16.6 — Arrays and loops

learncpp.com/cpp-tutorial/arrays-and-loops/

In the introductory lesson for this chapter (16.1 -- Introduction to containers and arrays), we introduced the scalability challenges that occur when we have many related individual variables. In this lesson, we'll reintroduce the problem, and then discuss how arrays can help us elegantly solve problems such as these.

The variable scalability challenge, revisited

Consider the case where we want to find the average test score for a class of students. To keep these examples concise, we'll assume the class has only 5 students.

Here's how we might solve this using individual variables:

```
#include <iostream>

int main()
{
    // allocate 5 integer variables (each with a different name)
    int testScore1{ 84 };
    int testScore2{ 92 };
    int testScore3{ 76 };
    int testScore4{ 81 };
    int testScore5{ 56 };

    int average { (testScore1 + testScore2 + testScore3 + testScore4 + testScore5) / 
5 };

    std::cout << "The class average is: " << average << '\n';

    return 0;
}
```

That's a lot of variables and a lot of typing. Imagine how much work we'd have to do for 30 students, or 600. Furthermore, if a new test score is added, a new variable has to be declared, initialized, and added to the average calculation. And did you remember to update the divisor? If you forgot, you now have a semantic error. Any time you have to modify existing code, you run the risk of introducing errors.

By now, you know that we should be using an array when we have a bunch of related variables. So let's replace our individual variables with a `std::vector`:

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

int main()
{
    std::vector testScore { 84, 92, 76, 81, 56 };
    std::size_t length { testScore.size() };

    int average { (testScore[0] + testScore[1] + testScore[2] + testScore[3] +
testScore[4])
        / static_cast<int>(length) };

    std::cout << "The class average is: " << average << '\n';

    return 0;
}
```

This is better -- we've cut down on the number of variables defined significantly, and the divisor for the average calculation is now determined directly from the length of the array.

But the average calculation is still a problem, as we're having to manually list each element individually. And because we have to list each element explicitly, our average calculation only works for arrays with exactly as many elements as we list. If we also want to be able to average arrays of other lengths, we'll need to write a new average calculation for each different array length.

What we really need is some way to access each array element without having to explicitly list each one.

Arrays and loops

In previous lessons, we noted that array subscripts do not need to be constant expressions -- they can be runtime expressions. This means we can use the value of a variable as an index.

Also note that the array indices used in the average calculation of the previous example are an ascending sequence: 0, 1, 2, 3, 4. Therefore, if we had some way to set a variable to values 0, 1, 2, 3, and 4 in sequence, then we could just use that variable as our array index instead of literals. And we already know how to do that -- with a for-loop.

Related content

We covered for loops in lesson 8.10 -- For statements.

Let's rewrite the above example using a for-loop, where the loop variable is used as the array index:

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

int main()
{
    std::vector testScore { 84, 92, 76, 81, 56 };
    std::size_t length { testScore.size() };

    int average { 0 };
    for (std::size_t index{ 0 }; index < length; ++index) // index from 0 to length-1
        average += testScore[index];                      // add the value of element
with index `index`
    average /= static_cast<int>(length);                  // calculate the average

    std::cout << "The class average is: " << average << '\n';

    return 0;
}
```

This should be pretty straightforward. `index` starts at `0`, `testScore[0]` is added to `average`, and `index` is incremeted to `1`. `testScore[1]` is added to `average`, and `index` is incremented to `2`. Eventually, when `index` is incremented to `5`, `index < length` is false, and the loop terminates.

At this point, the loop has added the values of `testScore[0]`, `testScore[1]`, `testScore[2]`, `testScore[3]`, and `testScore[4]` to `average`.

Finally, we calculate our average by dividing the accumulated values by the array length.

This solution is ideal in terms of maintainability. The number of times the loop iterates is determined from the length of the array, and the loop variable is used to do all of the array indexing. We no longer have to manually list each array element.

If we want to add or remove a test score, we can just modify the number of array initializers, and the rest of the code will still work without further changes!

Accessing each element of a container in some order is called **traversal**, or **traversing** the container. Traversal is often called **iteration**, or **iterating over** or **iterating through** the container.

Author's note

Since the container classes use type `size_t` for length and indices, in this lesson, we'll do the same. We'll discuss using signed lengths and indices in upcoming lesson 16.7 -- Arrays, loops, and sign challenge solutions.

Templates, arrays, and loops unlock scalability

Arrays provide a way to store multiple objects without having to name each element.

Loops provide a way to traverse an array without having to explicitly list each element.

Templates provide a way to parameterize the element type.

Together, templates, arrays, and loops allow us to write code that can operate on a container of elements, regardless of the element type or number of elements in the container!

