Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice
- This slideset may not be mined using AI or algorithms and otherwise generating derivative products without permission.

Please do not redistribute slides/source without prior written permission.

Back To Basics
Ranges

MIKE SHAH

Back To Basics
Ranges

MIKE SHAH

Web: mshah.io
YouTube www.youtube.com/c/MikeShah
Social: mikeshah.bsky.social
Courses: courses.mshah.io
Talks: http://tinyurl.com/mike-talks

60 minutes | Audience: Beginner
9:00am - 10:00 Wed, Sept. 17, 2025

Abstract (Which you already read :))

Talk Abstract: Ranges have been available since C++20 to help enable writing and composing powerful algorithms. The use of ranges (along with the pipe syntax) can help you write more elegant and less error-prone code. But how and where to get started? Have no fear, in this talk I will provide a gentle introduction to getting you started with ranges. After attending this talk, you will be able to navigate ranges, know the difference between a range 'adaptor' and a range 'view', and be able to start using ranges in any C++20 compiler.

Slides/Code will be available after on www.mshah.io

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

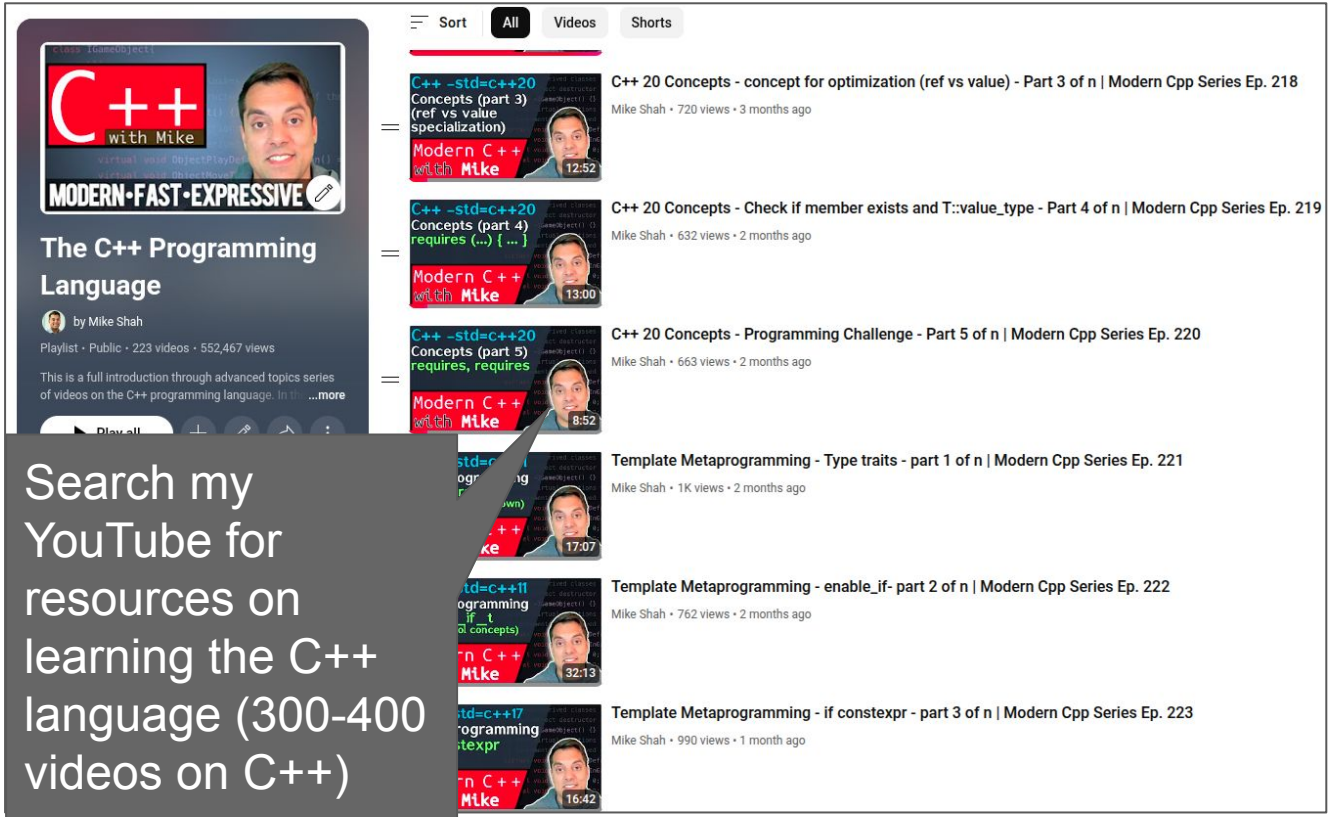
Your Tour Guide for Today

Mike Shah

- **My role at Cppcon:**
  - **Co-Chair of Back to Basics Track (**2022-2025) and **Software-Design Track** (2021-2025)
- **Current Job:** Teaching Faculty at **Yale University**
  - **Teach/Research**: computer systems, graphics, geometry, game engines, software engineering, and computer science education.
- **Available for:**
  - **Technical Training/Talks:** Occasionally
    - Please reach out, and I'll let you know if I have expertise that might add value for short term arrangements
- **Fun**:
  - Enjoy travel, NBA basketball, Olympics, running/weights, video games, music, DuoLingo (French mostly), and pizza making

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Find my programming content on YouTube
https://www.youtube.com/c/mikeshah



Search my YouTube for resources on learning the C++ language (300-400 videos on C++)

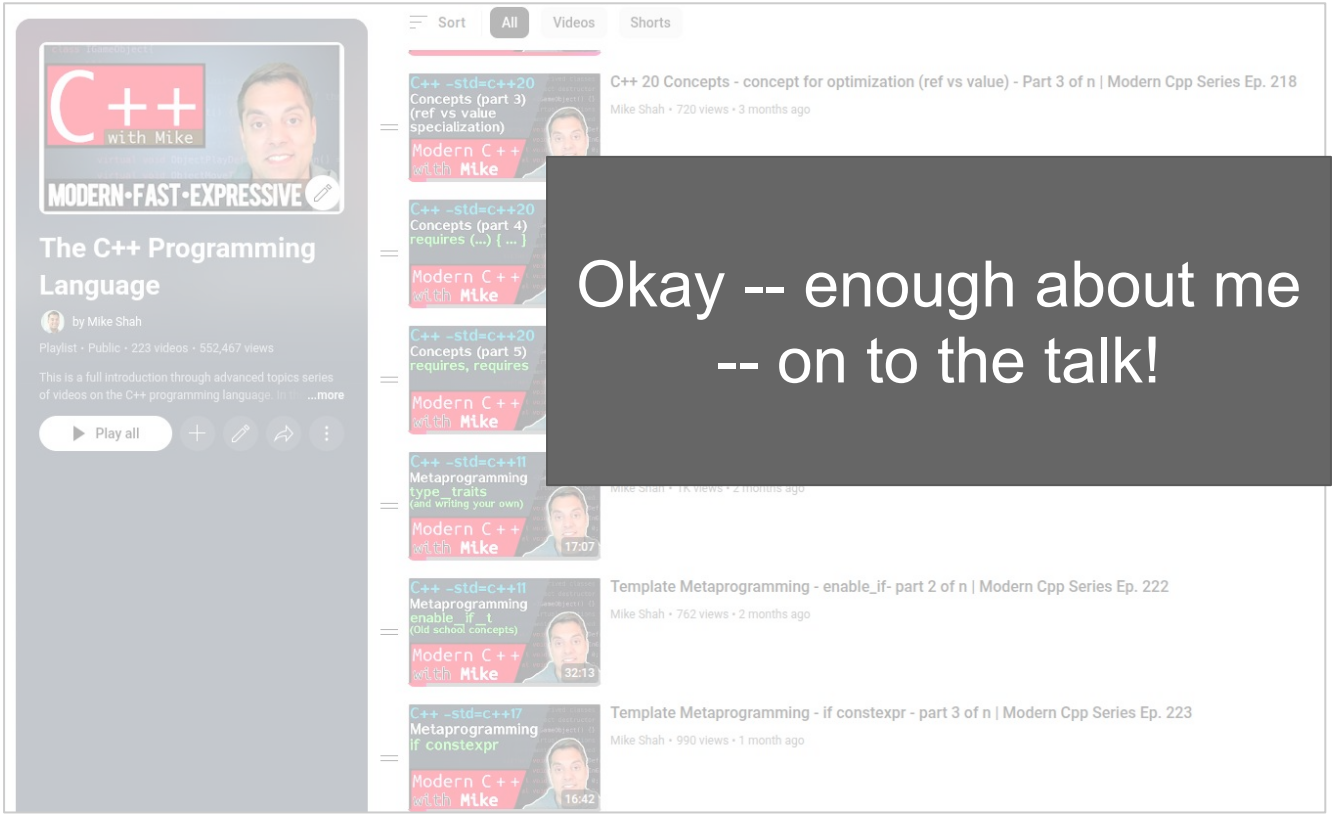
Web
www.mshah.io

YouTube
https://www.youtube.com/c/MikeShah

Non-Academic Courses
courses.mshah.io

Conference Talks
http://tinyurl.com/mike-talks

Find my programming content on YouTube
https://www.youtube.com/c/mikeshah



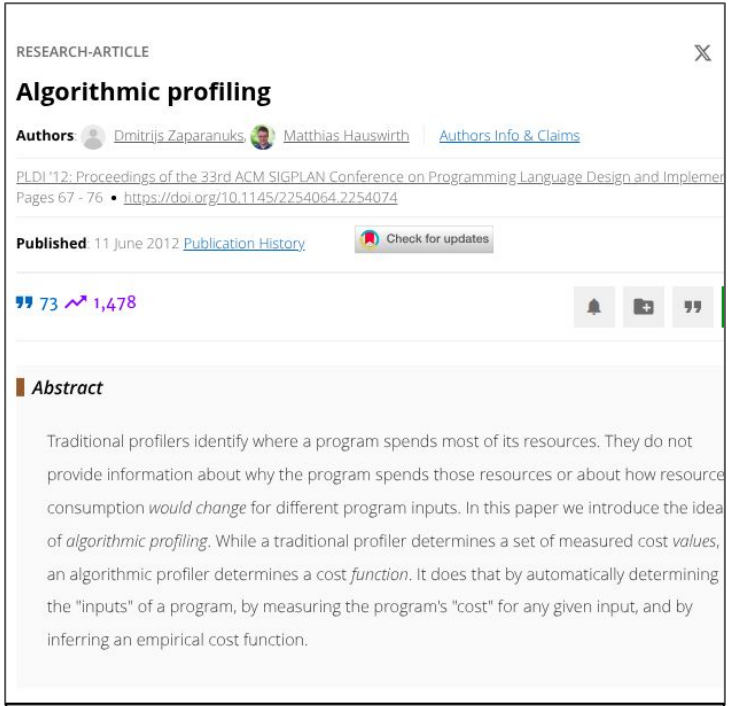
Okay -- enough about me -- on to the talk!

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Algorithmic Profiling

Algorithmic Profiling

- There's a paper I read in graduate school on algorithmic profiling that always sticks in my mind
  - Part of the idea was to find loops in your program, and then try to find or categorize the algorithmic complexity like some of us learn in school with a 'cost function' and empirical evaluation
  - One of the takeaways from myself, is that our programs are basically just loops
- So today we are going to talk about loops, iteration, and of course ranges!



RESEARCH-ARTICLE

Algorithmic profiling

Authors: Dmitrijs Zaparanuks, Matthias Hauswirth | Authors Info & Claims

PLDI '12: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implemen
Pages 67 - 76 • https://doi.org/10.1145/2254064.2254074

Published: 11 June 2012 Publication History | Check for updates

73 1,478

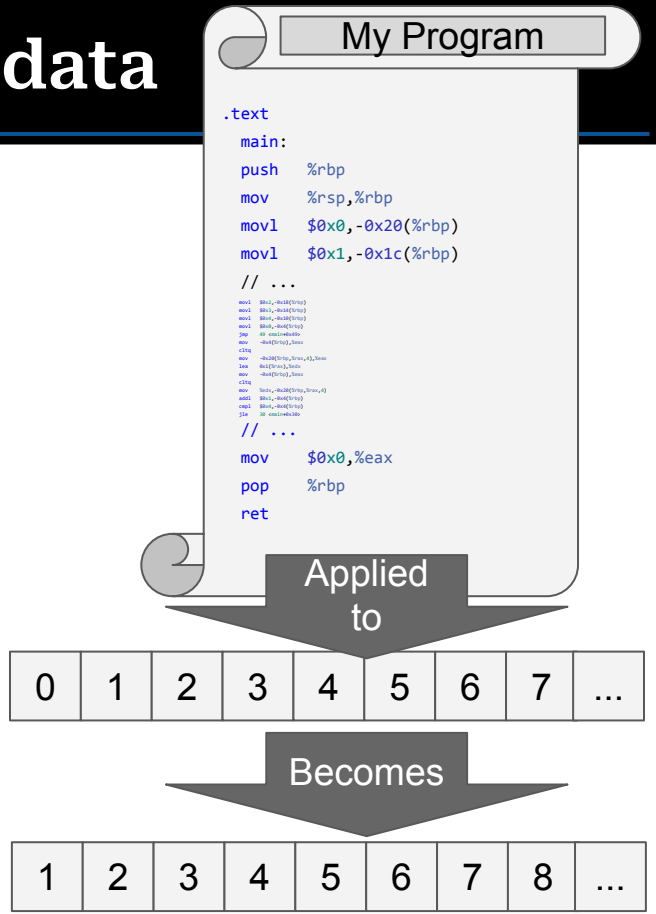
Abstract

Traditional profilers identify where a program spends most of its resources. They do not provide information about why the program spends those resources or about how resource consumption *would change* for different program inputs. In this paper we introduce the idea of *algorithmic profiling*. While a traditional profiler determines a set of measured cost *values*, an algorithmic profiler determines a cost *function*. It does that by automatically determining the "inputs" of a program, by measuring the program's "cost" for any given input, and by inferring an empirical cost function.

https://dl.acm.org/doi/10.1145/2254064.2254074

Remember Computers transform data

- That's about half of the job of a computer (the other half is to store the data), okay this is the ultimate simplification -- but
  - The better abstractions we have to perform transformations, perhaps the easier it will be for us help the computer do its job
- Loops are a fundamental tool we have that allows us to repeatedly transform data.
  - So, let's start with talking about loops and transformation/scanning of data



Goal for Today

- Show you how to write loops using C++2x std::ranges
  - 2x meaning ‘c++20 or later’
- Using std::ranges I will argue:
  - is safer
  - more composable
  - more maintainable
  - creates more malleable code
- Let’s understand why -- starting with our basic building blocks
  - **(next slide to begin!)**

```
// ========================================
// Prior to Ranges
std::vector v1{3,9,7,5,1};
std::sort(std::begin(v1), std::end(v1));
std::println("{}",v1);

// With ranges
std::vector v2{3,9,7,5,1};
std::ranges::sort(v2);
std::println("{}",v2);
// ========================================
```

```
std::vector v5{1,3,5,7,11,13,17,19};
auto doubleValue = [](int i){ return i*2; };
auto AddOne = [](int i){ return i+1; };

// Notice this time no 'auto&' -- need the 'l-value'
for(auto elem : v5
               | std::views::transform(doubleValue)
               | std::views::transform(AddOne)

){
    std::cout << elem << " ";
}
std::cout << std::endl;
```

Iteration

“I remember learning repeat...until in BASIC and thinking that was a revelation when I started programming -- for-loops were even cooler!”

