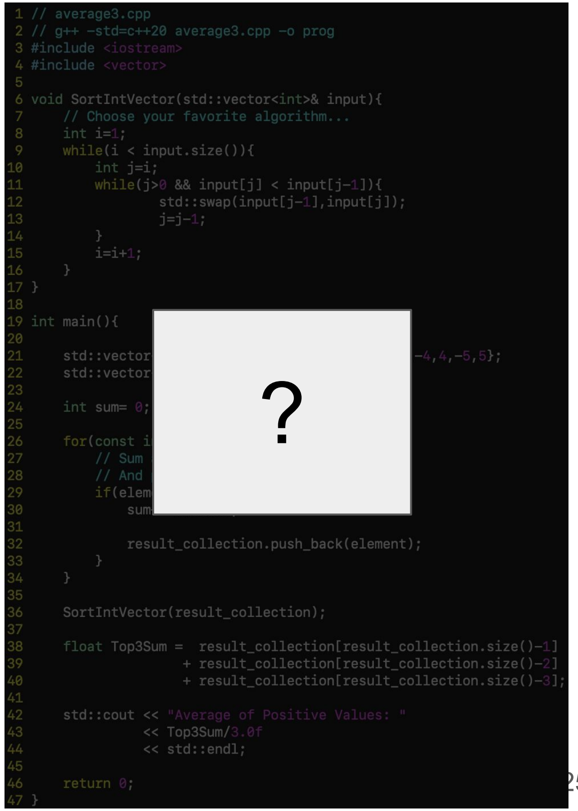
```cpp
// par.cpp
// g++ -std=c++20 par.cpp -o prog -ltbb
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <execution>

int main(){

    std::vector<int> collection(100);
    // Populate
    std::iota(collection.begin(),collection.end(),0);
    // Parallel operaition
    std::for_each(std::execution::par_unseq,begin(collection),
                                end(collection),
                                [](int& n){
                                    n*=4;
                                });

    // Print result
    std::for_each(collection.begin(),
                  collection.end(),
                  [](int n){
                      std::cout << n << ",";
                  });

    return 0;
}
```

Bonus Section

(If Time Allows)

Did you notice the error in one of my code examples?

- I left it in, because after hours of preparing these slides, I thought it was fitting for a talk motivating `std::algorithm`

