# 6.17 — Introduction to iterators

BY ALEX ON DECEMBER 17TH, 2019 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Iterating through an array (or other structure) of data is quite a common thing to do in programming. And so far, we've covered many different ways to do so: with loops and an index (`for-loops` and `while loops`), with pointers and pointer arithmetic, and with `range-based for-loops`:

```
#include <array>
#include <iostream>

int main()
{
  // The type is automatically deduced to std::array<int, 7> (Requires C++17).
  // Use the type std::array<int, 7> if your compiler doesn't support C++17.
  std::array data{ 0, 1, 2, 3, 4, 5, 6 };
  std::size_t length{ std::size(data) };

  // while-loop with explicit index
  std::size_t index{ 0 };
  while (index != length)
  {
    std::cout << data[index] << ' ';
    ++index;
  }
  std::cout << '\n';

  // for-loop with explicit index
  for (index = 0; index < length; ++index)
  {
    std::cout << data[index] << ' ';
  }
  std::cout << '\n';

  // for-loop with pointer (Note: ptr can't be const, because we increment it)
  for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
  {
    std::cout << *ptr << ' ';
  }
  std::cout << '\n';

  // ranged-based for loop
  for (int i : data)
  {
    std::cout << i << ' ';
  }
  std::cout << '\n';

  return 0;
}
```

Looping using indexes is more typing than needed if we only use the index to access elements. It also only works if the container (e.g. the array) provides direct access to elements (which arrays do, but some other types of containers, such as lists, do not).

Looping with pointers and pointer arithmetic is verbose, and can be confusing to readers who don't know the rules of pointer arithmetic. Pointer arithmetic also only works if elements are consecutive in memory (which is true for arrays, but not true for other types of containers, such as lists, trees, and maps).

**For advanced readers**

Pointers (without pointer arithmetic) can also be used to iterate through some non-sequential structures. In a linked list, each element is connected to the prior element by a pointer. We can iterate through the list by following the chain of pointers.

Range-based for-loops are a little more interesting, as the mechanism for iterating through our container is hidden -- and yet, they still work for all kinds of different structures (arrays, lists, trees, maps, etc...). How do these work? They use iterators.

## Iterators

An **iterator** is an object designed to traverse through a container (e.g. the values in an array, or the characters in a string), providing access to each element along the way.

A container may provide different kinds of iterators. For example, an array container might offer a forwards iterator that walks through the array in forward order, and a reverse iterator that walks through the array in reverse order.

Once the appropriate type of iterator is created, the programmer can then use the interface provided by the iterator to traverse and access elements without having to worry about what kind of traversal is being done or how the data is being stored in the container. And because C++ iterators typically use the same interface for traversal (operator++ to move to the next element) and access (operator* to access the current element), we can iterate through a wide variety of different container types using a consistent method.

## Pointers as an iterator

The simplest kind of iterator is a pointer, which (using pointer arithmetic) works for data store sequentially in memory. Let's revisit a simple array traversal using a pointer and pointer arithmetic:

```
#include <array>
#include <iostream>

int main()
{
  std::array data{ 0, 1, 2, 3, 4, 5, 6 };

  auto begin{ &data[0] };
  // note that this points to one spot beyond the last element
  auto end{ begin + std::size(data) };

  // for-loop with pointer
  for (auto ptr{ begin }; ptr != end; ++ptr) // ++ to move to next element
  {
    std::cout << *ptr << ' '; // dereference to get value of current element
  }
  std::cout << '\n';

  return 0;
}

```

Output:

```
0 1 2 3 4 5 6
```

In the above, we defined two variables: `begin` (which points to the beginning of our container), and end (which marks an end point). For arrays, the end marker is typically the place in memory where the last element would be if the container contained one more element.

The pointer then iterates between `begin` and end, and the current element can be accessed by dereferencing the pointer.

**Warning**

You might be tempted to calculate the end marker using the address-of operator and array syntax like so:

```
int* end{ &array[std::size(array)] };
```

But this causes undefined behavior, because `array[std::size(array)]` accesses an element that is off the end of the array.