`for` loop

Conditionally executes a statement repeatedly, where the statement does not need to manage the loop condition.

Syntax

attr(optional) **for** **(** *init-statement condition*(optional) **;** *expression*(optional) **)** *statement*

attr - (since C++11) any number of attributes

init-statement - one of

- an expression statement (which may be a null statement `;`)
- a simple declaration (typically a declaration of a loop counter variable with initializer), it may declare arbitrary many variables or structured bindings(since C++17)
- an alias declaration (since C++23)

Note that any *init-statement* must end with a semicolon. This is why it is often described informally as an expression or a declaration followed by a semicolon.

condition - a condition

expression - an expression (typically an expression that increments the loop counter)

statement - a statement (typically a compound statement)

https://cppreference.com/w/cpp/language/for.html

For-Loop (1/2)

- A for-loop is one of the fundamental building blocks for defining where computation (1) starts, (2) continues, and (3) ends
  - Note: In an algorithms textbook you may learn these as initialization, maintenance, and termination in regard to a loop variant

```
 1 // @file: for3.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 for3.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <string>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   char message[] = "Hello everyone, welcome!";
11
12
13
14
15   for(std::size_t i=0; i <        sizeof(message); ++i){
16     std::print("{}",message[i]);
17   }
18   std::println();
19
20
21
22
23
24
25   return 0;
26 }
~
~
"for3.cpp" 26L, 359B                                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for3.cpp -o prog && ./prog
Hello everyone, welcome!
```

For-Loop (2/2)

- In this example the for-loop that is traversing a the contents of the array ‘message’
  - The work we are doing is simply printing out the characters

```
 1 // @file: for3.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 for3.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <string>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   char message[] = "Hello everyone, welcome!";
11
12
13
14
15   for(std::size_t i=0; i <        sizeof(message); ++i){
16     std::print("{}",message[i]);
17   }
18   std::println();
19
20
21
22
23
24
25   return 0;
26 }
~
~
"for3.cpp" 26L, 359B                                  1,1           All

mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for3.cpp -o prog && ./prog
Hello everyone, welcome!
```

For-loop: std::array

- Here is the same example, but I use a std::array container with each element as a character.
- Note:
  - std::array is often preferred -- why?
    - A std::array knows its length
    - std::array does not decaying into a pointer when passed as a function arguments
    - (See also std::span)

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-stack

```
1  // @file: for4.cpp
2  // Build and run
3  // g++-14 -std=c++23 for4.cpp -o prog && ./prog
4  #include <iostream>
5  #include <array>
6  #include <print>
7
8  int main(int argc, char* argv[]){
9
10   std::array message {'H','e','l','l','o',' ','e','v','e','r','y'
11                ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // This also works
13   // std::array<char,24> message = {"Hello everyone welcome!"};
14   // Class Template Argument Deduction (CTAD),
15   // allows us to omit type and size parameters
16   // for std::array if it can be inferred.
17
18   for(std::size_t i=0; i < message.size(); ++i){
19     std::print("{}",message[i]);
20   }
21   std::println();
22
23
24
25
26   return 0;
27 }
~
"for4.cpp" 27L, 656B written                          19,27         All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for4.cpp -o prog && ./prog
Hello everyone welcome!
mike@mike-MS-7B17:ranges$
```

For-Loop: string (1/2)

- Choosing perhaps a better Standard Template Library (STL) container gives us a more ideal way to represent a ‘string’
- But again, the same work is being done.
  - And a std::string provides us some advantages of knowing the length and preserving the type like the std::array did.

```
1  // @file: for.cpp
2  // Build and run
3  // g++-14 -std=c++23 for.cpp -o prog && ./prog
4  #include <iostream>
5  #include <string>
6  #include <print>
7
8  int main(int argc, char* argv[]){
9
10   std::string        message = "Hello everyone, welcome!"
11
12
13
14
15
16   for(std::size_t i=0; i <        message.length(); ++i){
17     std::print("{}",message[i]);
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"for.cpp" 27L, 371B written
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for.cpp -o prog && ./prog
Hello everyone, welcome!
mike@mike-MS-7B17:ranges$
```

For-Loop: string (2/2)

- Highted are two for-loops doing work to transform each character in a std::string
  - **loop 1:** encoding a secret message first
  - **loop 2:** decoding the secret message in the second loop
- In this instance each loop is doing it's own separate algorithm, separated into two chunks of code.

```
1  // @file: for2.cpp
2  // Build and run
3  // g++-14 -std=c++23 for2.cpp -o prog && ./prog
4  #include <iostream>
5  #include <string>
6  #include <print>
7
8  int main(int argc, char* argv[]){
9
10   std::string secret_message = "Hello everyone, welcome!";
11   int key = 1;
12   std::string encoded_message;
13   std::string decoded_message;
14
1    // 'caesar cipher' to 'shift' characters
16   for(std::size_t i=0; i < secret_message.length(); ++i){
17     encoded_message += secret_message[i] + key;
18   }
19   std::println("encoded message: {}",encoded_message);
20
2    // 'Run opposite' algorithm to decode
22   for(std::size_t i=0; i < secret_message.length(); ++i){
23     decoded_message += encoded_message[i] - key;
24   }
25   std::println("decoded_message: {}",decoded_message);
26   return 0;
27 }
~
"for.cpp" 27L, 741B written
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for2.cpp -o prog && ./prog
encoded_message: Ifmmp!fwfszpof-!xfmdpnf"
decoded_message: Hello everyone, welcome!
```

pointer for iteration (1/2)

- Here is the std::array example again, but this time I am ‘preparing’ for my iteration outside the for-loop
- Rather than doing this directly in the for-loop, I setup ‘**ptr**’ and **‘end’**
  - **ptr** represents the start of my computation
  - **end** represents the termination of computation.

```
1  // @file: for5.cpp
2  // Build and run
3  // g++-14 -std=c++23 for5.cpp -o prog && ./prog
4  #include <iostream>
5  #include <array>
6  #include <print>
7
8  int main(int argc, char* argv[]){
9
10   std::array message {'H','e','l','l','o',' ','e','v','e','r','y'
11                ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13   auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr   = data;
15   auto end   = data + message.size(); // last element
16   for(/* empty */ ; ptr != end ; ++ptr){
17     std::print("{}",*ptr);        // or std::cout << *ptr
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"for5.cpp" 27L, 631B                                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for5.cpp -o prog && ./prog
Hello everyone welcome!
```

pointer for iteration (2/2)

- Observe the ‘conditional’ portion of the for-loop is using a comparison
- Observe the ‘expression’ portion of the for-loop is using (pointer) arithmetic

```
 1 // @file: for5.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 for5.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <array>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   std::array message {'H','e','l','l','o',' ','e','v','e','r','y'
11                ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13   auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr   = data;
15   auto end   = data + message.size(); // last element
16   for(/* empty */ ; ptr != end ; ++ptr){
17     std::print("{}",*ptr);        // or std::cout << *ptr
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"for5.cpp" 27L, 631B                                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for5.cpp -o prog && ./prog
Hello everyone welcome!
```

Pointers Generalize

- The reason for using **pointers** is as a strategy for *generalizing how to iterate* a data structure
- So the question is -- how could I *generalize* this idea of moving pointers in a flexible way?
- Next slide for the thought exercise!

```
1  // @file: for5.cpp
2  // Build and run
3  // g++-14 -std=c++23 for5.cpp -o prog && ./prog
4  #include <iostream>
5  #include <array>
6  #include <print>
7
8  int main(int argc, char* argv[]){
9
10   std::array message {'H','e','l','l','o',' ','e','v','e','r','y'
11                ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13   auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr   = data;
15   auto end   = data + message.size(); // last element
16   for(/* empty */ ; ptr != end ; ++ptr){
17     std::print("{}",*ptr);        // or std::cout << *ptr
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"for5.cpp" 27L, 631B                                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for5.cpp -o prog && ./prog
Hello everyone welcome!
```

(The Question is…)

- What if we had a ‘tree’ or graph structure to iterate?
  - Ans: We would want to create perhaps a pointer ‘object type’ to represent our iteration (and then define what ++ means)
- This is exactly the idea of a *general iterator object*, which controls how we move to the next ‘element’

```
 1 // @file: for5.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 for5.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <array>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   std::array message {'H','e','l','l','o',' ','e','v','e','r','y'
11                ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13   auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr   = data;
15   auto end   = data + message.size(); // last element
16   for(/* empty */ ; ptr != end ; ++ptr){
17     std::print("{}",*ptr);        // or std::cout << *ptr
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"for5.cpp" 27L, 631B                                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 for5.cpp -o prog && ./prog
Hello everyone welcome!
```

begin/end iterators

- The Standard Template Library (STL) gives us free functions `std::begin` and `std::end` which return an ‘iterator’ object for our containers
  - (And even other non-owning containers like std::span, std::mdspan, etc.)

```
// @file: array_iterator2.cpp
// Build and run
// g++-14 -std=c++23 array_iterator2.cpp -o prog && ./prog
#include <iostream>
#include <array>
#include <print>

int main(int argc, char* argv[]){

  std::array message {'H','e','l','l','o',' ','e','v','e','r','y'
                ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
  // find first element and point to it.
//  auto data = &message.data()[0]; // or equally 'message.data()';
  auto ptr   = std::begin(message);
  auto end   = std::end(message); // last element
  for(/* empty */ ; ptr != end ; ++ptr){
    std::print("{}",*ptr);        // or std::cout << *ptr
  }
  std::println();









  return 0;
}
~
"array_iterator2.cpp" 27L, 666B written                10,3          All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 array_iterator2.cpp -o prog
 && ./prog
Hello everyone welcome!
```

Iterators

A generalization of pointers

Iterator library

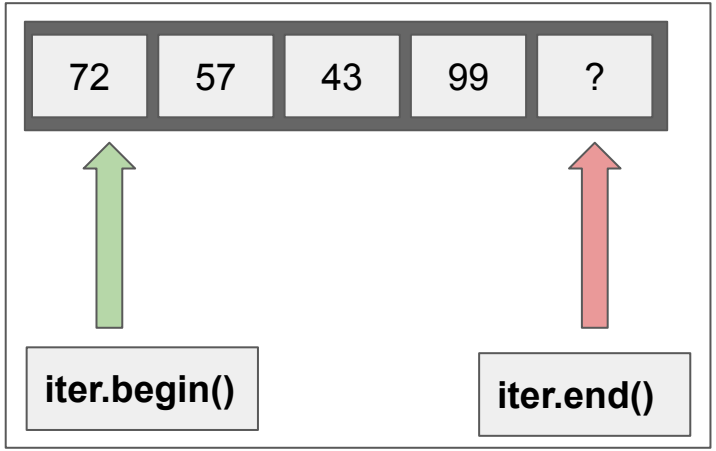
Iterators are a generalization of pointers that allow a C++ program to work with different data structures (for example, containers and ranges(since C++20)) in a uniform manner. The iterator library provides definitions for iterators, as well as iterator traits, adaptors, and utility functions.

Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers.

https://en.cppreference.com/w/cpp/iterator.html

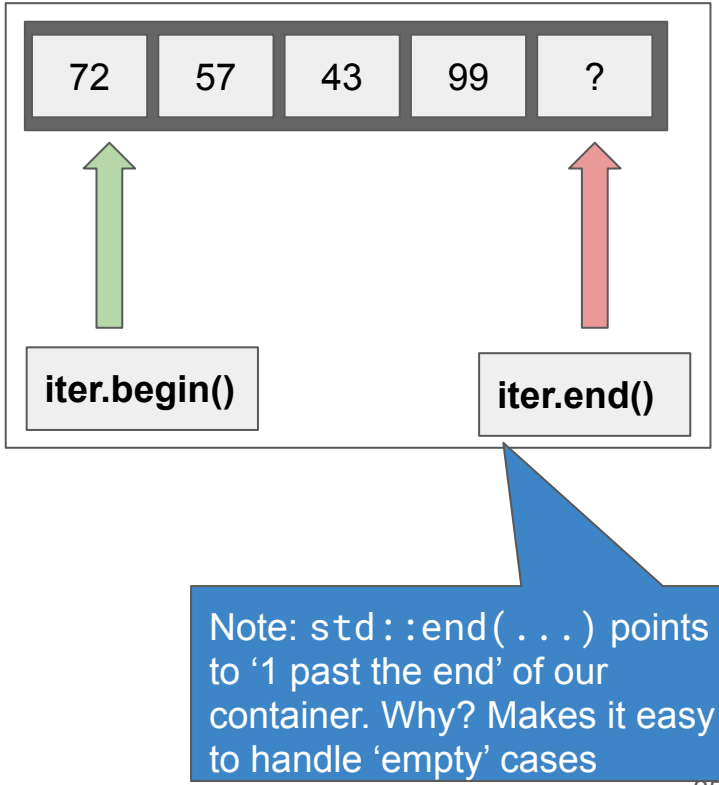
Iterators in C++ (1/5)

- An iterator in C++ is an object that allows you to access elements in a collection.
  - Internally -- an iterator may be a pointer to a specific element or an index, but we don't really care
- What we care about, is having a consistent way to 'advance' (iterate) forwards
  - (Or perhaps backwards, or perhaps with random access -- we can control an iterators power)



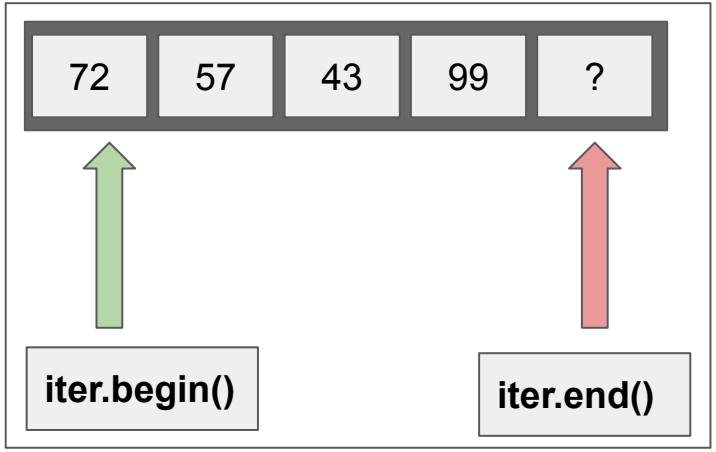
Iterators in C++ (2/5)

- An iterator in C++ is an object that allows you to access elements in a collection.
  - Internally -- an iterator may be a pointer to a specific element or an index, but we don't really care
- What we care about, is having a consistent way to 'advance' (iterate) forwards
  - (Or perhaps backwards, or perhaps with random access -- we can control an iterators power)



Iterators in C++ (3/5)

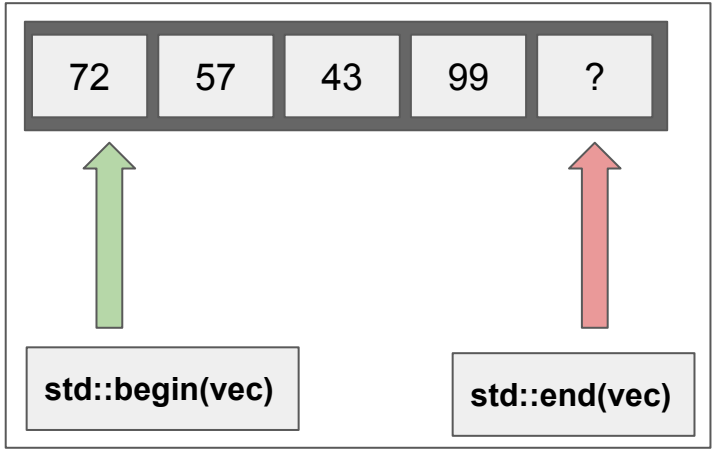
- Most containers provide `.begin()` and `.end()` member functions that will return an iterator that points to the beginning and end of the container.



Iterators	
begin **cbegin** (C++11)	returns an iterator to the beginning (public member function)
end **cend** (C++11)	returns an iterator to the end (public member function)
rbegin **crbegin** (C++11)	returns a reverse iterator to the beginning (public member function)
rend **crend** (C++11)	returns a reverse iterator to the end (public member function)

Iterators in C++ (4/5)

- Note: I prefer `std::begin` and `std::end` free functions which are more generic versus the specific member functions for each container
  - (and as far as implementation details, these free functions may likely call the specific member function anyway)



std::begin, std::cbegin

Returns an iterator to the beginning of the given range.

std::end, std::cend

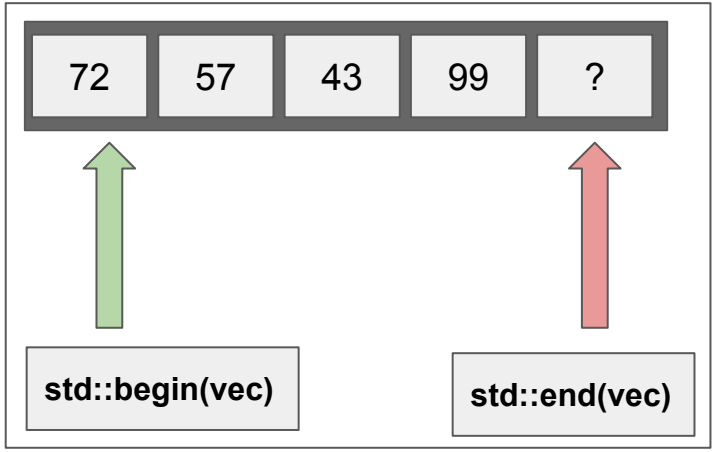
Returns an iterator to the end (i.e. the element after the last element) of the given range.

Iterators

begin cbegin (C++11)	returns an iterator to the beginning (public member function)
end (C++11)	returns an iterator to the end (public member function)
in gin (C++11)	returns a reverse iterator to the beginning (public member function)
rend crend (C++11)	returns a reverse iterator to the end (public member function)

Iterators in C++ (5/5)

- Iterators can advance to the next element in some manner:
  - (`std::advance`, `operator++`, `operator--`, etc.)
- How we are allowed to advance is defined by the specific iterator
  - We can move forward sequentially, backwards, or perhaps random access depending on the iterator object [more on iterator library]
  - See [1] [2]
- Other generic functions like std::advance or std::distance otherwise exist

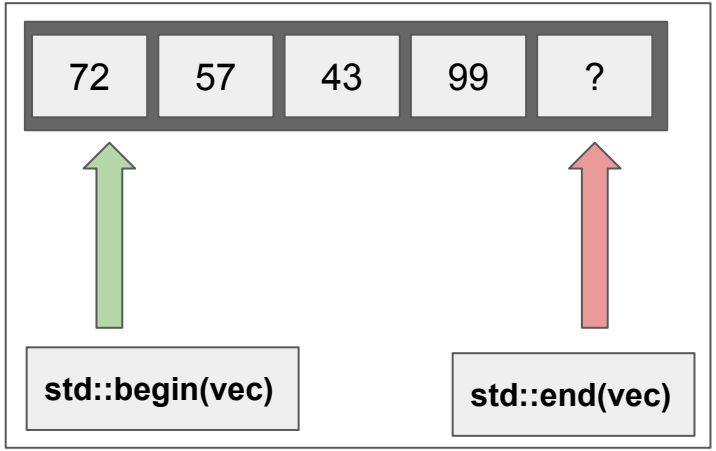


[1] STL C++ Iterators - Introduction | Modern Cpp Series Ep. 135

[2] STL C++ Iterators - Categories | Modern Cpp Series Ep. 136

Iterators in C++ - 2 iterators make a range

- Generally speaking we use a pair of iterators to move through the beginning and end of a container 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 in some algorithm
  - Note: A std::range is subtly different, but for the most part we can think of it this way



[1] STL C++ Iterators - Introduction | Modern Cpp Series Ep. 135

[2] STL C++ Iterators - Categories | Modern Cpp Series Ep. 136

begin/end iterators

- So, here's where we left off, but now we understand std::begin and std::end are a begin/end iterator pair defining the range of our computation
- We have a **container** (std::array) and a pair of **iterators**

```
// @file: array_iterator2.cpp
// Build and run
// g++-14 -std=c++23 array_iterator2.cpp -o prog && ./prog
#include <iostream>
#include <array>
#include <print>

int main(int argc, char* argv[]){

  std::array message {'H','e','l','l','o',' ','e','v','e','r','y'
                ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
  // find first element and point to it.
//  auto data = &message.data()[0]; // or equally 'message.data()';
  auto ptr   = std::begin(message);
  auto end   = std::end(message); // last element
  for(/* empty */ ; ptr != end ; ++ptr){
    std::print("{}",*ptr);        // or std::cout << *ptr
  }
  std::println();









  return 0;
}
~
"array_iterator2.cpp" 27L, 666B written                10,3          All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 array_iterator2.cpp -o prog
 && ./prog
Hello everyone welcome!
```

Iterators are generic

- Observe I just switched the data type from std::array to std::vector
- **Iterators *help* support writing generic code**
  - What I mean by this, is I can use the same functions begin/end pair of functions as a user of the standard template library -- and usually this *just works* if I have templated code as well.

```
 1 // @file: vector_iterator.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 vector_iterator.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <vector>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   std::vector message {'H','e','l','l','o',' ','e','v','e','r','y'
11                       ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13 //  auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr   = std::begin(message);
15   auto end   = std::end(message); // last element
16   for(/* empty */ ; ptr != end ; ++ptr){
17     std::print("{}",*ptr);        // or std::cout << *ptr
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"vector_iterator.cpp" 27L, 668B written                10,13         All

mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 vector_iterator.cpp -o prog
 && ./prog
Hello everyone welcome!
```

Iterators are generic

- Simple arithmetic can be used to ‘advance’ most iterators
  - In this example, I only want to look from the beginning to 5 elements past the beginning -- so I simply add 5 to the beginning

```
 1 // @file: vector_iterator2.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 vector_iterator2.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <vector>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   std::vector message {'H','e','l','l','o',' ','e','v','e','r','y'
11                       ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13 //  auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr   = std::begin(message);
15   auto end   = std::begin(message)+5; // first 'n' elements
16   for(/* empty */ ; ptr != end ; ++ptr){
17     std::print("{}",*ptr);        // or std::cout << *ptr
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"vector_iterator2.cpp" 27L, 680B                    1,1           All

mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 vector_iterator2.cpp -o pro
g && ./prog
Hello
```

Iterators are generic

- Alternatively, we also can ‘advance’ our iterator with std::advance.
  - This is also useful for data structures that do not have random-access, may use std::advance.
  - (i.e. auto end = std::begin(message)+5 is not supported for std::list)

```
 1 // @file: list_iterator.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 list_iterator.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <list>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   std::list message {'H','e','l','l','o',' ','e','v','e','r','y'
11                     ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13 //  auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr   = std::begin(message);
15   auto end   = std::begin(message);
16   std::advance(end,5); // first 'n' elements
17   for(/* empty */ ; ptr != end ; ++ptr){
18     std::print("{}",*ptr);        // or std::cout << *ptr
19   }
20   std::println();
21
22
23
24
25
26
27   return 0;
28 }
"list_iterator.cpp" 28L, 691B written                 17,31         All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 list_iterator.cpp -o prog &
& ./prog
Hello
```

Iterators are generic

- Important note when doing arithmetic whether manually or with std::advance
- You always have to be careful of your ‘bounds’ when using arithmetic in this manner
  - Observe the extra ‘bounds check’
- **That is part of the theme with iterators -- to be careful!**

```
 1 // @file: vector_iterator3.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 vector_iterator3.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <vector>
 6 #include <print>
 7
 8 int main(int argc, char* argv[]){
 9
10   std::vector message {'H','e','l','l','o',' ','e','v','e','r','y'
11                       ,'o','n','e',' ', 'w','e','l','c','o','m','e','!'};
12   // find first element and point to it.
13 //  auto data = &message.data()[0]; // or equally 'message.data()';
14   auto ptr    = std::begin(message);
15   auto end    = std::begin(message)+500; // first 'n' elements
16   auto actual_end = std::end(message);
17   for(/* empty */ ; ptr != end && ptr!=actual_end ; ++ptr){
18     std::print("{}",*ptr);        // or std::cout << *ptr
19   }
20   std::println();
21
22
23
24
25
26
27   return 0;
28 }
"vector_iterator3.cpp" 28L, 740B written                15,38         All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 vector_iterator3.cpp -o pro
g && ./prog
Hello everyone welcome!
```

Iterators are generic

- Note on **auto**
- In this example I have listed out the explicit type for the iterator so you can see.
  - Sometimes typing out the full type makes it easier to map back to any errors reported.
    - It also makes sense if you want to be explicit and show intent as to what datatype is needed.
  - Typically I like using ‘auto’ of typing out the full iterator type.
    - It keeps my code malleable if I later change the data type.

```
1 // @file: explicit_types.cpp
2 // Build and run
3 // g++-14 -std=c++23 explicit_types.cpp -o prog && ./prog
4 #include <iostream>
5 #include <list>
6 #include <print>
7
8 int main(int argc, char* argv[]){
9
10   std::list message {'H','e','l','l','o','\n','\0'};
11   // find first element and point to it.
12 //  auto data = &message.data()[0]; // or equally 'message.data()';
13   auto                        ptr   = std::begin(message);
14   std::list<char>::iterator end   = std::begin(message);
15   std::advance(end,5); // first 'n' elements
16   for(/* empty */ ; ptr != end ; ++ptr){
17     std::print("{}",*ptr);        // or std::cout << *ptr
18   }
19   std::println();
20
21
22
23
24
25
26   return 0;
27 }
~
"explicit_types.cpp" 27L, 655B written                13,27         All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 explicit_types.cpp -o prog
&& ./prog
Hello
```

std::map iterator (1/2)