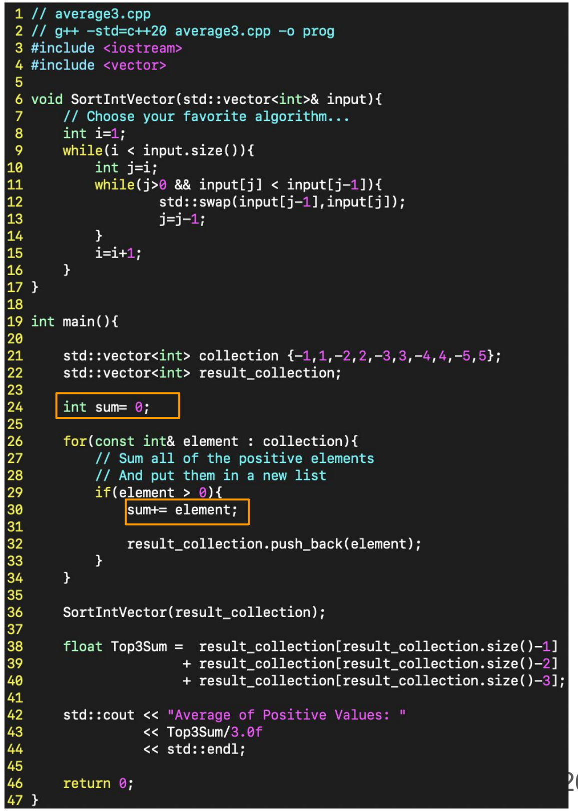


```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector                              -4,4,-5,5};
    std::vector

    int sum= 0;

    for(const i
        // Sum
        // And
        if(elem
            sum

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

?

Did you notice the error in one of my code examples?

- I left it in, because after hours of preparing these slides, I thought it was fitting for a talk motivating `std::algorithm`
    - Yup--some leftover unused variables during refactoring!
    - Static analysis might've picked this up, but it happens during code refactorings!

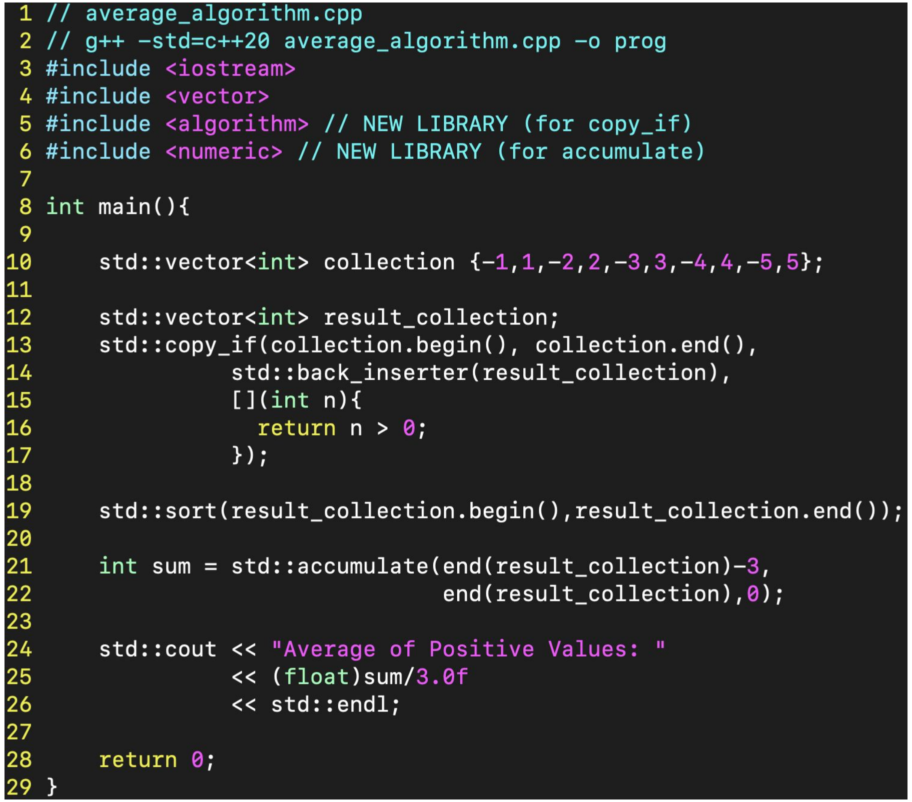


```cpp
// average3.cpp
// g++ -std=c++20 average3.cpp -o prog
#include <iostream>
#include <vector>

void SortIntVector(std::vector<int>& input){
    // Choose your favorite algorithm...
    int i=1;
    while(i < input.size()){
        int j=i;
        while(j>0 && input[j] < input[j-1]){
                std::swap(input[j-1],input[j]);
                j=j-1;
        }
        i=i+1;
    }
}

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};
    std::vector<int> result_collection;

    int sum= 0;

    for(const int& element : collection){
        // Sum all of the positive elements
        // And put them in a new list
        if(element > 0){
            sum+= element;

            result_collection.push_back(element);
        }
    }

    SortIntVector(result_collection);

    float Top3Sum =  result_collection[result_collection.size()-1]
                   + result_collection[result_collection.size()-2]
                   + result_collection[result_collection.size()-3];

    std::cout << "Average of Positive Values: "
              << Top3Sum/3.0f
              << std::endl;

    return 0;
}
```

(Audience thoughts?) std::algorithm - Code Review

- More precise
  - ??
- More resilient to bugs
  - ??
- More performant
  - ??
- Easier to maintain/reason about
  - ??

- Note:
  - For online/future listeners--how many other ways did you find to implement this?
  - How efficient can you make this if you're allowed to modify collection?)



```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
                 std::back_inserter(result_collection),
                 [](int n){
                   return n > 0;
                 });

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

    int sum = std::accumulate(end(result_collection)-3,
                                end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Conclusion