Instead, use:

```
int* end{ &array[0] + std::size(array) };
```

## Standard library iterators

Iterating is such a common operation that all standard library containers offer direct support for iteration. Instead of calculating our own begin and end points, we can simply ask the container for the begin and end points via functions conveniently named `begin()` and `end()`:

```
#include <array>
#include <iostream>

int main()
{
  std::array array{ 1, 2, 3 };

  // Ask our array for the begin and end points (via the begin and end member functions).
  auto begin{ array.begin() };
  auto end{ array.end() };

  for (auto p{ begin }; p != end; ++p) // ++ to move to next element.
  {
    std::cout << *p << ' '; // dereference to get value of current element.
  }
  std::cout << '\n';

  return 0;
}
```

This prints:

```
1 2 3
```

The `iterator` header also contains two generic functions (`std::begin` and `std::end`) that can be used:

```
#include <array>
#include <iostream>
#include <iterator> // For std::begin and std::end

int main()
{
  std::array array{ 1, 2, 3 };

  // Use std::begin and std::end to get the begin and end points.
  auto begin{ std::begin(array) };
  auto end{ std::end(array) };

  for (auto p{ begin }; p != end; ++p) // ++ to move to next element
  {
    std::cout << *p << ' '; // dereference to get value of current element
  }
  std::cout << '\n';

  return 0;
}
```

This also prints:

```
1 2 3
```

Don't worry about the types of the iterators for now, we'll re-visit iterators in a later chapter. The important thing is that the iterator takes care of the details of iterating through the container. All we need are four things: the begin point, the end point, operator++ to move the iterator to the next element (or the end), and operator* to get the value of the current element.

---

## Back to range-based for loops

All types that have `begin` and `end` member functions or can be used with `std::begin` and `std::end` are usable in range-based for-loops.

```
#include <array>
#include <iostream>

int main()
{
  std::array array{ 1, 2, 3 };

  // This does exactly the same as the loop we used before.
  for (int i : array)
  {
    std::cout << i << ' ';
  }
  std::cout << '\n';

  return 0;
}
```

Behind the scenes, the range-based for-loop calls `begin()` and `end()` of the type to iterate over. `std::array` has `begin` and `end` member functions, so we can use it in a range-based loop. C-style fixed arrays can be used with `std::begin` and `std::end` functions, so we can loop through them with a range-based loop as well. Dynamic arrays don't work though, because there is no `std::end` function for them (because the type information doesn't contain the array's length).

You'll learn how to add functions to your types later, so that they can be used with range-based for-loops too.

Range-based for-loops aren't the only thing that makes use of iterators. They're also used in `std::sort` and other algorithms. Now that you know what they are, you'll notice they're used quite a bit in the standard library.

## Iterator invalidation (dangling iterators)

Much like pointers and references, iterators can be left "dangling" if the elements being iterated over change address or are destroyed. When this happens, we say the iterator has been **invalidated**. Accessing an invalidated iterator produces undefined behavior.

Some operations that modify containers (such as adding an element to a `std::vector`) can have the side of effect of causing the elements in the container to change addresses. When this happens, existing iterators to those elements will be invalidated. Good C++ reference documentation should note which container operations may or will invalidate iterators. As an example, see the **"Iterator invalidation" section of `std::vector` on cppreference**.

Here's an example of this:

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

int main()
{
    std::vector v { 1, 2, 3, 4, 5, 6, 7 };

    auto it { v.begin() };

    ++it; // move to second element
    std::cout << *it << '\n'; // ok: prints 2

    v.erase(it); // erase the element currently being iterated over

    // erase() invalidates iterators to the erased element (and subsequent elements)
    // so iterator "it" is now invalidated

    ++it; // undefined behavior
    std::cout << *it << '\n'; // undefined behavior

    return 0;
}
```

**6.18 -- Introduction to standard library algorithms**

**Index**

**6.16 -- An introduction to std::vector**

(h/t to nascardriver for significant contributions to this lesson)

C++ TUTORIAL | PRINT THIS POST

## 23 comments to 6.17 — Introduction to iterators

Omran
February 4, 2020 at 6:01 am · Reply

hello!! , i know this questions out of this tutorial's context but Nascardriver are you a real nascar driver , sorry if this question was bothering anyone !! , i'm just curious , :)

nascardriver
February 4, 2020 at 8:42 am · Reply

Nope, I drive a normal car, also not in circles :)

Omran
February 4, 2020 at 8:57 am · Reply

hahahahahahahahahahahahaha cuz i was just reading something about centripetal acceleration so they covered some examples about nascar race or something :D , love you bro , you're helping a lot of c++ beginners , you and Alex :)

kavin
January 28, 2020 at 7:01 am · Reply

Hi, why am i not able to do this?

