6.12a — For-each loops

BY ALEX ON JULY 31ST, 2015 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 24TH, 2020

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

int main()
{
    constexpr int scores[]{ 84, 92, 76, 81, 56 };
    constexpr int numStudents{ static_cast<int>(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.

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()
{
    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 v

```

```
    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 fib
       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 e
        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:

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

Rule

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:

```
#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 va
        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.

```
#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::vect

    for (const 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!

```
#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[]{ 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!

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

6.13 -- Void pointers

Index

6.12 -- Member selection with pointers and references

C++ TUTORIAL | PRINT THIS POST

261 comments to 6.12a — For-each loops

« Older Comments 1 2 3 4

Owen Ashurst
January 23, 2020 at 10:08 am · Reply

Just thought I'd share my implementation of the quiz. I used a vector instead.

```
1 #include <iostream>
2 #include <vector>
```

```
3  #include <string>
4
5  std::vector<std::string> names{ "Alex", "Betty", "Caroline", "Dave", "Emily", "Fred", "Greg
6
7  std::string askForName()
8  {
9      std::string enteredName;
10
11     std::cout << "Enter a name: ";
12
13     std::cin >> enteredName;
14
15     return enteredName;
16 }
17
18 void checkIfNameIsFoundInListAndNotifyUser(std::string askedForName)
19 {
20     for (const auto &name : names)
21     {
22         if (name == askedForName)
23         {
24             std::cout << askedForName << " was found.";
25
26             return;
27         }
28     }
29
30     std::cout << askedForName << " was not found.";
31 }
32
33 int main()
34 {
35     std::string askedForName = askForName();
36
37     checkIfNameIsFoundInListAndNotifyUser(askedForName);
38
39     return 0;
40 }
```

nascardriver
January 24, 2020 at 1:06 am · Reply

Congratulations on solving this quiz!
Global variables are evil, and unnecessary in this example. `names` can be moved into `checkIfNameIsFoundInListAndNotifyUser`.

Vova
January 14, 2020 at 7:09 pm · Reply

Hi,
I just noticed a rule suggested in the "For each loops and references" section:
Rule: In for-each loops element declarations, if your elements are non-fundamental types, use references or const references for performance reasons.

But in the examples above didn't we use int? Isn't int a fundamental data type?

nascardriver

January 15, 2020 at 4:02 am · Reply

You're right! `int` is a fundamental type and should be copied rather than referenced. I updated the example to use `std::string`. Thanks!

Siva
December 17, 2019 at 2:42 am · Reply

Hi,

Can you please clarify below point.

How does for each loop know size of linked list in advance?

Regards,
Siva

nascardriver
December 17, 2019 at 2:51 am · Reply

Hi Siva!

The range-based loop doesn't know the length. It keeps looping until it finds the end of the list.

```
for (auto* element{ list.first }; element != list.end; element = element.next)
{
  // ...
}
```

Where `list.first` is the first element, `list.end` is the end marker, and `element.next` is the next element in the list.

A lesson about this is in the making. Look up "iterators" for more information.

Luiz Carlos
November 14, 2019 at 8:57 am · Reply

I have a question regarding the 'nameInput' variable I defined above the infinite loop. Where is the best place for it to be defined? Inside or outside the loop?

Also, is this an efficient way of accomplishing what I did?

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

int main()
{
    std::cout << "Check if a name is VIP or not" << '\n';
    std::cout << "Enter 'q' to exit" << '\n';

    const std::string vipList[]{ "Alex", "Betty", "Caroline", "Dave", "Emily", "Fred", "Gre

    // What is the best position to initialize or just define nameInput?

    // Position 1 (Current)
    std::string nameInput{};

    while (true)
    {
        // Position 2
```

```
19
20          std::cout << "Name: "; // Input
21          std::cin >> nameInput;
22
23          if (nameInput == "q") // Exit protocol
24          {
25              std::cout << "Terminating" << '\n';
26              break;
27          }
28
29          bool isVIP{ false };
30
31          for (const auto &name : vipList) // VIP check
32          {
33              if (name == nameInput)
34              {
35                  isVIP = true;
36                  break;
37              }
38          }
39
40          if (isVIP) // Output
41              std::cout << nameInput << " is a VIP!" << '\n';
42          else
43              std::cout << nameInput << " not a VIP." << '\n';
44      }
45
46      return 0;
47  }
```

nascardriver
November 15, 2019 at 2:27 am · Reply

Outside the loop is probably faster, because the string can re-use memory.
Generally, inside the loop is better to limit the scope.

