10.19 — For-each loops

ALEX JULY 22, 2021

In lesson 10.3 -- Arrays and loops, we showed examples where we used a *for loop* to iterate through each element of an array.

For example:

```
#include <iostream>
#include <iterator> // std::size

int main()
{
    constexpr int scores[]{ 84, 92, 76, 81, 56 };
    constexpr int numStudents{ std::size(scores) };

    int maxScore{ 0 }; // keep track of our largest score
    for (int student{ 0 }; student < numStudents;
++student)
    {
        if (scores[student] > maxScore)
        {
            maxScore = scores[student];
        }
    }

    std::cout << "The best score was " << maxScore <<
'\n';

    return 0;
}
```

While *for loops* provide a convenient and flexible way to iterate through an array, they are also easy to mess up and prone to off-by-one errors.

There's a simpler and safer type of loop called afor-each loop (also called arange-based for-loop) for cases where we want to iterate through every element in an array (or other list-type structure).

For-each loops

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

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

When this statement is encountered, the loop will iterate through each element in array, assigning the value of the current array element to the variable declared in element_declaration. For best results, element_declaration should have the same type as the array elements, otherwise type conversion will occur.

Let's take a look at a simple example that uses a *for-each* loop to print all of the elements in an array named fibonacci:

```
#include <iostream>

int main()
{
    constexpr int fibonacci[]{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
    for (int number : fibonacci) // iterate over array fibonacci
    {
       std::cout << number << ' '; // we access the array element for this iteration through variable number
    }

    std::cout << '\n';

    return 0;
}
```

This prints:

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

Let's take a closer look at how this works. First, the *for loop* executes, and variable number is set to the value of the first element, which has value 0. The program executes the statement, which prints 0. Then the *for loop* executes again, and number is set to the value of the second element, which has value 1. The statement executes again, which prints 1. The *for loop* continues to iterate through each of the numbers in turn, executing the 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 (returning 0 to the operating system).

Note that variable number is not an array index. It's assigned the value of the array element for the current loop iteration.

For each loops and the auto keyword

Because element_declaration should have the same type as the array elements, this is an ideal case in which to use the `auto` keyword, and let C++ deduce the type of the array elements for us.

Here's the above example, using auto:

```
#include <iostream>

int main()
{
    constexpr int fibonacci[]{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
    for (auto number : fibonacci) // type is auto, so number has its type deduced from the fibonacci array
    {
       std::cout << number << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

For-each loops and references

In the following for-each example, our element declarations are declared by value:

```
std::string array[]{ "peter", "likes", "frozen", "yogurt" };
for (auto element : array) // element will be a copy of the current array
element
{
    std::cout << element << ' ';
}
```

This means each array element iterated over will be copied into variable element. Copying array elements can be expensive, and most of the time we really just want to refer to the original element. Fortunately, we can use references for this:

```
std::string array[]{ "peter", "likes", "frozen", "yogurt" };
for (auto& element: array) // The ampersand makes element a reference to the actual array element,
preventing a copy from being made
{
    std::cout << element << ' ';
}
```

In the above example, element will be a reference to the currently iterated array element, avoiding having to make a copy. Also any changes to element will affect the array being iterated over, something not possible if element is a normal variable.

And, of course, it's a good idea to make your reference `const` if you're intending to use it in a read-only fashion:

```
std::string array[]{ "peter", "likes", "frozen", "yogurt" };
for (const auto& element: array) // element is a const reference to the currently iterated array
element
{
    std::cout << element << ' ';
}
```

Best practice

In for-each loops element declarations, if your elements are non-fundamental types, use references or `const` references for performance reasons.

Rewriting the max scores example using a for-each loop

Here's the example at the top of the lesson rewritten using a *for each* loop:

```cpp
#include <iostream>

int main()
{
    constexpr int scores[]{ 84, 92, 76, 81, 56 };
    int maxScore{ 0 }; // keep track of our largest score

    for (auto score : scores) // iterate over array scores, assigning each value in turn to variable score
    {
        if (score > maxScore)
        {
            maxScore = score;
        }
    }

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

Note that in this example, we no longer have to manually subscript the array or get its size. We can access the array element directly through variable score. The array has to have size information. An array that decayed to a pointer cannot be used in a for-each loop.