```
1 auto begin{array.begin()};
2 std::cout << begin;       //wont work. why?
3 std::cout<<array.begin(); //also won't work ?
```

what does array.begin() actually give? If it's address cout'ing begin should print address of 1st element right ?

nascardriver
January 28, 2020 at 7:06 am · Reply

It's an iterator, you can't print it. You can access the element the iterator is referring to via `*begin` or `*array.begin()`.
We haven't covered user-defined types or operator overloading yet, you'll understand how this works in chapter 9.

kavin
January 28, 2020 at 7:41 am · Reply

Oh okie, array.end() and std::end(array) also refers to value after the last element right?

nascardriver
January 28, 2020 at 7:44 am · Reply

One element past the end (ie. not a valid element). The last element is accessible via `array.back()` (Not an iterator).

hellmet
January 18, 2020 at 6:35 am · Reply

I think it's important to note that since we cannot use 'const auto&' in the range-based-for loop [not the 'for (int i : array)'; since we need to do ++iter], any modification to the underlying datastructure invalidates the iterator (I hope I got that right, from what I read online).

nascardriver
January 18, 2020 at 7:43 am · Reply

I don't understand what you mean, can you provide an example of a range-based for-loop in which we can't use `const auto&`?

Whether or not a modification to the data invalidates iterators depends on the container. In an `std::array`, you can modify all you want without invalidating iterators. `std::vector` on the other hand might perform reallocation, which invalidates iterators.

hellmet
January 18, 2020 at 8:01 am · Reply

I meant to say, one can't do this.
Also, mistake on my part! This is not a 'range-based for-loop'.

```
1 for (const auto& i = some_obj.cbegin(), i != some_obj.cend(), ++i) {...}
  // we can't do ++i as i is const
2
```

Oh right, the invalidation depends on the container! I meant perhaps adding the point here is invaluable information!

nascardriver
January 18, 2020 at 8:18 am · Reply

I added a note about the `const` iterator variable. Container invalidation doesn't fit this lesson, I'll try to remember it and add it to a lesson about algorithms or containers if it's not there already.

hellmet
January 18, 2020 at 8:42 am · Reply

I'm glad I could be of help!
Thanks!

Alex
January 22, 2020 at 11:07 pm · Reply

I do think iterator invalidation is a valid topic to be mentioned here. I added a section to the bottom of the lesson, including an example of invalidation that

uses only topics that have been covered so far. Let me know if you think it doesn't fit in for some reason.

Eddie Ball
January 7, 2020 at 6:57 am · Reply

Loving the new content! But I'm a little confused about what makes an iterator. In the section "Iterators" you say:

"An iterator is an object designed to iterate through a container without having to care about the internal structure of the data."

and then give an example of an iterator that uses pointers. How is is this iterator indifferent to the internal structure of the data? Is it because it uses pointers and the auto data type?

Thanks!

Alex
January 8, 2020 at 6:08 pm · Reply

That sentence was poorly written. To clarify: iterators themselves are tightly coupled to the data they're iterating over (the iterator has to care about the internal data structure to know how to move through it). However, once an iterator is created, the user of the iterator only has to use the iterator's interface to navigate and access the elements, and is thus abstracted from how the data is actually stored.

I've amended the lesson wording a bit to try and clarify.

siva
December 24, 2019 at 12:07 am · Reply

Hi,

Can you please explain the difference between std::size and array::size ?

Regards,
Siva.

nascardriver
December 24, 2019 at 3:03 am · Reply

```
1 std::size(arr);
```

If `arr` has a `.size()` function, `std::size(arr)` calls `arr.size()`.
`std::size` works with C-style arrays.

Jim
December 22, 2019 at 7:32 pm · Reply

std::size(data) works but why not data.size()?

nascardriver
December 23, 2019 at 7:28 am · Reply

`std::size` works for C-style arrays too. There are none in this lesson, I think this is a leftover from when the lesson used C-style arrays. `std::size` is fine too, so I'll leave it there.

Louis Cloete
December 21, 2019 at 12:18 pm · Reply

Great explanation! Just one suggestion: won't it make more sense / be clearer to initialize end like this:

```
1 auto begin{ &data[0] };
2 auto end{ begin + std::size(data) };
```

?

nascardriver
December 22, 2019 at 2:32 am · Reply

You're right! Lesson updated.

swap145
December 21, 2019 at 11:33 am · Reply

Thank you for this new section.

Alex
January 1, 2020 at 7:06 pm · Reply

Thanks to Nascardriver! He was the impetus for making this one happen.