16.8 — Range-based for loops (for-each)

In lesson 16.6 -- Arrays and loops, we showed examples where we used a for-loop to iterate through each element of an array using a loop variable as an index. Here's another example of such:

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

int main()
{
    std::vector fibonacci { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

    std::size_t length { fibonacci.size() };
    for (std::size_t index { 0 }; index < length; ++index)
       std::cout << fibonacci[index] << ' ';

    std::cout << '\n';

    return 0;
}
```

Although for-loops provide a convenient and flexible way to iterate through an array, they are also easy to mess up, prone to off-by-one errors, and subject to array indexing sign problems (discussed in lesson 16.7 -- Arrays, loops, and sign challenge solutions).

Because traversing (forwards) through an array is such a common thing to do, C++ supports another type of for-loop called a **range-based for loop** (also sometimes called a **for-each loop**) that allows traversal of a container without having to do explicit indexing. Range-based for loops are simpler, safer, and work with all the common array types in C++ (including `std::vector`, `std::array`, and C-style arrays).

Range-based for loops

The *range-based for* statement has a syntax that looks like this:

```
for (element_declaration : array_object)
   statement;
```

When a range-based for loop is encountered, the loop will iterate through each element in `array_object`. For each iteration, the value of the current array element will be assigned to the variable declared in `element_declaration`, and then `statement` will execute.

For best results, `element_declaration` should have the same type as the array elements, otherwise type conversion will occur.

Here's a simple example that uses a *range-based for* loop to print all of the elements in an array named `fibonacci`:

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

int main()
{
    std::vector fibonacci { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

    for (int num : fibonacci) // iterate over array fibonacci and copy each value
into `num`
       std::cout << num << ' '; // print the current value of `num`

    std::cout << '\n';

    return 0;
}
```

This prints:

```
0 1 1 2 3 5 8 13 21 34 55 89
```

Note that this example does not require us to use the array's length, nor does it require us to index the array!

Let's take a closer look at how this works. This range-based for loop will execute through all the elements of `fibonacci`. For the first iteration, variable `num` is assigned the value of the first element (`0`). Then the program executes the associated statement, which prints the value of `num` (`0`) to the console. For the second iteration, `num` is assigned the value of the second element (`1`). The associated statement executes again, which prints `1`. The range-based for loop continues to iterate through each of the array elements in turn, executing the associated statement for each one, until there are no elements left in the array to iterate over. At that point, the loop terminates, and the program continues execution (printing a newline and then returning `0` to the operating system).

Key insight

The declared element (`num` in the prior example) is not an array index. Rather, it is assigned the value of the array element being iterated over.

Because `num` is assigned the value of the array element, this does mean the array element is copied (which can be expensive for some types).

Best practice

Favor range-based for loops over regular for-loops when traversing containers.

Range-based for loops and type deduction using the `auto` keyword

Because `element_declaration` should have the same type as the array elements (to prevent type conversion from occurring), this is an ideal case in which to use the `auto` keyword, and let the compiler deduce the type of the array elements for us. That way we don't have to redundantly specify the type, and there's no risk of accidentally mistyping (and "mis-typing", ha!)

Here's the same example as above, but using `auto` as the type of `num`:

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

int main()
{
    std::vector fibonacci { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

    for (auto num : fibonacci) // compiler will deduce type of num to be `int`
       std::cout << num << ' ';

    std::cout << '\n';

    return 0;
}
```

Because `std::vector fibonacci` has elements of type `int`, `num` will be deduced to be an `int`.

Best practice

Use type deduction (`auto`) with range-based for loops to have the compiler deduce the type of the array element.

Another benefit to using `auto` is that if the element type of the array is ever updated (e.g. from `int` to `long`), `auto` will automatically deduce the updated element type, ensuring they stay in sync (and preventing type conversion from occurring).

Avoid element copies using references

Consider the following range-based for loop, which iterates over an array of `std::string`:

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

int main()
{
    using namespace std::literals; // for s suffix for std::string literals
    std::vector words{ "peter"s, "likes"s, "frozen"s, "yogurt"s }; //
std::vector<std::string>

    for (auto word : words)
        std::cout << word << ' ';

    std::cout << '\n';

    return 0;
}
```

For each iteration of this loop, the next `std::string` element from the `words` array will be assigned (copied) into the variable `word`. Copying a `std::string` is expensive, which is why we typically pass `std::string` to functions by const reference. We want to avoid making copies of things that are expensive to copy unless we really need a copy. In this case, we're just printing the value of the copy and then the copy is destroyed. It would be better if we could avoid making a copy and just reference the actual array element.

Fortunately, we can do exactly that by making our `element_declaration` a (const) reference:

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

int main()
{
    using namespace std::literals; // for s suffix for std::string literals
    std::vector words{ "peter"s, "likes"s, "frozen"s, "yogurt"s }; //
std::vector<std::string>

    for (const auto& word : words) // word is now a const reference
        std::cout << word << ' ';

    std::cout << '\n';

    return 0;
}
```

In the above example, `word` is now a const reference. With each iteration of this loop, `word` will be bound to the next array element. This allows us to access the array element's value without having to make an expensive copy.

If the reference is non-const, it can also be used to change the values in the array (something not possible if our `element_declaration` is a copy of the value).

Best practice

In range-based for loops, the element declaration should use a (const) reference whenever you would normally pass that element type by (const) reference.

Consider always using `const auto&` when you don't want to work with copies of elements

Normally we'd use `auto` for cheap-to-copy types, and `const auto&` for expensive-to-copy types. But with range-based for loops, many developers believe it is preferable to always use `const auto&` because it is more future-proof.

For example, consider the following example:

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

int main()
{
    using namespace std::literals;
    std::vector words{ "peter"sv, "likes"sv, "frozen"sv, "yogurt"sv }; //
std::vector<std::string_view>

    for (auto word : words) // We normally pass string_view by value, so we'll use
auto here
        std::cout << word << ' ';

    std::cout << '\n';

    return 0;
}
```

In this example, we have a `std::vector` containing `std::string_view` objects. Since `std::string_view` is normally passed by value, using `auto` seems appropriate.

