# 6.12a — For-each loops

BY ALEX ON JULY 31ST, 2015 | LAST MODIFIED BY ALEX ON FEBRUARY 15TH, 2018

In lesson **6.3 -- Arrays and loops**, we showed examples where we used a *for loop* to iterate through each element of an array.

For example:

```
#include <iostream>

int main()
{
    const int numStudents = 5;
    int scores[numStudents] = { 84, 92, 76, 81, 56 };
    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.

C++11 introduces a new type of loop called a **for-each** loop (also called a **range-based for** loop) that provides a simpler and safer method 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()
{
    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 n
umber

    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()
{
    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 ar
ray
        std::cout << number << ' ';

    return 0;
}
```

**For-each loops and references**

In the for-each examples above, our element declarations are declared by value:

```
    int array[5] = { 9, 7, 5, 3, 1 };
    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:

```
    int array[5] = { 9, 7, 5, 3, 1 };
    for (auto &element: array) // The ampersand makes element a reference to the actual array element, p
reventing 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 element const if you're intending to use it in a read-only fashion:

```
    int array[5] = { 9, 7, 5, 3, 1 };
    for (const auto &element: array) // element is a const reference to the currently iterated array ele
ment
        std::cout << element << ' ';
```

*Rule: Use references or const references for your element declaration in for-each loops 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:

```
#include <iostream>

int main()
{
    const int numStudents = 5;
    int scores[numStudents] = { 84, 92, 76, 81, 56 };
    int maxScore = 0; // keep track of our largest score
    for (const auto &score: scores) // iterate over array scores, assigning each value in turn to varia
ble 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. We can access the array element directly through variable score.

**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.

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

int main()
{
    std::vector<int> fibonacci = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 }; // note use of std::vecto
r here rather than a fixed array
    for (const auto &number : fibonacci)
        std::cout << number << ' ';

    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!

```
#include <iostream>

int sumArray(int array[]) // array is a pointer
{
    int sum = 0;
    for (const auto &number : array) // compile error, the size of array isn't known
        sum += number;

    return sum;
}

int main()
{
     int array[5] = { 9, 7, 5, 3, 1 };
     std::cout << sumArray(array); // 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!

**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.

Note that because *for each* was added in C++11, it won't work with compilers that don't have support for C++11.

**Quiz**

This one should be easy.

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 as your array type.

1) **Hide Solution**

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

int main()
{
    const std::string names[] = { "Alex", "Betty", "Caroline", "Dave", "Emily", "Fred", "Greg", "Holly"
 };

    std::cout << "Enter a name: ";
    std::string username;
    std::cin >> username;

    bool found(false);
    for (const auto &name : names)
        if (name == username)
        {
            found = true;
            break;
        }

    if (found)
        std::cout << username << " was found.\n";
    else
        std::cout << username << " was not found.\n";

    return 0;
}
```



**6.13 -- Void pointers**


**Index**


**6.12 -- Member selection with pointers and references**