- Here is another example with iterators working on a std::map
  - Note: I am using ‘using’ to create less noise later on with the type
  - Otherwise the iterator type is:
  - `std::map<std::string,int>::iterator`

```
 1 // @file: explicit_types2.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 explicit_types2.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <string>
 6 #include <map>
 7 #include <print>
 8
 9 int main(int argc, char* argv[]){
10   // An 'ordered' map
11   // Note: I use 'using' in this example so that it's easier
12   //       later on with my '::iterator', or otherwise if I want
13   //       to change the type to something like unordered_map..
14   using StringIntMap = std::map<std::string, int>;
15   StringIntMap myMap {  {"Mike",123},
16                         {"Bjarne", 234},
17                         {"Walter", 345},
18                      };
19
20   // Iterators defined with a for-loop
21   for(StringIntMap::iterator iter = myMap.begin() ; // init
22       iter != myMap.end() ;                         // condition
23       iter++)                                       // expression
24   {
25     std::println("key: {} value: {}", iter->first, iter->second);
26   }
27   std::println();
28
29   return 0;
30 }
1 more line; before #6  1 second ago                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 range_based_for.cpp -o prog
 && ./prog
key: Bjarne value: 234
key: Mike value: 123
key: Walter value: 345
```

std::map iterator (2/2)

- Here is the iterator in use, this time encapsulated within the ‘for-loop’
  - This perhaps better ‘encapsulates’ our range of computation.

```
 1 // @file: explicit_types2.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 explicit_types2.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <string>
 6 #include <map>
 7 #include <print>
 8
 9 int main(int argc, char* argv[]){
10   // An 'ordered' map
11   // Note: I use 'using' in this example so that it's easier
12   //       later on with my '::iterator', or otherwise if I want
13   //       to change the type to something like unordered_map..
14   using StringIntMap = std::map<std::string, int>;
15   StringIntMap myMap {  {"Mike",123},
16                         {"Bjarne", 234},
17                         {"Walter", 345},
18                      };
19
20   // Iterators defined with a for-loop
21   for(StringIntMap::iterator iter = myMap.begin() ; // init
22       iter != myMap.end() ;                         // condition
23       iter++)                                       // expression
24   {
25     std::println("key: {} value: {}", iter->first, iter->second);
26   }
27   std::println();
28
29   return 0;
30 }
1 more line; before #6  1 second ago                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 range_based_for.cpp -o prog
 && ./prog
key: Bjarne value: 234
key: Mike value: 123
key: Walter value: 345
```

Range-based for loop

Range-based `for` loop (since C++11)

Executes a `for` loop over a range.

Used as a more readable equivalent to the traditional `for` loop operating over a range of values, such as all elements in a container.

Syntax

attr(optional) **for** **(** *init-statement*(optional) *item-declaration* **:** *range-initializer* **)** *statement*

attr	-	any number of attributes
init-statement	-	(since C++20) one of • an expression statement (which may be a null statement `;`) • a simple declaration (typically a declaration of a variable with initializer), it may declare arbitrarily many variables or be a structured binding declaration • an alias declaration (since C++23) Note that any *init-statement* must end with a semicolon. This is why it is often described informally as an expression or a declaration followed by a semicolon.
item-declaration	-	a declaration for each range item
range-initializer	-	an expression or brace-enclosed initializer list
statement	-	any statement (typically a compound statement)

https://cppreference.com/w/cpp/language/for.html

Improving our ‘for-loop’

- This pattern of using iterators to iterate through an entire container (beginning to end) is so common, that in C++11, we got ‘range-based for-loops’
- Let me show you the equivalent *range-based for-loop*
  - **next slide**

```
 1 // @file: explicit_types2.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 explicit_types2.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <string>
 6 #include <map>
 7 #include <print>
 8
 9 int main(int argc, char* argv[]){
10   // An 'ordered' map
11   // Note: I use 'using' in this example so that it's easier
12   //       later on with my '::iterator', or otherwise if I want
13   //       to change the type to something like unordered_map..
14   using StringIntMap = std::map<std::string, int>;
15   StringIntMap myMap {  {"Mike",123},
16                         {"Bjarne", 234},
17                         {"Walter", 345},
18                      };
19
20   // Iterators defined with a for-loop
21   for(StringIntMap::iterator iter = myMap.begin() ; // init
22       iter != myMap.end() ;                         // condition
23       iter++)                                       // expression
24   {
25     std::println("key: {} value: {}", iter->first, iter->second);
26   }
27   std::println();
28
29   return 0;
30 }
1 more line; before #6  1 second ago                  1,1           All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 range_based_for.cpp -o prog
 && ./prog
key: Bjarne value: 234
key: Mike value: 123
key: Walter value: 345
```

Ranged-Based for-loop

- Range-based for-loops more succinctly define our previous intent
- The interface is much easier to use
  - (Note: Yes, we have also had std::for_each for a long time too)

```
1  // @file: range_based_for.cpp
2  // Build and run
3  // g++-14 -std=c++23 range_based_for.cpp -o prog && ./prog
4  #include <iostream>
5  #include <string>
6  #include <map>
7  #include <print>
8
9  int main(int argc, char* argv[]){
10   // An 'ordered' map
11   // Note: I use 'using' in this example so that it's easier
12   //       later on with my '::iterator', or otherwise if I want
13   //       to change the type to something like unordered_map..
14   using StringIntMap = std::map<std::string, int>;
15   StringIntMap myMap {  {"Mike",123},
16                         {"Bjarne", 234},
17                         {"Walter", 345},
18                      };
19
20   // Iterators defined with a for-loop
21   for(auto& pair : myMap)
22
23
24   {
25     std::println("key: {} value: {}", pair.first, pair.second);
26   }
27   std::println();
28
29
30
"range_based_for.cpp" 35L, 797B                    1,1           Top
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 range_based_for.cpp -o prog
 && ./prog
key: Bjarne value: 234
key: Mike value: 123
key: Walter value: 345
```

- For-loop, ranged based for-loop, while, do-while, and std::fill | Modern Cpp Series Ep. 14
- See also std::for_each (still useful for the parallel execution policies) https://en.cppreference.com/w/cpp/algorithm/for_each.html

Structured Binding

- Structured bindings give us perhaps an easier way to pull out information than using the ‘pair’ defined within our range-based for-loop
- Note: Structured bindings also provide other syntax (e.g. ... (the three dots)) if you have more elements -- very powerful!

```
 1 // @file: structured_binding.cpp
 2 // Build and run
 3 // g++-14 -std=c++23 structured_binding.cpp -o prog && ./prog
 4 #include <iostream>
 5 #include <string>
 6 #include <map>
 7 #include <print>
 8
 9 int main(int argc, char* argv[]){
10   // An 'ordered' map
11   // Note: I use 'using' in this example so that it's easier
12   //       later on with my '::iterator', or otherwise if I want
13   //       to change the type to something like unordered_map..
14   using StringIntMap = std::map<std::string, int>;
15   StringIntMap myMap {  {"Mike",123},
16                         {"Bjarne", 234},
17                         {"Walter", 345},
18                      };
19
20   // Iterators defined with a for-loop
21   for(auto& [key,value] : myMap)
22
23
24   {
25     std::println("key: {} value: {}", key, value);
26   }
27   std::println();
28
29
30
"structured_binding.cpp" 35L, 797B written          25,48         Top
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 structured_binding.cpp -o p
rog && ./prog
key: Bjarne value: 234
key: Mike value: 123
key: Walter value: 345
```

Behind the scenes, ranged-based for-loop is lowered... (1/2)

- to use iterators (in a new scope)!
- See for yourself on cppinsights: https://cppinsights.io/s/b34ae70a

Source:

```
1  // @file: range_based_simple.cpp
2  // Build and run
3  // g++-14 -std=c++23 range_based_si
4  #include <iostream>
5  #include <vector>
6
7  int main(int argc, char* argv[]){
8
9    std::vector v = {1,3,5,7};
10   for(auto& elem : v)
11   {
12
13   }
14
15   return 0;
16 }
17
```

Insight:

```
1  #include <iostream>
2  #include <vector>
3
4  int main(int argc, char ** argv)
5  {
6    std::vector<int, std::allocator<int> > v = std::vector<int, std::allo
7    {
8      std::vector<int, std::allocator<int> > & __range1 = v;
9      std::__wrap_iter<int *> __begin1 = __range1.begin();
10     std::__wrap_iter<int *> __end1 = __range1.end();
11     for(; !std::operator==(__begin1, __end1); __begin1.operator++()) {
12       int & elem = __begin1.operator*();
13     }
14
15   }
16   return 0;
17 }
```

Behind the scenes, ranged-based for-loop is lowered... (2/2)

- So it's good we know iterators!
- Some of the new C++ features, just build on old abstractions

Source:

```
1  // @file: range_based_simple.cpp
2  // Build and run
3  // g++-14 -std=c++23 range_based_si
4  #include <iostream>
5  #include <vector>
6
7  int main(int argc, char* argv[]){
8
9    std::vector v = {1,3,5,7};
10   for(auto& elem : v)
11   {
12
13   }
14
15   return 0;
16 }
17
```

Insight:

```
1  #include <iostream>
2  #include <vector>
3
4  int main(int argc, char ** argv)
5  {
6    std::vector<int, std::allocator<int> > v = std::vector<int, std::allo
7    {
8      std::vector<int, std::allocator<int> > & __range1 = v;
9      std::__wrap_iter<int *> __begin1 = __range1.begin();
10     std::__wrap_iter<int *> __end1 = __range1.end();
11     for(; !std::operator==(__begin1, __end1); __begin1.operator++()) {
12       int & elem = __begin1.operator*();
13     }
14
15   }
16   return 0;
17 }
```

Aside: Structured binding also lowers code as well

- In this case, the std::pair is simply incremented
  - The initial pair is wherever the ‘begin’ iterator points
  - https://cppinsights.io/s/5b6d1ba8

```
// Build and run
// g++-14 -std=c++23 structured_binding.
#include <iostream>
#include <string>
#include <map>
#include <print>

int main(int argc, char* argv[]){
  using StringIntMap = std::map<std::str
  StringIntMap myMap {  {"Mike",123},
                        {"Bjarne", 234},
                        {"Walter", 345},
                     };
  // Iterators defined with a for-loop
  for(auto& [key,value] : myMap)
  {
    std::println("key: {} value: {}", ke
  }
  std::println();
  return 0;
}
```

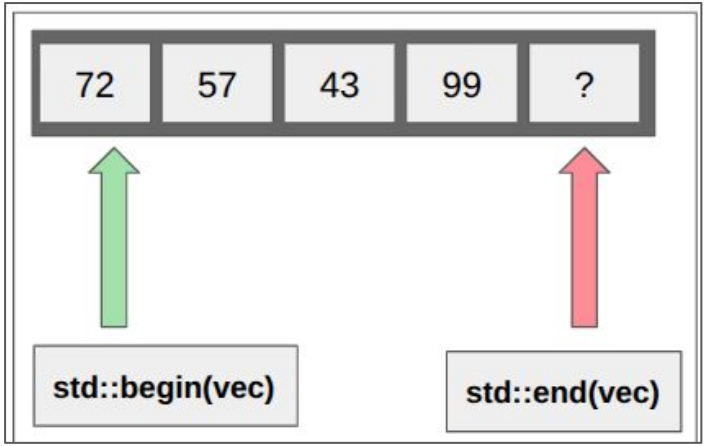
```
#include <iostream>
#include <string>
#include <map>
#include <print>

int main(int argc, char ** argv)
{
  using StringIntMap = std::map<std::basic_string<char, std::char_traits<char>, std:
  std::map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, i
  {
    std::map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >,
    std::__map_iterator<std::__tree_iterator<std::__value_type<std::basic_string<cha
    std::__map_iterator<std::__tree_iterator<std::__value_type<std::basic_string<cha
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
      std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<
      const std::basic_string<char, std::char_traits<char>, std::allocator<char> > &
      int & value = std::get<1UL>(__operator16);
      std::println(std::basic_format_string<char, const std::basic_string<char, std:
    }

  }
```

Quick Recap

Recap (1/2)

- I'm 100% certain everyone knows what a for-loop is
  - But we did see an interesting development of how we can use pointers in our iteration instead of indices (which are just pointer offsets anyway)
- We then generalized that idea of the pointer into an object called an **iterator**
  - (See links for writing your own that aren't part of the STL)

```
auto data = &message.data()[0]; // or equally 'message.data()';
auto ptr  = data;
auto end  = data + message.size(); // last element
for(/* empty */ ; ptr != end ; ++ptr){
  std::print("{}",*ptr);        // or std::cout << *ptr
}
std::println();
```



STL C++ Iterators - Writing an iterator from scratch | Modern Cpp Series Ep. 138
STL C++ Iterators - Writing an iterator from scratch (Code Review) | Modern Cpp Series Ep. 139

Recap: Benefits of Iterators (2/2)

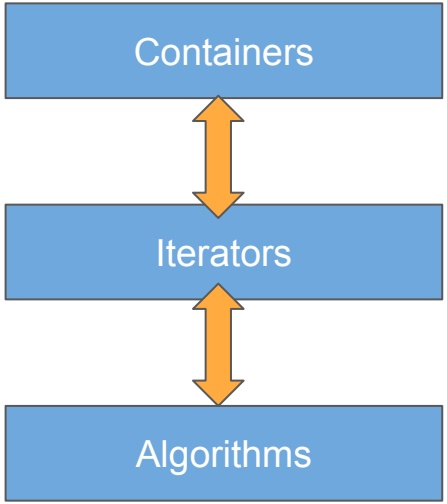
- **Iterators** are one of the behavioral design patterns:
  - The 'begin' and 'end' pair more clearly the intent to iterate through the entire collection from beginning to end.
  - Generally speaking it is easy to modify code, because the common interface for the iteration is otherwise the same (std::begin/std::end)
  - From a codegen perspective, usually equivalent (I can probably come up with a corner cases, but as always measure)
  - **(Coming up -- more benefits for your API)**
    - Using iterators is powerful because we decouple the 'algorithm' of how we iterate (in this example forward iteration) from our actual container data structure.
    - Decouples our traversal code from the container.

```
1  // iterator.cpp
2  // g++ -std=c++20 iterator.cpp -o prog
3  #include <iostream>
4  #include <vector>
5
6  int main(){
7
8      std::vector<int> collection {1,2,3};
9      collection.push_back(4);
10
11     // iterator
12     for(std::vector<int>::iterator it = collection.begin();
13                                     it != collection.end();
14                                   ++it)
15     {
16         std::cout << *it << std::endl;
17     }
18
19     return 0;
20 }
```

