6.3 — Arrays and loops

BY ALEX ON JULY 2ND, 2007 | LAST MODIFIED BY ALEX ON NOVEMBER 5TH, 2017

Consider the case where we want to find the average test score of a class of students. Using individual variables:

```
const int numStudents = 5;
int score0 = 84;
int score1 = 92;
int score2 = 76;
int score3 = 81;
int score4 = 56;

int totalScore = score0 + score1 + score2 + score3 + score4;
double averageScore = static_cast<double>(totalScore) / numStudents;
```

That's a lot of variables and a lot of typing -- and this is just 5 students! Imagine how much work we'd have to do for 30 students, or 150.

Plus, if a new student is added, a new variable has to be declared, initialized, and added to the totalScore calculation. Any time you have to modify old code, you run the risk of introducing errors.

Using arrays offers a slightly better solution:

```
const int numStudents = 5;
int scores[numStudents] = { 84, 92, 76, 81, 56 };
int totalScore = scores[0] + scores[1] + scores[2] + scores[3] + scores[4];
double averageScore = static_cast<double>(totalScore) / numStudents;
```

This cuts down on the number of variables declared significantly, but totalScore still requires each array element be listed individually. And as above, changing the number of students means the totalScore formula needs to be manually adjusted.

If only there were a way to loop through our array and calculate totalScore directly.

Loops and arrays

In a previous lesson, you learned that the array subscript doesn't need to be a constant value -- it can be a variable. This means we can use a loop variable as an array index to loop through all of the elements of our array and perform some calculation on them. This is such a common thing to do that wherever you find arrays, you will almost certainly find loops! When a loop is used to access each array element in turn, this is often called **iterating** through the array.

Here's our example above using a *for loop*:

```
int scores[] = { 84, 92, 76, 81, 56 };
const int numStudents = sizeof(scores) / sizeof(scores[0]);
int totalScore = 0;

// use a loop to calculate totalScore
for (int student = 0; student < numStudents; ++student)
    totalScore += scores[student];

double averageScore = static_cast<double>(totalScore) / numStudents;
```

This solution is ideal in terms of both readability and maintenance. Because the loop does all of our array element accesses, the formulas adjust automatically to account for the number of elements in the array. This means the calculations do not have to be manually altered to account for new students, and we do not have to manually add the name of new array elements!

Here's an example of using a loop to search an array in order to determine the best score in the class:

```
#include <iostream>

int main()
{
    int scores[] = { 84, 92, 76, 81, 56 };
```

```
    const int numStudents = sizeof(scores) / sizeof(scores[0]);

    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;
}
```

In this example, we use a non-loop variable called maxScore to keep track of the highest score we've seen. maxScore is initialized to 0 to represent that we have not seen any scores yet. We then iterate through each element of the array, and if we find a score that is higher than any we've seen before, we set maxScore to that value. Thus, maxScore always represents the highest score out of all the elements we've searched so far. By the time we reach the end of the array, maxScore holds the highest score in the entire array.

Mixing loops and arrays

Loops are typically used with arrays to do one of three things:
1) Calculate a value (e.g. average value, total value)
2) Search for a value (e.g. highest value, lowest value).
3) Reorganize the array (e.g. ascending order, descending order)

When calculating a value, a variable is typically used to hold an intermediate result that is used to calculate the final value. In the above example where we are calculating an average score, totalScore holds the total score for all the elements examined so far.

When searching for a value, a variable is typically used to hold the best candidate value seen so far (or the array index of the best candidate). In the above example where we use a loop to find the best score, maxScore is used to hold the highest score encountered so far.

Sorting an array is a bit more tricky, as it typically involves nested loops. We will cover sorting an array in the next lesson.

Arrays and off-by-one errors

One of the trickiest parts of using loops with arrays is making sure the loop iterates the proper number of times. Off-by-one errors are easy to make, and trying to access an element that is larger than the length of the array can have dire consequences. Consider the following program:

```
    int scores[] = { 84, 92, 76, 81, 56 };
    const int numStudents = sizeof(scores) / sizeof(scores[0]);

    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';
```

The problem with this program is that the conditional in the for loop is wrong! The array declared has 5 elements, indexed from 0 to 4. However, this array loops from 0 to 5. Consequently, on the last iteration, the array will execute this:

```
        if (scores[5] > maxScore)
            maxScore = scores[5];
```

But scores[5] is undefined! This can cause all sorts of issues, with the most likely being that scores[5] results in a garbage value. In this case, the probable result is that maxScore will be wrong.

However, imagine what would happen if we inadvertently assigned a value to array[5]! We might overwrite another variable (or part of it), or perhaps corrupt something -- these types of bugs can be very hard to track down!

Consequently, when using loops with arrays, always double-check your loop conditions to make sure you do not introduce off-by-one errors.

Quiz

1) Print the following array to the screen using a loop:

```
    int array[] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
```

Hint: You can use the sizeof() trick to determine the array length.

2) Given the array in question 1:

Ask the user for a number between 1 and 9. If the user does not enter a number between 1 and 9, repeatedly ask for an integer value until they do. Once they have entered a number between 1 and 9, print the array. Then search the array for the value that the user entered and print the index of that element.

You can test std::cin for invalid input by using the following code:

```
// if the user entered something invalid
if (std::cin.fail())
{
    std::cin.clear(); // reset any error flags
    std::cin.ignore(32767, '\n'); // ignore any characters in the input buffer
}
```

3) Modify the following program so that instead of having maxScore hold the largest score directly, a variable named maxIndex holds the index of the largest score.

```
#include <iostream>

int main()
{
    int scores[] = { 84, 92, 76, 81, 56 };
    const int numStudents = sizeof(scores) / sizeof(scores[0]);

    int maxScore = 0; // keep track of our largest score

    // now look for a larger 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;
}
```

Quiz solutions

1) **Hide Solution**

```
#include <iostream>
int main()
{
    int array[] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
    const int arrayLength = sizeof(array) / sizeof(array[0]);

    for (int index=0; index < arrayLength; ++index)
        std::cout << array[index] << " ";
    return 0;
}
```

2) **Hide Solution**

```
#include <iostream>

int main()
{
    // First, read in valid input from user
    int number = 0;
    do
    {
```

```
        std::cout << "Enter a number between 1 and 9: ";
        std::cin >> number;

        // if the user entered an invalid character
        if (std::cin.fail())
            std::cin.clear(); // reset any error flags

        std::cin.ignore(32767, '\n'); // ignore any extra characters in the input buffer

    } while (number < 1 || number > 9);

    // Next, print the array
    int array[] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
    const int arrayLength = sizeof(array) / sizeof(array[0]);

    for (int index=0; index < arrayLength; ++index)
        std::cout << array[index] << " ";

    std::cout << "\n";

    // Then, search the array to find the matching number and print the index
    for (int index=0; index < arrayLength; ++index)
    {
        if (array[index] == number)
        {
            std::cout <<  "The number " << number << " has index " <<  index << "\n";
            break; // since each # in the array is unique, no need to search rest of array
        }
    }

    return 0;
}
```

3) **Hide Solution**

```
#include <iostream>

int main()
{
    int scores[] = { 84, 92, 76, 81, 56 };
    const int numStudents = sizeof(scores) / sizeof(scores[0]);

    int maxIndex = 0; // keep track of index of our largest score

    // now look for a larger score
    for (int student = 0; student < numStudents; ++student)
        if (scores[student] > scores[maxIndex])
            maxIndex = student;

    std::cout << "The best score was " << scores[maxIndex] << '\n';

    return 0;
}
```


6.4 -- Sorting an array using selection sort

Index

6.2 -- Arrays (Part II)