For-each loops and non-arrays

For-each **loops don't only work with fixed arrays, they work with many kinds of list-like structures, such as vectors (e.g.** `std::vector`**), linked lists, 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 for each loops provide a flexible and generic way to iterate through more than just arrays.**

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

int main()
{
    std::vector fibonacci{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 }; // note use of std::vector here rather than a fixed array
    // Before C++17
    // std::vector<int> fibonacci{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

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

    std::cout << '\n';

    return 0;
}
```

For-each doesn't work with pointers to an array

In order to iterate through the array, for-each needs to know how big the array is, which means knowing the array size. Because arrays that have decayed into a pointer do not know their size, for-each loops will not work with them!

```cpp
#include <iostream>

int sumArray(const int array[]) // array is a pointer
{
    int sum{ 0 };

    for (auto number : array) // compile error, the size of array isn't known
    {
        sum += number;
    }

    return sum;
}

int main()
{
     constexpr int array[]{ 9, 7, 5, 3, 1 };

     std::cout << sumArray(array) << '\n'; // array decays into a pointer here

     return 0;
}
```

Similarly, dynamic arrays won't work with for-each loops for the same reason.

Can I get the index of the current element?

***For-each* loops do *not* provide a direct way to get the array index of the current element. This is because many of the structures that *for-each* loops can be used with (such as linked lists) are not directly indexable!**

Since C++20, range-based for-loops can be used with aninit-statement just like the init-statement in if-statements. We can use the init-statement to create a manual index counter without polluting the function in which the for-loop is placed.

The init-statement is placed right before the loop variable:

```
for (init-statement; element_declaration : array)
   statement;
```

In the following code, we have two arrays which are correlated by index. For example, the student with the name at `names[3]` has a score of `scores[3]`. Whenever a student with a new high score is found, we print their name and difference in points to the previous high score.

```cpp
#include <iostream>

int main()
{
    std::string names[]{ "Alex", "Betty", "Caroline", "Dave", "Emily" }; // Names of the students
    constexpr int scores[]{ 84, 92, 76, 81, 56 };
    int maxScore{ 0 };

    for (int i{ 0 }; auto score : scores) // i is the index of the current element
    {
        if (score > maxScore)
        {
            std::cout << names[i] << " beat the previous best score of " << maxScore << " by " << (score - maxScore) << " points!\n";
            maxScore = score;
        }

        ++i;
    }

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

Output

```
Alex beat the previous best score of 0 by 84 points!
Betty beat the previous best score of 84 by 8 points!
The best score was 92
```

The `int i{ 0 };` is the init-statement, it only gets executed once when the loop starts. At the end of each iteration, we increment `i`, similar to a normal for-loop. However, if we were to use `continue` inside the loop, the `++i` would get skipped, leading to unexpected results. If you use `continue`, you need to make sure that `i` gets incremented before the `continue` is encountered.

Before C++20, the index variable `i` had to be declared outside of the loop, which could lead to name conflicts when we wanted to define another variable named `i` later in the function.

Conclusion

***For-each* loops provide a superior syntax for iterating through an array when we need to access all of the array elements in forwards sequential order. It should be preferred over the standard for loop in the cases where it can be used. To prevent making copies of each element, the element declaration should ideally be a reference.**

Quiz time

This one should be easy.

Question #1

Declare a fixed array with the following names: Alex, Betty, Caroline, Dave, Emily, Fred, Greg, and Holly. Ask the user to enter a name. Use a *for each* 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_view` as your array type.

Show Solution

Next lesson
10.20 Void pointers

Back to table of contents

Previous lesson
10.18 Member selection with pointers and references

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ co

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

©2021 Learn C++