Standard Template Library (STL) (1/2)

The STL gave us 3 powerful components for writing code

1. **Containers** (where our data is held)
   a. We have seen already: std::vector, std::string, std::array, std::list, etc.
2. **Iterators** (Connect containers and algorithms)
   a. Abstraction for the 'starting point' and 'ending point' of our data
3. **Algorithms**
   a. Generic building blocks -- effectively replacing loops



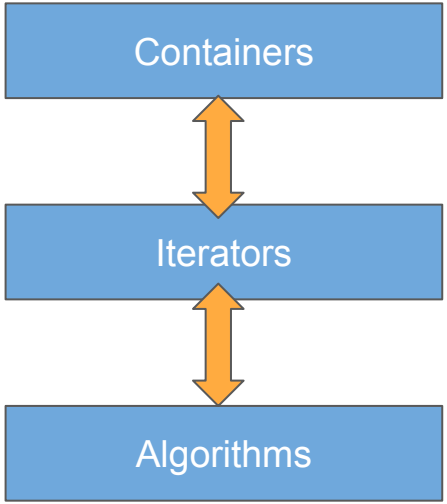
See these talks for more:
GoingNative 2013 C++ Seasoning - Sean Parent
Back to Basics: Classic STL - Bob Steagall - CppCon 2021
Standard Template Library (STL) Short Overview | Modern Cpp Series Ep. 111
Alex Stepanov, Generic Programming, and the C++ STL - Jon Kalb - C++Now 2025

Standard Template Library (STL) (2/2)

The STL gave
writing code

1. **Containers** (
   a. We have see
      std::list, etc.
2. **Iterators** (Connect containers and algorithms)
   a. Abstraction for the 'starting point' and 'ending point' of our data
3. **Algorithms**
   a. Generic building blocks -- effectively replacing loops

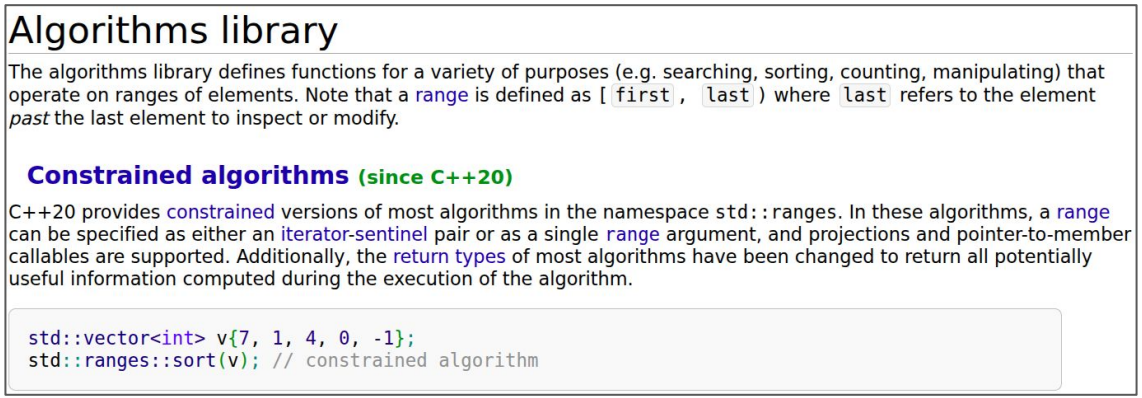
We have seen iterators with containers, now let's see them with std::algorithm



See these talks for more:
GoingNative 2013 C++ Seasoning - Sean Parent
Back to Basics: Classic STL - Bob Steagall - CppCon 2021
Alex Stepanov, Generic Programming, and the C++ STL - Jon Kalb - C++Now 2025

std::algorithm

C++ STL algorithm - Brief Introduction | Modern Cpp Series Ep. 141



Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as [`first` , `last`) where `last` refers to the element *past* the last element to inspect or modify.

Constrained algorithms (since C++20)

C++20 provides constrained versions of most algorithms in the namespace `std::ranges`. In these algorithms, a range can be specified as either an iterator-sentinel pair or as a single range argument, and projections and pointer-to-member callables are supported. Additionally, the return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm.

```
std::vector<int> v{7, 1, 4, 0, -1};
std::ranges::sort(v); // constrained algorithm
```

Iterators + Algorithms

- Besides iterators helping us access and ‘iterate’ through containers, they allow us to easily use ‘std::algorithm’ building blocks.
- In this example I use `std::sort` [1], which takes a pair of iterators (defining the range) from which we would like sorted.

See also
CppCon 2015: Michael VanLoon “STL Algorithms in Action ”
'STL Algorithms – How to Use Them and How to Write Your Own' - Marshall Clow [ACCU 2016]

```
1  // @file: algorithms.cpp
2  // Build and run
3  // g++-14 -std=c++23 algorithms.cpp -o prog && ./prog
4  #include <iostream>
5  #include <string>
6  #include <vector>
7  #include <print>
8
9  template <typename T>
10 void PrintCollection(const T& collection, std::string note=""){
11   println("{}",note);
12   for(const auto& elem : collection){
13     std::print("{} ",elem);
14   }
15   std::println();
16 }
17
18 int main(int argc, char* argv[]){
19   // (0) Initial data
20   std::vector<std::string> v{"B","C","D","G", "F", "E", "A"};
21
22   PrintCollection(v,"=== Before ==="); // (1) Before our work
23   std::sort(v.begin(),v.end());        // (2) Perform algoritm
24   PrintCollection(v,"=== After ===");  // (3) View results
25
26
27   return 0;
28 }
~
~
"algorithms.cpp" 28L, 689B written                    19,21         All
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 algorithms.cpp -o prog && .
/prog
=== Before ===
B C D G F E A
=== After ===
A B C D E F G
```

Iterators + Algorithms

- And there's all sorts of cool algorithms otherwise defined in <algorithm> and <numeric> headers.

Note: Yes, you can use also use std::print("{}",v); instead of 'PrintCollection'. Sorry for switching on some examples

```
1 // @file: algorithms.cpp
2 // Build and run
3 // g++-14 -std=c++23 algorithms.cpp -o prog && ./prog
4 #include <iostream>
5 #include <string>
6 #include <vector>
7 #include <print>
8
9 template <typename T>
10 void PrintCollection(const T& collection, std::string note=""){
11   println("{}",note);
12   for(const auto& elem : collection){
13     std::print("{} ",elem);
14   }
15   std::println();
16 }
17
18 int main(int argc, char* argv[]){
19   // (0) Initial data
20   std::vector<std::string> v{"B","C","D","G", "F", "E", "A"};
21
22   PrintCollection(v,"=== Before ==="); // (1) Before our work
23   std::sort(v.begin(),v.end());        // (2) Perform algoritm
24   PrintCollection(v,"=== After ===");  // (3) View results
25
26
27   return 0;
28 }
~
~
"algorithms.cpp" 28L, 689B written                    19,21         All
```

```
mike@mike-MS-7B17:ranges$ g++-14 -std=c++23 algorithms.cpp -o prog && .
/prog
=== Before ===
B C D G F E A
=== After ===
A B C D E F G
```

Iterators + Algorithms

- And there's all sorts of cool algorithms otherwise defined in <algorithm> and <numeric> headers.
  - Partitioning

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

int main(){

    std::vector v1{1,3,5,7};
    std::vector v2{1,7,5,3};

    auto predictate = [](int i){ return i < 4;};

    auto v1_test = std::is_partitioned(v1.begin(), v1.end(), predictate);
    auto v2_test = std::is_partitioned(v2.begin(), v2.end(), predictate);

    std::cout.setf(std::ios_base::boolalpha);
    std::cout << v1_test << std::endl;
    std::cout << v2_test << std::endl;


    auto it2 = std::partition(v2.begin(),v2.end(), predictate);
    for(auto elem: v2){
        std::cout << elem << " ";
    }
    std::cout << std::endl;
    v2_test = std::is_partitioned(v2.begin(), v2.end(), predictate);
    std::cout << v2_test << std::endl;

    return 0;
}
```

Iterators + Algorithms

- And there's all sorts of cool algorithms otherwise defined in <algorithm> and <numeric> headers.
  - copying
    - Observe the 'output' iterator 'back_inserter' that allocates in the destination collection
  - shuffling

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <random>
#include <list>

int main(){

    auto print_c = [](const auto& c, const char* msg){
        std::cout << msg << ":";
        for(auto elem: c){
            std::cout << elem << " ";
        }
        std::cout << std::endl;
    };

    std::random_device rd;
    std::mt19937 g(rd());

    std::vector<int> v{1,2,3,4};
    std::vector<int> original_v;
    std::copy(v.begin(),v.end(),std::back_inserter(original_v));

    for(size_t i=0; i < 10; i++){
        print_c(original_v,"original");
        print_c(v,"vector:");

        std::shuffle(v.begin(),v.end(),g);
        print_c(v,"shuffled:");
        std::cout << std::endl;
    }

    return 0;
}
```

Iterators + Algorithms

- And there's all sorts of cool algorithms otherwise defined in <algorithm> and <numeric> headers.
  - rotating
    - We are using 'std::upper_bound' within rotate to do a 'sort'
    - **Note:** This is a little bit ugly -- and std::ranges will help with the composition -- stay tuned!

```
1 #include <iostream>
2 #include <algorithm>
3 #include <vector>
4
5 int main(){
6
7     auto print_v = [](auto c,const char* msg){
8         std::cout << msg;
9         for(auto elem: c){
10            std::cout << elem << " ";
11        }
12        std::cout << std::endl;
13    };
14
15    std::vector v{2,1,4,3,5};
16    print_v(v,"before:");
17
18    for(auto i= v.begin(); i != v.end(); ++i){
19        std::rotate(std::upper_bound(v.begin(),i,*i),i,i+1);
20    }
21    print_v(v,"sorted:");
22
23
24    return 0;
25 }
```

Iterators + Algorithms

- And there's all sorts of cool algorithms otherwise defined in <algorithm> and <numeric> headers.
  - transforming

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

int main(){

    std::string s{"mike"};
    std::string out;

    // Sometimes we want to do things in-place, or
    // sometimes we want a copy of our data.
    std::transform(s.cbegin(),s.cend(),
                   std::back_inserter(out),
                   [](unsigned char c){
                        return std::toupper(c);
                   });

    std::cout << "s   :" << s << std::endl;
    std::cout << "out:" << out << std::endl;

    return 0;
}
```

Iterators + Algorithms

- And there's all sorts of cool algorithms otherwise defined in <algorithm> and <numeric> headers.
  - transforming reducing

```cpp
#include <iostream>
#include <vector>
#include <numeric>
#include <execution>

int main(){
    std::vector<int> v      {1,2,3,4,5,6,7,8};
    std::vector<int> result{3,3,3,3,3,3,3,3};
    // (1)
    std::cout << std::transform_reduce(v.begin(),v.end(),
                                       result.begin(),0)
      << std::endl;
    // (3)
    std::cout << std::transform_reduce(v.begin(),v.end(),
                                       0,std::plus<>(), [](int v){return v *= 3;}
)     << std::endl;
    // (4)
    std::cout << std::transform_reduce(std::execution::par,v.begin(),v.end(),
                                                                  result.begin(),0)
      << std::endl;
    // (6)
    std::cout << std::transform_reduce(std::execution::seq,v.begin(),v.end(),
                                                                  0,
                                                                  std::plus<>(),
                                                                  [](int v){ return v*=3
;}   << std::endl;
    return 0;
}
```

Iterators + Algorithms

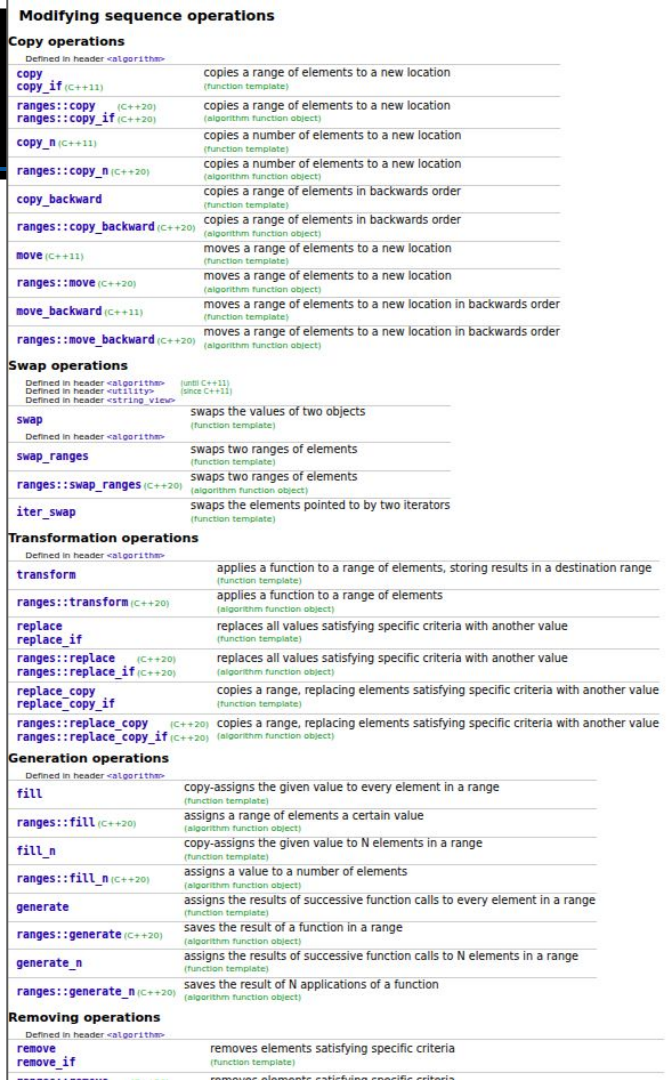
- And there's all sorts of cool algorithms otherwise defined in <algorithm> and <numeric> headers.
  - product