To illustrate this further, let’s rewrite the above example, refactoring the average calculation into a function template:

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

// Function template to calculate the average of the values in a std::vector
template <typename T>
T calculateAverage(const std::vector<T>& arr)
{
    std::size_t length { arr.size() };

    T average { 0 };                                     // if our array has
elements of type T, our average should of type T too
    for (std::size_t index{ 0 }; index < length; ++index) // iterate through all the
elements
        average += arr[index];                           // sum up all the elements
    average /= static_cast<int>(length);

    return average;
}

int main()
{
    std::vector class1 { 84, 92, 76, 81, 56 };
    std::cout << "The class 1 average is: " << calculateAverage(class1) << '\n'; //
calc average of 5 ints

    std::vector class2 { 93.2, 88.6, 64.2, 81.0 };
    std::cout << "The class 2 average is: " << calculateAverage(class2) << '\n'; //
calc average of 4 doubles

    return 0;
}
```

This prints:

```
The class 1 average is: 77
The class 2 average is: 81.75
```

In the above example, we've created function template `calculateAverage()`, which takes a `std::vector` of any element type and any length, and returns the average. In `main()`, we demonstrate that this function works equally well when called with an array of 5 `int` elements or an array of 4 `double` elements!

`calculateAverage()` will work for any type T that supports the operators used inside the function (`operator+=(T)`, `operator/=(int)`). If you try to use a T that does not support these operators, the compiler will error when trying to compile the instantiated function template.

What we can do with arrays and loops

Now that we know how to traverse a container of elements using a loop, let's look at the most common things that we can use container traversal for. We typically traverse a container in order to do one of four things:

1. Calculate a new value based on the value of existing elements (e.g. average value, sum of values).
2. Search for an existing element (e.g. has exact match, count number of matches, find highest value).
3. Operate on each element (e.g. output each element, multiply all elements by 2).
4. Reorder the elements (e.g. sort the elements in ascending order).

The first three of these are fairly straightforward. We can use a single loop to traverse the array, inspecting or modifying each element as appropriate.

Reordering the elements of a container is quite a bit more tricky, as doing so typically involves using a loop inside another loop. While we can do this manually, it's usually better to use an existing algorithm from the standard library to do this. We'll cover this in more detail in a future chapter when we discuss algorithms.

Arrays and off-by-one errors

When traversing a container using an index, you must take care to ensure the loop executes the proper number of times. Off-by-one errors (where the loop body executes one too many or one too few times) are easy to make.

Typically, when traversing a container using an index, we will start the index at `0` and loop until `index < length`.

New programmers sometimes accidentally use `index <= length` as the loop condition. This will cause the loop to execute when `index == length`, which will result in an out of bounds subscript and undefined behavior.

Quiz time

Question #1

Write a short program that prints the elements of the following vector to the screen using a loop:

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

int main()
{
    std::vector arr{ 4, 6, 7, 3, 8, 2, 1, 9 };

    // Add your code here

    return 0;
}
```

The output should look like this:

```
4 6 7 3 8 2 1 9
```

Show Solution

Question #2

Update your code for the prior quiz solution so that the following program compiles and has the same output:

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

// Implement printArray() here

int main()
{
    std::vector arr{ 4, 6, 7, 3, 8, 2, 1, 9 };

    printArray(arr); // use function template to print array

    return 0;
}
```

Show Solution

Question #3

Given the solution for quiz 2, do the following:

- Ask the user for a value between 1 and 9. If the user does not enter a value between 1 and 9, repeatedly ask for an integer value until they do. If the user enters a number followed by extraneous input, ignore the extraneous input.
- Print the array.
- Write a function template to search the array for the value that the user entered. If the value is in the array, return the index of that element. If the value is not in the array, return an appropriate value.
- If the value was found, print the value and index. If the value was not found, print the value and that it was not found.

We cover how to handle invalid input in lesson 9.5 -- std::cin and handling invalid input.

Here are two sample runs of this program:

```
Enter a number between 1 and 9: d
Enter a number between 1 and 9: 6
4 6 7 3 8 2 1 9
The number 6 has index 1

Enter a number between 1 and 9: 5
4 6 7 3 8 2 1 9
The number 5 was not found
```

Show Solution

Question #4

Write a function template to find the largest value in a `std::vector`. If the vector is empty, return the default value for the element type.

The following code should execute:

```
int main()
{
    std::vector data1 { 84, 92, 76, 81, 56 };
    std::cout << findMax(data1) << '\n';

    std::vector data2 { -13.0, -26.7, -105.5, -14.8 };
    std::cout << findMax(data2) << '\n';

    std::vector<int> data3 { };
    std::cout << findMax(data3) << '\n';

    return 0;
}
```

And print the following result:

```
92
-13
0
```

Show Hint

Show Solution

Question #5

In the quiz for lesson 8.10 -- For statements, we implemented a game called FizzBuzz for the numbers three, five, and seven.

In this quiz, implement the game as follows:

- Numbers divisible by only 3 should print “fizz”.
- Numbers divisible by only 5 should print “buzz”.
- Numbers divisible by only 7 should print “pop”.
- Numbers divisible by only 11 should print “bang”.
- Numbers divisible by only 13 should print “jazz”.
- Numbers divisible by only 17 should print “pow”.
- Numbers divisible by only 19 should print “boom”.
- Numbers divisible by more than one of the above should print each of the words associated with its divisors.
- Numbers not divisible by any of the above should just print the number.

Use a `std::vector` to hold the divisors, and another `std::vector` to hold the words (as type `std::string_view`). If the arrays do not have the same length, the program should assert. Produce output for 150 numbers.

Show Hint

Show Hint

Here’s the expected output from the first 21 iterations:

```
1
2
fizz
4
buzz
fizz
pop
8
fizz
buzz
bang
fizz
jazz
pop
fizzbuzz
16
pow
fizz
boom
buzz
fizzpop
```

Show Solution