Wrapping up what we’ve learned

Conclusion -- C++ Programmers

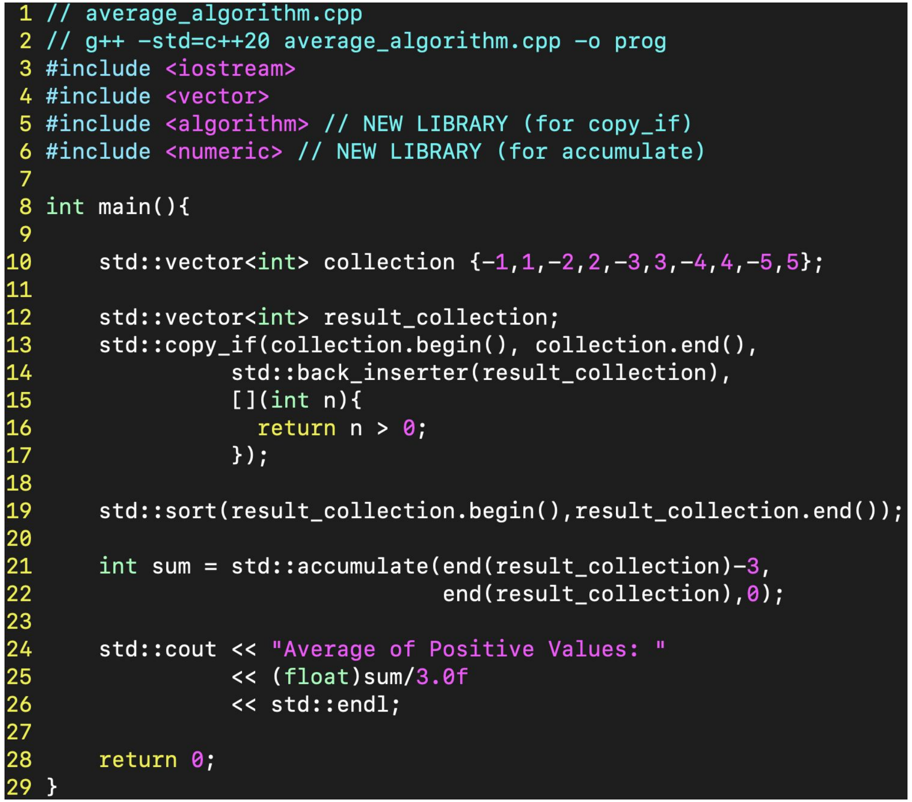
- We’ve taken a tour of writing a program (data structure + algorithm) from a very ‘**C with classes approach**’ to a ‘**C++ std::algorithm building blocks approach**’
- If you’re teaching C++ -- teach `std::algorithm` from the start.
    - Incorporate `std::algorithm` as early (as is reasonable) so your students can write better code.
    - We could have saved ourselves a long journey otherwise to writing more interesting code!
- `std::algorithm` can help you write more maintainable code that’s easier to reason about.
    - Yes--there are probably performance use cases if you’re building low latency trading systems or game engine programming where you’ll want to use vectorized loops and hand roll your own algorithms from scratch...

Further resources and training materials

- GoingNative 2013 - Sean Parent - C++ Seasoning
- CppCon 2015 - Michael VanLoon “STL Algorithms in Action ”
- CppCon 2016 - Marshall Clow “STL Algorithms - why you should use them, and how to write your own"
- CppCon 2018 - Jonathan Boccara “105 STL Algorithms in Less Than an Hour”
- CppCon 2019 - Dvir Yitzchaki - Range Algorithms, Views and Actions: A Comprehensive Guide
- CppCon 2019 - Conor Hoekstra “Algorithm Intuition (part 1 and part 2)”
- CppCon 2021 - Bob Steagall - Back to Basics: Classic STL
- CppCon 2021 - Bryce Adelstein Lelbach - C++ Standard Parallelism
- https://blog.tartanllama.xyz/accumulate-vs-reduce/
  - Discussion on std::accumulate vs std::reduce
- https://hackingcpp.com/cpp/std/algorithms/intro.html
  - Nice visualizations and cheat sheets on algorithms