```
#include <iostream>
#include <numeric>
#include <array>
#include <functional>

bool greater(const int a, const int b){
    return a > b;
}


int main(){

    std::array<int,5> a{0,1,2,3,4};
    std::array        b{5,4,2,3,1};

    auto sum_of_products = std::inner_product(a.begin(),a.end(),b.begin(
),0);
    std::cout << "sum_of_products: " << sum_of_products << std::endl;

    int accumulator =0;
    for(size_t i=0; i != a.size(); ++i){
        accumulator += a[i]*b[i];
    }
    std::cout << "manual accumulator: " << accumulator << std::endl;

    auto sum_of_products2 = std::inner_product(a.begin(),a.end(),b.begin
(),0,
                                               std::plus<>(),std::multip
lies<>());
    std::cout << "sum_of_products2: " << sum_of_products2 << std::endl;

    auto matches = std::inner_product(a.begin(),a.end(),b.begin(),0,
                                      std::plus<>(), std::equal_to<>()
);
    std::cout << "matches: " << matches << std::endl;
"stl_inner_product/main.cpp" 46L, 1476B                    1,1           Top
```

Iterators + Algorithms

- ...you get the idea.
- Algorithms use iterators and operator through them
  - Reduces the number of ‘raw’ loops that we have to manually write.
- There are otherwise hundreds of these algorithms available (See link below for more)

[CppCon 2018: Jonathan Boccara “105 STL Algorithms in Less Than an Hour”]



Modifying sequence operations

Copy operations

Defined in header <algorithm>

copy copy_if (C++11)	copies a range of elements to a new location (function template)
ranges::copy (C++20) ranges::copy_if (C++20)	copies a range of elements to a new location (algorithm function object)
copy_n (C++11)	copies a number of elements to a new location (function template)
ranges::copy_n (C++20)	copies a number of elements to a new location (algorithm function object)
copy_backward	copies a range of elements in backwards order (function template)
ranges::copy_backward (C++20)	copies a range of elements in backwards order (algorithm function object)
move (C++11)	moves a range of elements to a new location (function template)
ranges::move (C++20)	moves a range of elements to a new location (algorithm function object)
move_backward (C++11)	moves a range of elements to a new location in backwards order (function template)
ranges::move_backward (C++20)	moves a range of elements to a new location in backwards order (algorithm function object)

Swap operations

Defined in header <algorithm> (until C++11)
Defined in header <utility> (since C++11)
Defined in header <string_view>

swap	swaps the values of two objects (function template)

Defined in header <algorithm>

swap_ranges	swaps two ranges of elements (function template)
ranges::swap_ranges (C++20)	swaps two ranges of elements (algorithm function object)
iter_swap	swaps the elements pointed to by two iterators (function template)

Transformation operations

Defined in header <algorithm>

transform	applies a function to a range of elements, storing results in a destination range (function template)
ranges::transform (C++20)	applies a function to a range of elements (algorithm function object)
replace replace_if	replaces all values satisfying specific criteria with another value (function template)
ranges::replace (C++20) ranges::replace_if (C++20)	replaces all values satisfying specific criteria with another value (algorithm function object)
replace_copy replace_copy_if	copies a range, replacing elements satisfying specific criteria with another value (function template)
ranges::replace_copy (C++20) ranges::replace_copy_if (C++20)	copies a range, replacing elements satisfying specific criteria with another value (algorithm function object)

Generation operations

Defined in header <algorithm>

fill	copy-assigns the given value to every element in a range (function template)
ranges::fill (C++20)	assigns a range of elements a certain value (algorithm function object)
fill_n	copy-assigns the given value to N elements in a range (function template)
ranges::fill_n (C++20)	assigns a value to a number of elements (algorithm function object)
generate	assigns the results of successive function calls to every element in a range (function template)
ranges::generate (C++20)	saves the result of a function in a range (algorithm function object)
generate_n	assigns the results of successive function calls to N elements in a range (function template)
ranges::generate_n (C++20)	saves the result of N applications of a function (algorithm function object)

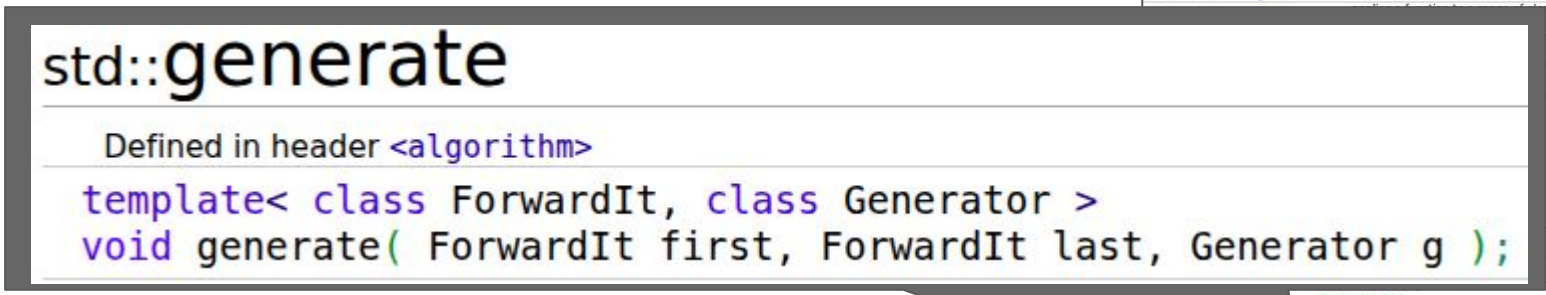
Removing operations

Defined in header <algorithm>

remove remove_if	removes elements satisfying specific criteria (function template)

Iterators + Algorithms

- Pretty much every std::algorithm prior to C++20 took a pair of iterators and then some other predicate or function
  - You can study some previous examples to see usage.



Modifying sequence operations

Copy operations

Defined in header <algorithm>

copy copy_if (C++11)	copies a range of elements to a new location (function template)
ranges::copy (C++20) ranges::copy_if (C++20)	copies a range of elements to a new location (algorithm function object)
copy_n (C++11)	copies a number of elements to a new location (function template)
ranges::copy_n (C++20)	copies a number of elements to a new location (algorithm function object)
copy_backward	copies a range of elements in backwards order (function template)
ranges::copy_backward (C++20)	copies a range of elements in backwards order (algorithm function object)
move (C++11)	moves a range of elements to a new location (function template)
ranges::move (C++20)	moves a range of elements to a new location (algorithm function object)
move_backward (C++11)	moves a range of elements to a new location in backwards order (function template)
ranges::move_backward (C++20)	moves a range of elements to a new location in backwards order (algorithm function object)

Swap operations

Defined in header <algorithm> (until C++11)
Defined in header <utility> (since C++11)
Defined in header <string_view>

swap	swaps the values of two objects (function template)

Defined in header <algorithm>

swap_ranges	swaps two ranges of elements (function template)
ranges::swap_ranges (C++20)	swaps two ranges of elements (algorithm function object)
iter_swap	swaps the elements pointed to by two iterators (function template)

Transformation operations

Defined in header <algorithm>

	ments, storing results in a destination range
	ments
	ic criteria with another value
	ic criteria with another value
	satisfying specific criteria with another value
	satisfying specific criteria with another value
	ment in a range
	ue
	ts in a range (function template)
	assigns a value to a number of elements (algorithm function object)
	assigns the results of successive function calls to every element in a range (function template)
	saves the result of a function in a range (algorithm function object)
generate_n	assigns the results of successive function calls to N elements in a range (function template)
ranges::generate_n (C++20)	saves the result of N applications of a function (algorithm function object)

Removing operations

Defined in header <algorithm>

remove remove_if	removes elements satisfying specific criteria (function template)

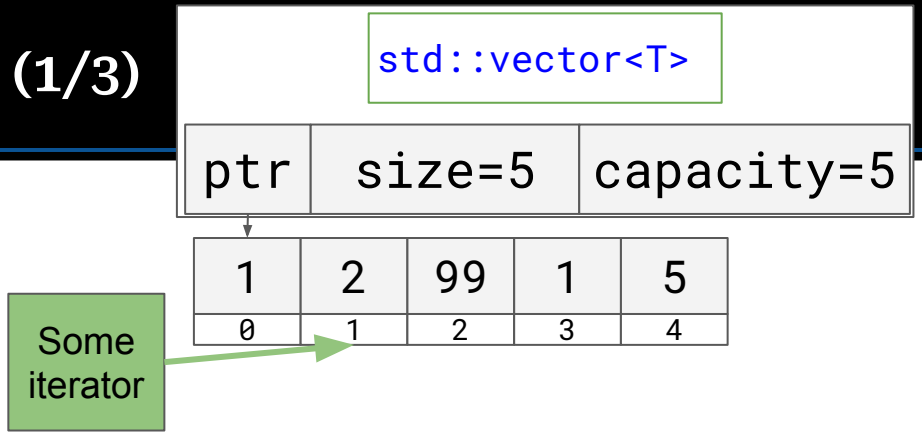
Problems/Pitfalls with Iterators

https://en.cppreference.com/w/cpp/iterator.html

Iterators are Handy

- And we still need them, even in C++26
  - The idea is good -- but there are a few ‘pitfalls’ to worry about

Pitfall #1 - Iterator Invalidation (1/3)



- Consider after we insert a new element in a vector, we may have a ‘reallocation’ to a new block of memory.
  - Any previous iterators are considered ‘invalid’ as they do not point to the current vectors allocated memory
  - e.g.
    - (next slide)

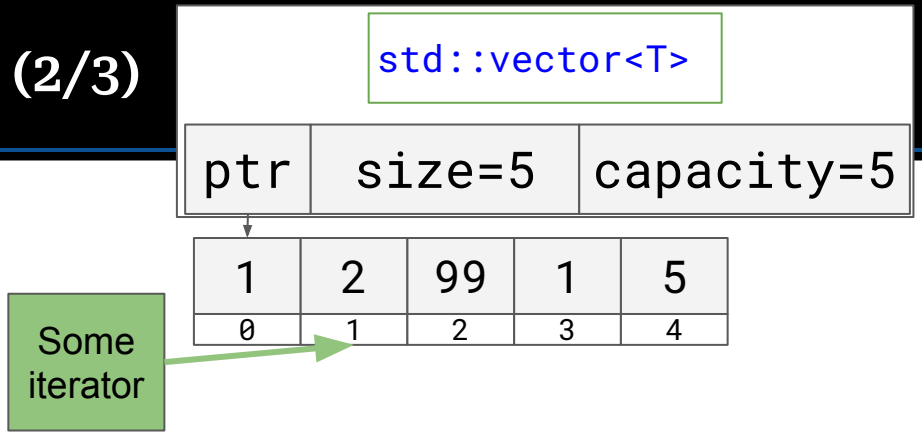
Iterator invalidation

Read-only methods never invalidate iterators or references. Methods which modify the contents of a container may invalidate iterators and/or references, as summarized in this table.

Category	Container	After **insertion**, are...		After **erasure**, are...		Conditionally
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s) (for insertion only if capacity didn't change)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
		No		No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		
Associative containers	`set` `multiset` `map` `multimap`	Yes		Yes, except erased element(s)		
Unordered associative containers	`unordered_set` `unordered_multiset` `unordered_map` `unordered_multimap`	No	Yes	N/A		Insertion caused rehash
		Yes		Yes, except erased element(s)		No rehash

https://en.cppreference.com/w/cpp/container

Pitfall #1 - Iterator Invalidation (2/3)



- Consider after we insert a new element in a vector, we may have a ‘reallocation’ to a new block of memory.
  - Any previous iterators are considered ‘invalid’ as they do not point to the current vectors allocated memory
  - e.g.
    - `push_back(7)`
      - Let’s assume this forces a new allocation

Iterator invalidation

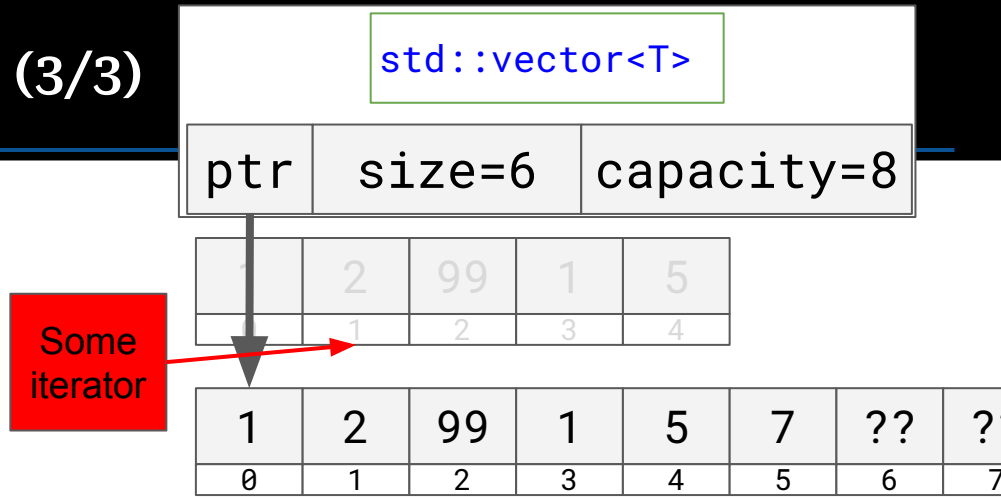
Read-only methods never invalidate iterators or references. Methods which modify the contents of a container may invalidate iterators and/or references, as summarized in this table.

Category	Container	After **insertion**, are... iterators valid?	After **insertion**, are... references valid?	After **erasure**, are... iterators valid?	After **erasure**, are... references valid?	Conditionally
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s) (for insertion only if capacity didn't change)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
		No		No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		
Associative containers	`set` `multiset` `map` `multimap`	Yes		Yes, except erased element(s)		
Unordered associative containers	`unordered_set` `unordered_multiset` `unordered_map` `unordered_multimap`	No	Yes	N/A		Insertion caused rehash
		Yes		Yes, except erased element(s)		No rehash

https://en.cppreference.com/w/cpp/container

Pitfall #1 - Iterator Invalidation (3/3)

- Consider after we insert a new element in a vector, we may have a ‘reallocation’ to a new block of memory.
  - Any previous iterators are considered ‘invalid’ as they do not point to the current vectors allocated memory
  - e.g.
    - `push_back(7)`
      - Let’s assume this forces a new allocation
    - Observe any iterators point to the old data -- thus invalidated.
    - Note: Safe iterator libraries abstractions can catch some of these bugs



Iterator invalidation

Read-only methods never invalidate iterators or references. Methods which modify the contents of a container may invalidate iterators and/or references, as summarized in this table.

Category	Container	After **insertion**, are... iterators valid?	After **insertion**, are... references valid?	After **erasure**, are... iterators valid?	After **erasure**, are... references valid?	Conditionally
Sequence containers	array	N/A		N/A		
	vector	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s) (for insertion only if capacity didn't change)
		No		No		At or after modified element(s)
	deque	No	Yes	Yes, except erased element(s)		Modified first or last element
		No		No		Modified middle only
	list	Yes		Yes, except erased element(s)		
	forward_list	Yes		Yes, except erased element(s)		
Associative containers	set multiset map multimap	Yes		Yes, except erased element(s)		
Unordered associative containers	unordered_set unordered_multiset unordered_map unordered_multimap	No	Yes	N/A		Insertion caused rehash
		Yes		Yes, except erased element(s)		No rehash

https://en.cppreference.com/w/cpp/container

Pitfall #2 - Wrong Pair of Iterators (1/2)

- Wrong pair of iterators
  - This will not throw a compile-time error, but you'll eventually crash!