Efficiency really isn't a concern when you're blocking to ask for user input, as that's what takes up the majority of time.
A minor performance improvement is to not use `"text" << '\n'` but instead `"text\n"`. If you don't like that, you can also do `"text" "\n"`. This will concatenate the strings at compile-time but keep a space between the text and the line break.

Luiz
November 15, 2019 at 9:32 am · Reply

okay, thanks

alfonso
October 23, 2019 at 12:17 am · Reply

Pretty much the same solution.

```
1  #include <iostream>
2  #include <string>
3  #include <limits>
4
5  std::string getUserName () {
```

```
6       std::string name {};
7
8       while (true) {
9           std::cout << "Enter a name: ";
10          getline (std::cin, name);
11
12          if (std::cin.fail ()) {
13              std::cin.clear ();
14              std::cin.ignore (std::numeric_limits <std::streamsize>::max (), '\n');
15              continue;
16          } else {
17              return name;
18          }
19      }
20  }
21
22
23  int main () {
24      const int len {8};
25      const std::string names [len]
26          {"Alex", "Betty", "Caroline", "Dave", "Emily", "Fred", "Greg", "Holly"};
27
28      std::string userName {getUserName ()};
29      bool found {false};
30
31      for (const auto& name : names) {
32          if (userName == name) {
33              found = true;
34              break;
35          }
36      }
37
38      if (found) {
39          std::cout << userName << " was found.\n";
40      } else {
41          std::cout << userName << " was not found.\n";
42      }
43
44      return 0;
45  }
```

nascardriver
October 23, 2019 at 3:10 am · Reply

Hello alfonso!

Unless you have a specific reason to manually set the array's length (Line 24), you should n't do so. If you create the array first and then get its length, if at all, your code is easier to update. In this case, you can remove `len` and your program will work just fine.
Your code looks good otherwise, keep it up :)

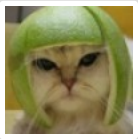
alfonso
October 23, 2019 at 11:17 pm · Reply

Thank you!

Alex

October 9, 2019 at 6:32 pm · Reply

It might be a good idea to change the title of this article; foreach loops are more commonly called "range-based for loops" in C++. "Foreach" is more of a C# term.



Alex
October 18, 2019 at 11:22 am · Reply

I'll do this when I revise this lesson. Thanks for the suggestion!

Alex
November 17, 2019 at 11:46 am · Reply

I had some confusion on this "range-based for loops" vs std::for_each loop.

I hope that distinction will be made clearer when you revise.

Still, I am very grateful for this website. Thank you.

pibaereg
October 1, 2019 at 3:35 pm · Reply

Hi!
Is it okay to simply add a return statement like this? Or should I avoid doing this? Thanks!

```
#include <iostream>


using namespace std;

int main()
{
    string x;
    cout << "Enter a name: ";
    cin >> x;
    string array[] = { "Alex", "Betty", "Caroline", "Dave",
         "Emily", "Fred", "Greg", "Holly" };
    for (const auto &b : array) {
        if (x == b) {
            cout << x << " was found." ;
            return 0;
        }
    }
    cout << x << " was not found.";

    return 0;
}
```

nascardriver
October 2, 2019 at 4:36 am · Reply

It's best to avoid `return` and `break`. If you think you need to `return` from a loop, move your code into a separate function, eg. `arrayContains`. Then call `arrayContains` from `main`. The `return` in `arrayContains` is easily spotted, not so much the `return` in `main`.

« Older Comments 1 2 3 4