But consider what happens if `words` is later updated to an array of `std::string` instead.

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

int main()
{
    using namespace std::literals;
    std::vector words{ "peter"s, "likes"s, "frozen"s, "yogurt"s }; // obvious we
should update this

    for (auto word : words) // Probably not obvious we should update this too
        std::cout << word << ' ';

    std::cout << '\n';

    return 0;
}
```

The range-based for loop will compile and execute just fine, but `word` will now be deduced to be a `std::string`, and because we're using `auto`, our loop will silently make expensive copies of `std::string` elements. We just took a huge performance hit!

There are a couple of reasonable ways to ensure this doesn't happen:

- Don't use type deduction in your range-based for loop. If we'd explicitly specified the element type as `std::string_view`, then when the array was later updated to `std::string`, the `std::string` elements will implicitly convert to `std::string_view`, which is no problem. If the array is updated to some other type that is not convertible, the compiler will error, and we can figure out what the appropriate thing to do is at that point.
- Always use `const auto&` instead of `auto` when using type deduction in your range-based for loop when you don't want to work with copies. The performance penalty of accessing elements through a reference instead of by value is likely to be small, and this insulates us from potentially significant performance penalties down the road if the element type is later changed to something that is expensive to copy.

Tip

If using type deduction in a range-based for loop, consider always using `const auto&` unless you need to work with copies. This will ensure copies aren't made even if the element type is later changed.

Range-based for loops and other standard container types

Range-based for loops work with a wide variety of array types, including (non-decayed) C-style arrays, `std::array`, `std::vector`, linked list, trees, and maps. We haven't covered any of these yet, so don't worry if you don't know what these are. Just remember that *range-*

based for loops provide a flexible and generic way to iterate through more than just `std::vector`:

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

int main()
{
    std::array fibonacci{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 }; // note use of
std::array here

    for (auto number : fibonacci)
    {
        std::cout << number << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

For advanced readers

Range-based for loops won't work with decayed C-style arrays. This is because a range-based for-loop needs to know the length of the array to know when traversal is complete, and decayed C-style arrays do not contain this information.

Range-based for loops also won't work with enumerations. We show a method for working around this in lesson 17.6 -- std::array and enumerations.

Getting the index of the current element

Range-based for loops do *not* provide a direct way to get the array index of the current element. This is because many of the structures that a range-based for loop can iterate over (such as `std::list`) do not support indices.

However, because range-based for loops always iterate in a forwards direction and don't skip elements, you can always declare (and increment) your own counter. However, if you're going to do this, you should consider whether you're better off using a normal for-loop instead of a range-based for loop.

Range-based for loops in reverse C++20

Range-based for loops only iterate in forwards order. However, there are cases where we want to traverse an array in reverse order. Prior to C++20, range-based for loops could not be easily used for this purpose, and other solutions had to be employed (typically normal for-loops).

However, as of C++20, you can use the `std::views::reverse` capability of the Ranges library to create a reverse view of the elements that can be traversed:

```
#include <iostream>
#include <ranges> // C++20
#include <string_view>
#include <vector>

int main()
{
    using namespace std::literals; // for sv suffix for std::string_view literals
    std::vector words{ "Alex"sv, "Bobby"sv, "Chad"sv, "Dave"sv }; // sorted in
alphabetical order

    for (const auto& word : std::views::reverse(words)) // create a reverse view
        std::cout << word << ' ';

    std::cout << '\n';

    return 0;
}
```

This prints:

```
Dave
Chad
Bobby
Alex
```

We haven't covered the ranges library, so consider this a useful bit of magic for now.

Quiz time

Question #1

Define a `std::vector` with the following names: "Alex", "Betty", "Caroline", "Dave", "Emily", "Fred", "Greg", and "Holly". Ask the user to enter a name. Use a range-based for loop to see if the name the user entered is in the array.

Sample output:

```
Enter a name: Betty
Betty was found.

Enter a name: Megatron
Megatron was not found.
```

Hint: Use `std::string` to hold the string the user inputs.
Hint: `std::string_view` is cheap to copy.

Show Solution

Question #2

Modify your solution to quiz 1. In this version, create a function template (not a normal function) called `isValueInArray()` that takes two parameters: a `std::vector`, and a value. The function should return `true` if the value is in the array, and `false` otherwise. Call the function from `main()` and pass it the array of names and the name the user entered.

Reminder:

- A function template that is using template argument deduction (when template type arguments are not explicitly specified) will not do conversions to match the template type parameters. The call either matches the template (and the template type can be deduced) or it doesn't.
- A function template with an explicitly specified template type argument will convert arguments to match the parameter type (since the type is known).

Show Solution