6.2 — Arrays (Part II)

BY ALEX ON JUNE 27TH, 2007 | LAST MODIFIED BY ALEX ON MARCH 20TH, 2018

This lesson continues the discussion of arrays that began in lesson **6.1 -- Arrays (part I)**

Initializing fixed arrays

Array elements are treated just like normal variables, and as such, they are not initialized when created.

One way to initialize an array is to do it element by element:

```
    int prime[5]; // hold the first 5 prime numbers
    prime[0] = 2;
    prime[1] = 3;
    prime[2] = 5;
    prime[3] = 7;
    prime[4] = 11;
```

However, this is a pain, especially as the array gets larger.

Fortunately, C++ provides a more convenient way to initialize entire arrays via use of an **initializer list**. The following example is equivalent to the one above:

```
int prime[5] = { 2, 3, 5, 7, 11 }; // use initializer list to initialize the fixed array
```

If there are more initializers in the list than the array can hold, the compiler will generate an error.

However, if there are less initializers in the list than the array can hold, the remaining elements are initialized to 0. The following example shows this in action:

```
#include <iostream>

int main()
{
    int array[5] = { 7, 4, 5 }; // only initialize first 3 elements

    std::cout << array[0] << '\n';
    std::cout << array[1] << '\n';
    std::cout << array[2] << '\n';
    std::cout << array[3] << '\n';
    std::cout << array[4] << '\n';

    return 0;
}
```

This prints:

```
7
4
5
0
0
```

Consequently, to initialize all the elements of an array to 0, you can do this:

```
// Initialize all elements to 0
int array[5] = { };
```

In C++11, the uniform initialization syntax can be used instead:

```
int prime[5] { 2, 3, 5, 7, 11 }; // use uniform initialization to initialize the fixed array
// note the lack of the equals sign in this syntax
```

Omitted length

If you are initializing a fixed array of elements using an initializer list, the compiler can figure out the length of the array for you, and you can omit explicitly declaring the length of the array.

The following two lines are equivalent:

```
int array[5] = { 0, 1, 2, 3, 4 }; // explicitly define length of the array
int array[] = { 0, 1, 2, 3, 4 }; // let initializer list set length of the array
```

This not only saves typing, it also means you don't have to update the array length if you add or remove elements later.

Arrays and enums

One of the big documentation problems with arrays is that integer indices do not provide any information to the programmer about the meaning of the index. Consider a class of 5 students:

```
const int numberOfStudents(5);
int testScores[numberOfStudents];
testScores[2] = 76;
```

Who is represented by testScores[2]? It's not clear.

This can be solved by setting up an enumeration where one enumerator maps to each of the possible array indices:

```
enum StudentNames
{
    KENNY, // 0
    KYLE, // 1
    STAN, // 2
    BUTTERS, // 3
    CARTMAN, // 4
    MAX_STUDENTS // 5
};

int main()
{
    int testScores[MAX_STUDENTS]; // allocate 5 integers
    testScores[STAN] = 76;

    return 0;
}
```

In this way, it's much clearer what each of the array elements represents. Note that an extra enumerator named MAX_STUDENTS has been added. This enumerator is used during the array declaration to ensure the array has the proper length (as the array length should be one greater than the largest index). This is useful both for documentation purposes, and because the array will automatically be resized if another enumerator is added:

```
enum StudentNames
{
    KENNY, // 0
    KYLE, // 1
    STAN, // 2
    BUTTERS, // 3
    CARTMAN, // 4
    WENDY, // 5
    MAX_STUDENTS // 6
};

int main()
{
    int testScores[MAX_STUDENTS]; // allocate 6 integers
    testScores[STAN] = 76; // still works

    return 0;
}
```

Note that this “trick” only works if you do not change the enumerator values manually!

Arrays and enum classes

Enum classes don’t have an implicit conversion to integer, so if you try the following:

```
enum class StudentNames
{
    KENNY, // 0
    KYLE, // 1
    STAN, // 2
    BUTTERS, // 3
    CARTMAN, // 4
    WENDY, // 5
    MAX_STUDENTS // 6
};

int main()
{
    int testScores[StudentNames::MAX_STUDENTS]; // allocate 6 integers
    testScores[StudentNames::STAN] = 76;
}
```

You’ll get a compiler error. This can be addressed by using a static_cast to convert the enumerator to an integer:

```
int main()
{
    int testScores[static_cast<int>(StudentNames::MAX_STUDENTS)]; // allocate 6 integers
    testScores[static_cast<int>(StudentNames::STAN)] = 76;
}
```

However, doing this is somewhat of a pain, so it might be better to use a standard enum inside of a namespace:

```
namespace StudentNames
{
    enum StudentNames
    {
        KENNY, // 0
        KYLE, // 1
        STAN, // 2
        BUTTERS, // 3
        CARTMAN, // 4
        WENDY, // 5
        MAX_STUDENTS // 6
    };
}

int main()
{
    int testScores[StudentNames::MAX_STUDENTS]; // allocate 6 integers
    testScores[StudentNames::STAN] = 76;
}
```

Passing arrays to functions

Although passing an array to a function at first glance looks just like passing a normal variable, underneath the hood, C++ treats arrays differently.

When a normal variable is passed by value, C++ copies the value of the argument into the function parameter. Because the parameter is a copy, changing the value of the parameter does not change the value of the original argument.

However, because copying large arrays can be very expensive, C++ does *not* copy an array when an array is passed into a function. Instead, the *actual* array is passed. This has the side effect of allowing functions to directly change the value of array elements!