```
13 int main(int argc, char* argv[]){
14
15
16     std::vector v1{1,3,5,7,9};
17     std::vector v2{1,3,5,7,9};
18
19     std::sort(v1.begin(),v2.end());
20
21
22     return 0;
23 }
```

Pitfall #2 - Wrong Pair of Iterators (2/2)

- Not always easy to catch however the ‘wrong pairs’ either
- This ‘works’ -- but it’s not necessarily true that the begin/end pair of iterators match
  - Think carefully here about if a ‘function call’ that returns a new vector each time has matching iterators
  - Ans: Nope!

```
 8 std::vector<int> DownloadData() {
 9     std::vector<int> result{1,3,5,7};
10     return result;
11 }
12
13 int main(int argc, char* argv[]){
14
15
16     auto results = std::all_of(DownloadData().begin(),DownloadData().end(),
17                                [](int i) { return i > 2;});
18
19     std::println("{}",results);
20
21     return 0;
22 }
```

Based off of: C++ Weekly - Ep 342 - C++20's Ranges: A Quick Start
https://youtu.be/sZy9XcGHmI4?si=llzZAoNS7wr2Wz3s&t=36

Pitfall #3 - Consistency

- Take care that the first two parameters are not always ‘begin’ and ‘end’
  - This can sometimes cause problems, you just have to pay attention to.

```
ForwardIt rotate( ForwardIt first, ForwardIt middle, ForwardIt last );
```

Pitfall #4 - Not enough space to write into

- Your output (d_first) needs enough memory allocated otherwise
  - Sometimes this is not a problem if you use an expanding data structure (e.g. a std::vector with back_inserter), but if you are outputting into a fixed-container this could cause an overflow/segmentation fault!

```
OutputIt transform( InputIt first1, InputIt last1,
                    OutputIt d_first, UnaryOp unary_op );
```

So Iterators are not perfect

- Iterators Must Go
  - Andrei Alexandrescu
  - https://accu.org/conf-docs/PDFs_2009/AndreiAlexandrescu_iterators-must-go.pdf
- Or otherwise -- can we improve on iterators?
- The good news is -- we can!

Final nail in the coffin

- *All* iterator primitives are fundamentally unsafe
- For most iterator types, given an iterator
  - Can't say whether it can be compared
  - Can't say whether it can be incremented
  - Can't say whether it can be dereferenced
- Safe iterators can and have been written
  - At a high size+speed cost
  - Mostly a good argument that the design hasn't been cut quite right

I'm not sure where the 'video' is of this talk, but the slides are otherwise linked. Thanks to anyone who helps me find Andrei's 2009 talk again!
There is otherwise a 2014 Boostcon (C++Now) talk 'iterators must stay' by Sebastian Redl

std::ranges

Improving our interface to std::algorithm building blocks

Ranges library (since C++20)

The ranges library is an extension and generalization of the algorithms and iterator libraries that makes them more 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

- `[begin, end)` – iterator pairs, e.g. ranges made by implicit conversion from containers. All algorithms that take iterator pairs now have overloads that accept ranges (e.g. `ranges::sort`)
- `begin + [0, size)` – counted sequences, e.g. range returned by `views::counted`
- `[begin, predicate)` – conditionally-terminated sequences, e.g. range returned by `views::take_while`
- `[begin, ..)` – unbounded sequences, e.g. range returned by `views::iota`

The ranges library includes range algorithms, which are applied to ranges eagerly, and range adaptors, which are applied to views lazily. Adaptors can be composed into pipelines, so that their actions take place as the view is iterated.

Defined in header <ranges>

```
namespace std {
    namespace views = ranges::views;    (since C++20)
}
```

What is a Range (#include <ranges>)

- Same thing we learned before
  - A range is a pair of iterators
  - The pair is represented by a ‘start’ and a ‘sentinel’ value which better defines the end of the range
    - Note: there’s a bit more flexibility in that the types don’t have to be the same however.
    - empty range
      - start and end point to the same ‘location’

```
template<typename T>
struct range{
    T start;
    T sentinel;
};
```

What is a Range (#include <ranges>)

- In C++20, we also got concepts, which helped make possible the ranges API
  - A range thus can be defined with a concept that has a ‘begin’ and ‘end’
- Note:
  - All of the range algorithms we will learn about are also written with concepts
  - This typically means we’ll get as good or better error messaging than before

```
template<typename T>
concept range = requires(T& t)
{
    ranges::begin(t);
    ranges::end(t);
};
```

Ranges Use (1/4)

- Using ranges is as simple as:
  - #include <ranges>
    - New header added in C++ 20 for ranges and views
- Namespaces:
  - std::ranges
    - Contains many of the range algorithms
  - std::ranges::views
    - Range adaptors that turn ranges into views (more on this)

Sorting operations

Defined in header <algorithm>

sort

ranges::sort (C++20)

stable_sort

ranges::stable_sort (C++20)

partial_sort

ranges::partial_sort (C++20)

partial_sort_copy

ranges::partial_sort_copy (C++20)

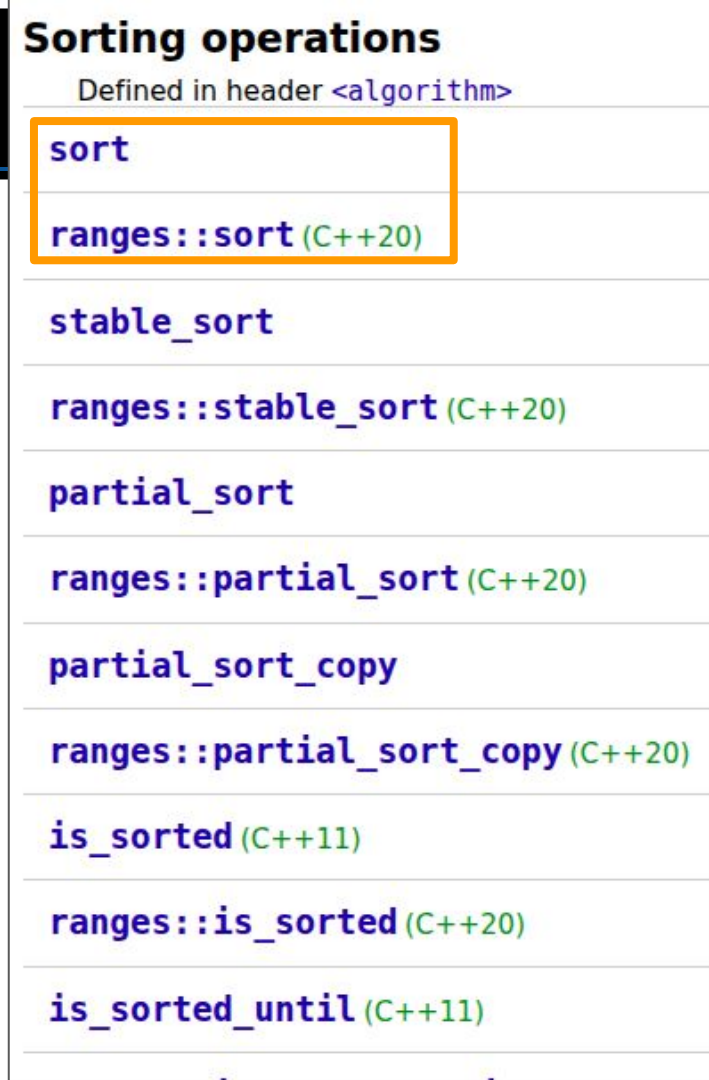
is_sorted (C++11)

ranges::is_sorted (C++20)

is_sorted_until (C++11)

Ranges Use (2/4)

- Observe most STL algorithms have a range version now



Sorting operations

Defined in header <algorithm>

- sort
- ranges::sort (C++20)
- stable_sort
- ranges::stable_sort (C++20)
- partial_sort
- ranges::partial_sort (C++20)
- partial_sort_copy
- ranges::partial_sort_copy (C++20)
- is_sorted (C++11)
- ranges::is_sorted (C++20)
- is_sorted_until (C++11)

Ranges Use (3/4)

- Range version takes 1 less parameter
  - iterator based version otherwise still there for when you need finer grain control
  - std::ranges sort assumes (begin to end) when passing container
    - (We'll learn about views in a moment to adjust begin/end of range otherwise)

```
// ========================================
// Prior to Ranges
std::vector v1{3,9,7,5,1};
std::sort(std::begin(v1), std::end(v1));
std::println("{}",v1);

// With ranges
std::vector v2{3,9,7,5,1};
std::ranges::sort(v2);
std::println("{}",v2);
// ========================================
```

Sorting operations

Defined in header <algorithm>

sort

ranges::sort (C++20)

stable_sort

ranges::stable_sort (C++20)

partial_sort

ranges::partial_sort (C++20)

partial_sort_copy

ranges::partial_sort_copy (C++20)

is_sorted (C++11)

ranges::is_sorted (C++20)

is_sorted_until (C++11)

Ranges Use (4/4)

- Note:
  - You may here the ‘ranges’ versions called ‘constrained algorithms’

```
// ========================================
// Prior to Ranges
std::vector v1{3,9,7,5,1};
std::sort(std::begin(v1), std::end(v1));
std::println("{}",v1);

// With ranges
std::vector v2{3,9,7,5,1};
std::ranges::sort(v2);
std::println("{}",v2);
// ========================================
```

Sorting operations

Defined in header <algorithm>

- sort
- ranges::sort (C++20)
- stable_sort
- ranges::stable_sort (C++20)
- partial_sort
- ranges::partial_sort (C++20)
- partial_sort_copy
- ranges::partial_sort_copy (C++20)
- is_sorted (C++11)
- ranges::is_sorted (C++20)
- is_sorted_until (C++11)

Iterators and ranges (1/3)

- Observe even with ranges however, sometimes we still return an iterator
  - In both examples (top one highlighted) I show that find_if returns an iterator to the result.

```cpp
// ================ We still use iterators with ranges ==========
// Iterators don't go completely away, we still use them, and we build
// ranges on top of iterators.
std::vector data{3,9,7,5,1};
auto iter = std::ranges::find_if(data, [](int i){ return i==1; });
if( iter != std::ranges::end(data)){
    std::cout << "I found 1!!!" << *iter << std::endl;
}

std::string dna = "GCTCATC";
auto dna_iter=std::ranges::find(dna,'T');
if( dna_iter != std::ranges::end(dna)){
    std::cout << "I found T!!!" << *dna_iter << std::endl;
}
```

Iterators and ranges (2/3)

- Note:
  - We still have std::ranges::end just like we had std::end
  - And we have the equivalent ‘begin’ version as well
- So it’s important not to forget about our iterator knowledge from before
  - std::ranges is again, a pair of iterators behind the scenes.

```
// ================ We still use iterators with ranges ==========
// Iterators don't go completely away, we still use them, and we build
// ranges on top of iterators.
std::vector data{3,9,7,5,1};
auto iter = std::ranges::find_if(data, [](int i){ return i==1; });
if( iter != std::ranges::end(data)){
    std::cout << "I found 1!!!" << *iter << std::endl;
}

std::string dna = "GCTCATC";
auto dna_iter=std::ranges::find(dna,'T');
if( dna_iter != std::ranges::end(dna)){
    std::cout << "I found T!!!" << *dna_iter << std::endl;
}
```

Iterators and ranges (3/3)

- Here's an 'old style' for-loop with the ranges::begin and ranges::end variant just to show you it works
- This flexibility means ranges work effectively with:
  - All of the containers (array, vector, map, etc.) are ranges
  - All of the container adaptors (queue, stack, etc.) are ranges
  - And 'non-owning' or 'borrowed' containers like (std::string_view, std::span, are 'borrowed ranges'.)