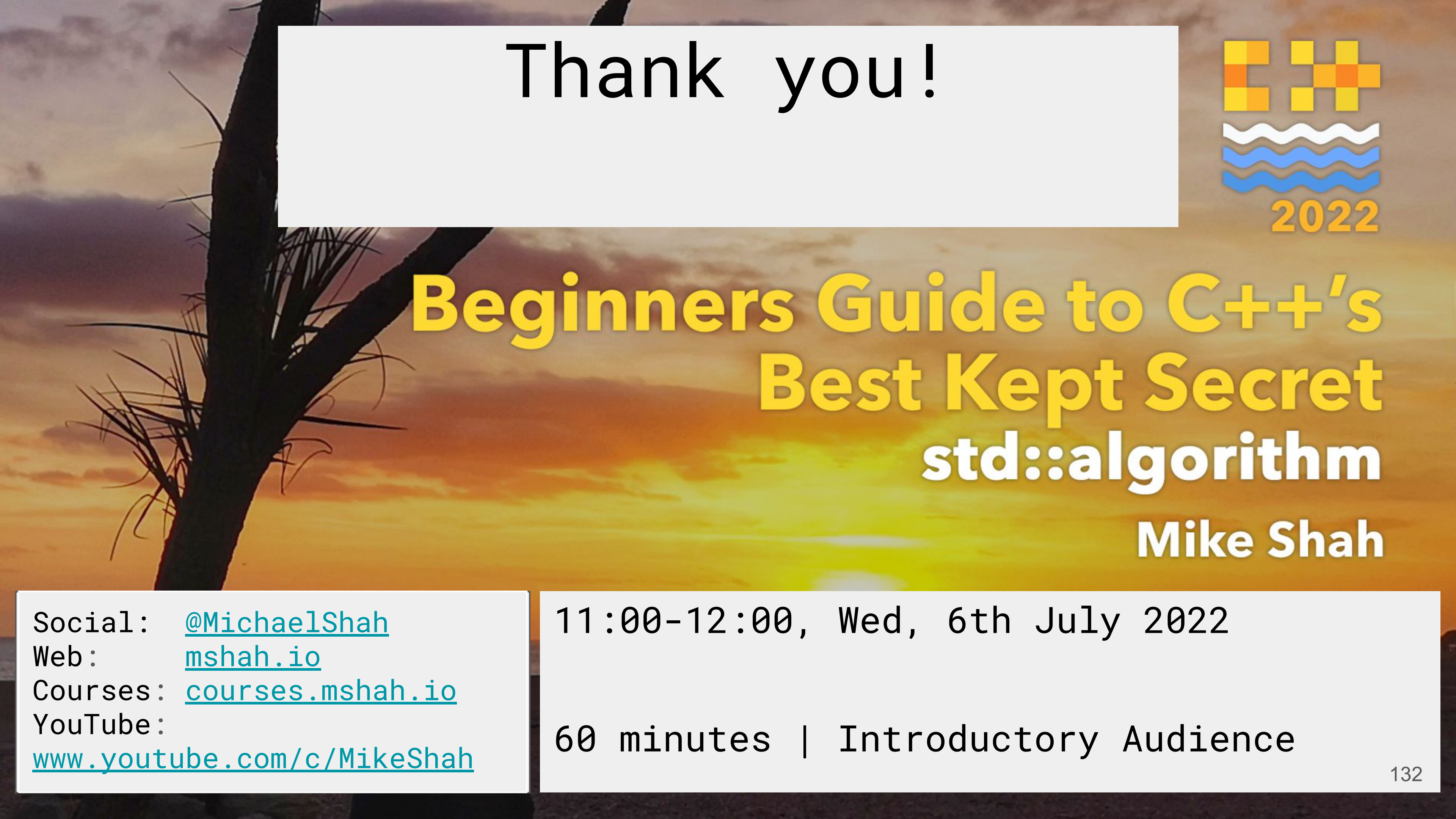
A Homework Assignment for Students

- Take a look at our example, and to rewrite it 5 different ways using different parts of std::algorithm.
  - **`remove_if`** - remove more safely negative values...
  - **`transform`** - Make all negative values 0, sort, then accumulate top 3 values
  - **`reverse`** - Sort, reverse, take top three values
  - **`make_heap`** - Then pop 3 elements
  - `etc.`
- As a learner get creative, try solutions, study complexity, and then measure -- while having fun!.



```cpp
// average_algorithm.cpp
// g++ -std=c++20 average_algorithm.cpp -o prog
#include <iostream>
#include <vector>
#include <algorithm> // NEW LIBRARY (for copy_if)
#include <numeric> // NEW LIBRARY (for accumulate)

int main(){

    std::vector<int> collection {-1,1,-2,2,-3,3,-4,4,-5,5};

    std::vector<int> result_collection;
    std::copy_if(collection.begin(), collection.end(),
              std::back_inserter(result_collection),
              [](int n){
                return n > 0;
              });

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

    int sum = std::accumulate(end(result_collection)-3,
                              end(result_collection),0);

    std::cout << "Average of Positive Values: "
              << (float)sum/3.0f
              << std::endl;

    return 0;
}
```

Thank you!



Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube:
www.youtube.com/c/MikeShah

11:00-12:00, Wed, 6th July 2022

60 minutes | Introductory Audience

Thank you!

Extra