18.2 — Introduction to iterators

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 <cstddef>
#include <iostream>

int main()
{
    // In C++17, the type of variable data is deduced to std::array<int, 7>
    // If you get an error compiling this example, see the warning below
    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';

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

    return 0;
}
```

Warning

The examples in this lesson use a C++17 feature called `class template argument deduction` to deduce the template arguments for a template variable from its initializer. In the example above, when the compiler sees `std::array data{ 0, 1, 2, 3, 4, 5, 6 };`, it will deduce that we want `std::array<int, 7> data { 0, 1, 2, 3, 4, 5, 6 };`.

If your compiler is not C++17 enabled, you'll get an error that says something like, "missing template arguments before 'data'". In that case, your best bet is to enable C++17, as per lesson 0.12 -- Configuring your compiler: Choosing a language standard. If you can not, you can replace the lines that use class template argument deduction with lines that have explicit template arguments (e.g. replace `std::array data{ 0, 1, 2, 3, 4, 5, 6 };` with `std::array<int, 7> data { 0, 1, 2, 3, 4, 5, 6 };`

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 stored 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 << ' '; // Indirection 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{ &data[std::size(data)] };
```

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

Instead, use:

```
int* end{ data.data() + std::size(data) }; // data() returns a pointer to the first
element
```

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 member 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 << ' '; // Indirection 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.

Tip

`std::begin` and `std::end` for C-style arrays are defined in the <iterator> header.

`std::begin` and `std::end` for containers that support iterators are defined in the header files for those containers (e.g. <array>, <vector>).

```
#include <array>    // includes <iterator>
#include <iostream>

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 << ' '; // Indirection 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.

`operator<` vs `operator!=` for iterators

In lesson 8.10 -- For statements, we noted that using `operator<` was preferred over `operator!=` when doing numeric comparisons in the loop condition:

```
for (index = 0; index < length; ++index)
```

With iterators, it is conventional to use `operator!=` to test whether the iterator has reached the end element:

```
for (auto p{ begin }; p != end; ++p)
```

This is because some iterator types are not relationally comparable. `operator!=` works with all iterator types.

Back to range-based for loops

All types that have both `begin()` and `end()` member functions, or that 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 C-style arrays (or decayed C-style 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 these 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 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.

Since range-based for-loops use iterators behind the scenes, we must be careful not to do anything that invalidates the iterators of the container we are actively traversing:

```
#include <vector>

int main()
{
    std::vector v { 0, 1, 2, 3 };

    for (auto num : v) // implicitly iterates over v
    {
        if (num % 2 == 0)
            v.push_back(num + 1); // when this invalidates the iterators of v,
undefined behavior will result
    }

    return 0;
}
```

Here's another example of iterator invalidation:

```
#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;
}
```

Invalidated iterators can be revalidated by assigning a valid iterator to them (e.g. `begin()`, `end()`, or the return value of some other function that returns an iterator).

The `erase()` function returns an iterator to the element one past the erased element (or `end()` if the last element was removed). Therefore, we can fix the above code like 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';

        it = v.erase(it); // erase the element currently being iterated over, set
`it` to next element

        std::cout << *it << '\n'; // now ok, prints 3

        return 0;
}
```

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