```
// ================ std::ranges::begin and end ==========
std::vector old{1,2,4,8,12,14};
for(auto start = std::ranges::begin(old);
         start != std::ranges::end(old);
         start++){
    std::print("{} ",*start);
}
std::println();
```

std::ranges::views

std::ranges::view, std::ranges::enable_view, std::ranges::view_base

Defined in header <ranges>

```
template<class T>
concept view = ranges::range<T> && std::movable<T> && ranges::enable_view<T>;
```
(1) (since C++20)

```
template<class T>
constexpr bool enable_view =
    std::derived_from<T, view_base> || /*is-derived-from-view-interface*/<T>;
```
(2) (since C++20)

```
struct view_base { };
```
(3) (since C++20)

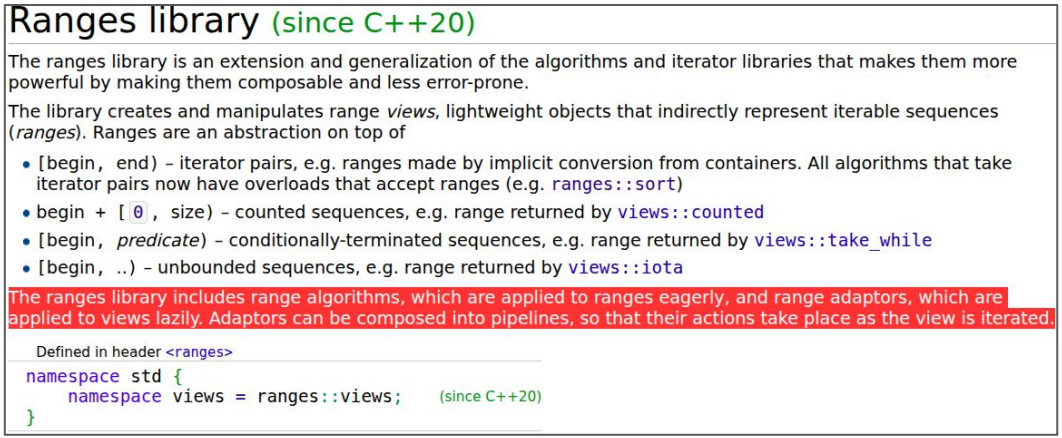
1) The `view` concept specifies the requirements of a `range` type that has suitable semantic properties for use in constructing range adaptor pipelines.

2) The `enable_view` variable template is used to indicate whether a `range` is a `view`.
`/*is-derived-from-view-interface*/<T>` is `true` if and only if T has exactly one public base class `ranges::view_interface<U>` for some type U, and T has no base classes of type `ranges::view_interface<V>` for any other type V.
Users may specialize `enable_view` to `true` for cv-unqualified program-defined types which model `view`, and `false` for types which do not. Such specializations must be usable in constant expressions and have type `const bool`.

3) Deriving from `view_base` enables `range` types to model `view`.

Hmm, so what can we use with ranges then?

- So we previously saw some range algorithms (e.g. std::ranges::sort)
  - These were simply the std::algorithms that takes a range as an input instead of a pair of iterators
- Now I want to draw your attention to **views** and **adaptors**
  - Range Adaptors turn ranges into ‘views’
  - Range Views (std::ranges::views) are ‘lazy’ ways to access items in a range
    - This means they’re cheap to copy and work with



Ranges library (since C++20)

The ranges library is an extension and generalization of the algorithms and iterator libraries that makes them more 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

- [begin, end) – iterator pairs, e.g. ranges made by implicit conversion from containers. All algorithms that take iterator pairs now have overloads that accept ranges (e.g. `ranges::sort`)
- begin + [`0`, size) – counted sequences, e.g. range returned by `views::counted`
- [begin, *predicate*) – conditionally-terminated sequences, e.g. range returned by `views::take_while`
- [begin, ..) – unbounded sequences, e.g. range returned by `views::iota`

The ranges library includes range algorithms, which are applied to ranges eagerly, and range adaptors, which are applied to views lazily. Adaptors can be composed into pipelines, so that their actions take place as the view is iterated.

Defined in header <ranges>

```
namespace std {
    namespace views = ranges::views;    (since C++20)
}
```

Eager versus Lazy Execution (1/2)

- Range algorithms can be ‘eagerly evaluated’ which means to execute immediately

```
// ========================================
// Prior to Ranges
std::vector v1{3,9,7,5,1};
std::sort(std::begin(v1), std::end(v1));
std::println("{}",v1);

// With ranges
std::vector v2{3,9,7,5,1};
std::ranges::sort(v2);
std::println("{}",v2);
// ========================================
```

Eager Evaluation

Eager versus Lazy Execution (2/2)

- But Range Views can be executed lazily
  - Roughly speaking, copy & assignment takes place in O(1)
  - This allows for infinite ranges (because we could keep adding to ‘v4’ in this example)
  - The std::views allow for composition (with the pipe ‘|’ syntax)
    - A much more ‘functional style’ if you’ve used functional languages or Python, D, etc.

```
// ========================================
// Range Adaptors -- to transform ranges to views
std::vector v4{1,1,1,1,5,6,7,8,9,10,11};
auto greaterThanFive = v4 | std::views::filter([](int i){return i > 5;});
for(auto& elem : greaterThanFive){
    std::cout << elem << " ";
}
std::cout << std::endl;
// ========================================
```

Lazily Evaluate each iteration of loop

Composition with Range views (1/2)

- Here's another example showing the pipe '|' syntax to otherwise compose two filters together
  - No work is computed ahead of time, only as we evaluate each element each iteration of the loop

```
// ======================================
// Composition example
std::vector<std::string> strings = {"Mike","Bob","Miguel","Marissa","Mary"};
auto startsWithM = [](std::string s){ return s[0] == 'M'; };
auto longerThan4 = [](std::string s){ return s.length() > 4; };

// Little exercise here
// How many pieces of data do we operate on here?
for(auto& elem : strings
                | std::views::filter(startsWithM)
                | std::views::filter(longerThan4)
){
    std::cout << elem << " ";
}
std::cout << std::endl;
// ======================================
```

Composition with Range views (2/2)

- Note: Because of the the lazy nature of the algorithm, you may want to consider which ‘view’ to ‘test’ first.
  - This could be based on the ‘expensiveness’ of the operation, or otherwise which filter you think will result in the minimum operations.

```
// ========================================
// Composition example
std::vector<std::string> strings = {"Mike","Bob","Miguel","Marissa","Mary"};
auto startsWithM = [](std::string s){ return s[0] == 'M'; };
auto longerThan4 = [](std::string s){ return s.length() > 4; };

// Little exercise here
// How many pieces of data do we operate on here?
for(auto& elem : strings
                | std::views::filter(startsWithM)
                | std::views::filter(longerThan4)
){
    std::cout << elem << " ";
}
std::cout << std::endl;
// ========================================
```

ranges::to (and composition) (2/2)

- Sometimes we may want to ‘save the result’ into a collection
  - (This is similar to where we used ‘back_inserter’ for output iterators)
- For that purpose we have `std::ranges::to`
  - This writes out the value (in this case, a push_back) to the vector of strings

```
// ====================================
// ranges::to
std::vector<std::string> strings2 = {"mike","bob","miguel","marissa","mary"};
auto startsWithm = [](std::string s){ return s[0] == 'm'; };
auto longerThan3 = [](std::string s){ return s.length() > 3; };

// Little exercise here
// How many pieces of data do we operate on here?
std::vector<std::string> collect =  strings2
                        | std::views::filter(startsWithm)
                        | std::views::filter(longerThan3)
                        | std::ranges::to<std::vector>();
std::println("collected: {}",collect);
// ====================================
```

ranges::to (and composition) (2/2)

- So again -- this is cool, because I find it's easy to 'build containers' using the '|' operator.

```
// ====================================
// ranges::to
std::vector<std::string> strings2 = {"mike","bob","miguel","marissa","mary"};
auto startsWithm = [](std::string s){ return s[0] == 'm'; };
auto longerThan3 = [](std::string s){ return s.length() > 3; };

// Little exercise here
// How many pieces of data do we operate on here?
std::vector<std::string> collect =  strings2
                        | std::views::filter(startsWithm)
                        | std::views::filter(longerThan3)
                        | std::ranges::to<std::vector>();
std::println("collected: {}",collect);
// ====================================
```

More examples - std::ranges::transform

- Another example this time with transform

```
// ========================================
// Another Composition example with Transform
std::vector v5{1,3,5,7,11,13,17,19};
auto doubleValue = [](int i){ return i*2; };
auto AddOne = [](int i){ return i+1; };

// Notice this time no 'auto&' -- need the 'l-value'
for(auto elem : v5
                | std::views::transform(doubleValue)
                | std::views::transform(AddOne)

){
    std::cout << elem << " ";
}
std::cout << std::endl;
// ========================================
```

More examples - std::ranges::views::drop_while

- Some other view algorithms to help us filter where to start our execution (in this case a ‘print’) with a for_each

```
// ======================================
// Here's an example of otherwise using a 'view' to
// sort of filter our computation.
std::vector<std::string> winners = {"Mike","Bob","Miguel","Marissa","Mary"};

auto removeMike = [](std::string s){ return s == "Mike";};
auto printSpace= [](std::string s){ std::cout << s << " "; };
auto startingPoint = std::ranges::views::drop_while(winners, removeMike);

std::ranges::for_each(startingPoint, printSpace);

// ======================================
```

Many std::ranges and std::ranges::views

- Because of their composability, I find ranges very powerful
- There is further power you can have, in which I'll recommend some resources to check out
  - Meaning:
    - How to enforce constraints on your range types in functions
    - Why ranges you'll see universal references everywhere
      - (Need to be careful of ownership)
    - Some of the std::ranges::views are meant to be 'used once' so to speak, and others 'cache' results.
      - Make sure to read the cppreference page

ranges::elements_view (C++20)
views::elements

ranges::keys_view (C++20)
views::keys

ranges::values_view (C++20)
views::values

ranges::enumerate_view (C++23)
views::enumerate

ranges::zip_view (C++23)
views::zip

ranges::zip_transform_view (C++23)
views::zip_transform

ranges::adjacent_view (C++23)
views::adjacent

ranges::adjacent_transform_view (C++23)
views::adjacent_transform

ranges::chunk_view (C++23)
views::chunk

ranges::slide_view (C++23)
views::slide

ranges::chunk_by_view (C++23)
views::chunk_by

ranges::stride_view (C++23)
views::stride

ranges::cartesian_product_view (C++23)
views::cartesian_product

ranges::cache_latest_view (C++26)
views::cache_latest

Wrap Up and More Resources

More Range and STL Talks

- Listed Roughly in the order I would watch them
- Introductory Ranges Talks
  - Back to Basics: Classic STL - Bob Steagall - CppCon 2021
    - https://www.youtube.com/watch?v=tXUXl_RzkAk
  - Back to Basics: (Range) Algorithms in C++ - Klaus Iglberger - CppCon 2023
    - https://www.youtube.com/watch?v=eJCA2fynzME
  - Back to Basics: Iterators in C++ - Nicolai Josuttis - CppCon 2023
    - https://www.youtube.com/watch?v=26aW6aBVpk0
  - Effective Ranges: A Tutorial for Using C++2x Ranges - Jeff Garland - CppCon 2023
    - https://www.youtube.com/watch?v=QoaVRQvA6hI
  - What Is a Range in C++? - Šimon Tóth - C++ on Sea 2024
    - https://www.youtube.com/watch?v=UsyA7zEC0DE
  - Iterators and Ranges: Comparing C++ to D to Rust - Barry Revzin - [CppNow 2021]
    - https://www.youtube.com/watch?v=d3qY4dZ2r4w

Summary

- Use Ranges, and prefer ranges for their composability and safety if you have access to C++ 20
  - Better error messages and interfaces with concepts support this
- Still important to understand basics of iterators otherwise
- But if you are already using std::algorithm, you should have no problem adjusting to std::ranges going forward
  - C++23, C++26, and beyond are continuing to add more range capabilities and functions.

Additional Notes if Time

- Why use lambdas everywhere?
  - Probably better than a function pointer
  - Easier for compiler to inline versus a pointer where that may not be the case
- How to debug views? (and debuggability of ranges)
  - Honestly, I usually just take out one ‘pipe’ at a time and try to track what is going on
  - You can use GDB to otherwise make ‘calls’ if you want to preview what a subset of the data may look like.
    - Stepping through in debugger otherwise helpful for lazy ranges, because you can see what is going on.
- Still lots of code generated with ranges, but seems to be as performant as otherwise regular for-loop

Questions/Clarifications after the talk

- Do views make a ‘copy of my ‘range/container/whatever was passed in’
  - No
  - Views are fast to create, they otherwise do not ‘own’ or make a copy of the container.
  - Changes made otherwise modify the container
- Do views own elements
  - No, think of them again sort of like std::span or std::string_view
  - This again connects with the first point that views don’t ‘own’ and this makes them fast and easy to compose -- no performance penalty
    - https://learn.microsoft.com/en-us/cpp/standard-library/view-classes?view=msvc-170
- When I use std::ranges::sort do I have to sort the whole thing?
  - No. I passed in the container (the whole vector) which implies the whole thing, but I could have passed a view that was the middle elements of the vector.
- Can I apply my own custom sorting algorithm as a parameter in std::ranges::sort
  - Not so much in terms of std::ranges::sort having a template parameter for algorithm to run.
    - Certainly you can change the comparison operator (‘comp’)
    - You can also control what field in a struct for example you sort on (that’s the ‘proj’)
  - That said, you can still use ‘views’ and write your algorithm with a for-loop.
    - See the building blocks like std::ranges::partition and std::ranges::upper_bound as good helpers
- Other notes
  - Careful of using views ‘again’ after modifying the container.
  - I tend to think of most views as ‘single shot’ -- some other views may cache stale data, so just be aware.

Thank you CppCon 2025!

Back To Basics
Ranges

MIKE SHAH

Web: mshah.io
YouTube www.youtube.com/c/MikeShah
Social: mikeshah.bsky.social
Courses: courses.mshah.io
Talks: http://tinyurl.com/mike-talks

60 minutes | Audience: Beginner
9:00am - 10:00 Wed, Sept. 17, 2025

Thank you!