The following example illustrates this concept:

```
#include <iostream>
```

```
2
3  void passValue(int value) // value is a copy of the argument
4  {
5      value = 99; // so changing it here won't change the value of the argument
6  }
7
8  void passArray(int prime[5]) // prime is the actual array
9  {
10     prime[0] = 11; // so changing it here will change the original argument!
11     prime[1] = 7;
12     prime[2] = 5;
13     prime[3] = 3;
14     prime[4] = 2;
15 }
16
17 int main()
18 {
19     int value = 1;
20     std::cout << "before passValue: " << value << "\n";
21     passValue(value);
22     std::cout << "after passValue: " << value << "\n";
23
24     int prime[5] = { 2, 3, 5, 7, 11 };
25     std::cout << "before passArray: " << prime[0] << " " << prime[1] << " " << prime[2] << " " << prime
26 [3] << " " << prime[4] << "\n";
27     passArray(prime);
28     std::cout << "after passArray: " << prime[0] << " " << prime[1] << " " << prime[2] << " " << prime[
29 3] << " " << prime[4] << "\n";
30
       return 0;
   }
```

```
before passValue: 1
after passValue: 1
before passArray: 2 3 5 7 11
after passArray: 11 7 5 3 2
```

In the above example, value is not changed in main() because the parameter value in function passValue() was a copy of variable value in function main(), not the actual variable. However, because the parameter array in function passArray() is the actual array, passArray() is able to directly change the value of the elements!

Why this happens is related to the way arrays are implemented in C++, a topic we'll revisit once we've covered pointers. For now, you can consider this as a quirk of the language.

As a side note, if you want to ensure a function does not modify the array elements passed into it, you can make the array const:

```
1  // even though prime is the actual array, within this function it should be treated as a constant
2  void passArray(const int prime[5])
3  {
4      // so each of these lines will cause a compile error!
5      prime[0] = 11;
6      prime[1] = 7;
7      prime[2] = 5;
8      prime[3] = 3;
9      prime[4] = 2;
10 }
```

sizeof and arrays

The sizeof operator can be used on arrays, and it will return the total size of the array (array length multiplied by element size). Note that due to the way C++ passes arrays to functions, this will _not_ work properly for arrays that have been passed to functions!

```
1  void printSize(int array[])
2  {
3      std::cout << sizeof(array) << '\n'; // prints the size of a pointer, not the size of the array!
```

```
}

int main()
{
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << sizeof(array) << '\n'; // will print the size of the array
    printSize(array);

    return 0;
}
```

On a machine with 4 byte integers and 4 byte pointers, this printed:

```
32
4
```

(You may get a slightly different result if the size of your types are different).

For this reason, be careful about using sizeof() on arrays!

Note: In common usage, the terms “array size” and “array length” are both most often used to refer to the array’s length (the size of the array isn’t useful in most cases, outside of the trick we’ve shown you above). In these tutorials, we’ll try to use the term “length” when we’re talking about the number of elements in the array, and “size” when we’re referring to how large something is in bytes.

Determining the length of a fixed array

One neat trick: we can determine the length of a fixed array by dividing the size of the entire array by the size of an array element:

```
int main()
{
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << "The array has: " << sizeof(array) / sizeof(array[0]) << "elements\n";

    return 0;
}
```

This prints:

```
The array has 8 elements
```

How does this work? First, note that the size of the entire array is equal to the array’s length multiplied by the size of an element. Put more compactly: array size = array length * element size.

Using algebra, we can rearrange this equation: array length = array size / element size. sizeof(array) is the array size, and sizeof(array[0]) is the element size, so our equation becomes array length = sizeof(array) / sizeof(array[0]). We typically use array element 0 for the array element, since it’s the only element guaranteed to exist no matter what the array length is.

Note that this will only work if the array is a fixed-length array, and you’re doing this trick in the same function that array is declared in (we’ll talk more about why this restriction exists in a future lesson in this chapter).

Indexing an array out of range

Remember that an array of length N has array elements 0 through N-1. So what happens if you try to access an array with a subscript outside of that range?

Consider the following program:

```
int main()
{
    int prime[5]; // hold the first 5 prime numbers
    prime[5] = 13;

    return 0;
}
```

In this program, our array is of length 5, but we're trying to write a test score into the 6th element (index 5).

C++ does *not* do any checking to make sure that your indices are valid for the length of your array. So in the above example, the value of 13 will be inserted into memory where the 6th element would have been had it existed. When this happens, you will get undefined behavior -- For example, this could overwrite the value of another variable, or cause your program to crash.

Although it happens less often, C++ will also let you use a negative index, with similarly undesirable results.

Rule: When using arrays, ensure that your indices are valid for the range of your array!.

Quiz

1) Declare an array to hold the high temperature (to the nearest tenth of a degree) for each day of a year (assume 365 days in a year). Initialize the array with a value of 0.0 for each day.

2) Set up an enum with the names of the following animals: chicken, dog, cat, elephant, duck, and snake. Put the enum in a namespace. Define an array with an element for each of these animals, and use an initializer list to initialize each element to hold the number of legs that animal has.

Write a main function that prints the number of legs an elephant has, using the enumerator.

Quiz answers

1) **Hide Solution**

```
double temperature[365] = { 0.0 };
```

Note: when space isn't a constraint, doubles should be preferred over floats.

2) **Hide Solution**

```
namespace Animals
{
    enum Animals
    {
        CHICKEN,
        DOG,
        CAT,
        ELEPHANT,
        DUCK,
        SNAKE,
        MAX_ANIMALS
    };
}

int main()
{
    int legs[Animals::MAX_ANIMALS] = { 2, 4, 4, 4, 2, 0 };

    std::cout << "An elephant has " << legs[Animals::ELEPHANT] << " legs.\n";

    return 0;
}
```



6.3 -- Arrays and loops


Index


6.1 -- Arrays (Part I)