10.1 — Arrays (Part I)

ALEX AUGUST 17, 2021

Note: This chapter is a bit harder than the previous ones. If you feel a little discouraged, stick with it. The best stuff is yet to come!

In lesson 9.4 -- Structs, you learned that you can use a struct to aggregate many different data types into one identifier. This is great for the case where we want to model a single object that has many different properties. However, this is not so great for the case where we want to track many related instances of something.

Fortunately, structs are not the only aggregate data type in C++. An array is an aggregate data type that lets us access many variables of the same type through a single identifier.

Consider the case where you want to record the test scores for 30 students in a class. Without arrays, you would have to allocate 30 almost-identical variables!

```
// allocate 30 integer variables (each with a different name)
int testScoreStudent1{};
int testScoreStudent2{};
int testScoreStudent3{};
// ...
int testScoreStudent30{};
```

Arrays give us a much easier way to do this. The following array definition is essentially equivalent:

```
int testScore[30]{}; // allocate 30 integer variables in a fixed array
```

In an array variable declaration, we use square brackets ([]) to tell the compiler both that this is an array variable (instead of a normal variable), as well as how many variables to allocate (called the array length).

In the above example, we declare a fixed array named testScore, with a length of 30. A fixed array (also called a fixed length array or fixed size array) is an array where the length is known at compile time. When testScore is instantiated, 30 integers will be allocated.

Array elements and subscripting

Each of the variables in an array is called an element. Elements do not have their own unique names. Instead, to access individual elements of an array, we use the array name, along with the subscript operator ([]), and a parameter called a subscript (or index) that tells the compiler which element we want. This process is called subscripting or indexing the array.

In the example above, the first element in our array is testScore[0]. The second is testScore[1]. The tenth is testScore[9]. The last element in our testScore array is testScore[29]. This is great because we no longer need to keep track of a bunch of different (but related) names -- we can just vary the subscript to access different elements.

Important: Unlike everyday life, where we typically count starting from 1, in C++, arrays always count starting from 0!

For an array of length N, the array elements are numbered 0 through N-1. This is called the array's range.

An example array program

Here's a sample program that puts together the definition and indexing of an array:

```cpp
#include <iostream>

int main()
{
    int prime[5]{}; // hold the first 5 prime numbers
    prime[0] = 2; // The first element has index 0
    prime[1] = 3;
    prime[2] = 5;
    prime[3] = 7;
    prime[4] = 11; // The last element has index 4 (array length-1)

    std::cout << "The lowest prime number is: " << prime[0] << '\n';
    std::cout << "The sum of the first 5 primes is: " << prime[0] + prime[1] + prime[2] + prime[3] +
prime[4] << '\n';

    return 0;
}
```

This prints:

```
The lowest prime number is: 2
The sum of the first 5 primes is: 28
```

Array data types

Arrays can be made from any data type. Consider the following example, where we declare an array of doubles:

```cpp
#include <iostream>

int main()
{
    double batteryLifeInHours[3]{}; // allocate 3 doubles
    batteryLifeInHours[0] = 2.0;
    batteryLifeInHours[1] = 3.0;
    batteryLifeInHours[2] = 4.3;

    std::cout << "The average battery life is " << (batteryLifeInHours[0] + batteryLifeInHours[1] +
batteryLifeInHours[2]) / 3.0 << " hour(s)\n";

    return 0;
}
```

This program produces the result:

```
The average battery life is 3.1 hour(s)
```

Arrays can also be made from structs. Consider the following example:

```
struct Rectangle
{
    int length{};
    int width{};
};
Rectangle rects[5]{}; // declare an array of 5 Rectangle
```

To access a struct member of an array element, first pick which array element you want, and then use the member selection operator to select the struct member you want:

```
rects[0].length = 24;
```

Arrays can even be made from arrays, a topic that we'll cover in a future lesson.

Array subscripts

In C++, array subscripts must always be an integral type. This includes char, short, int, long, long long, etc... and strangely enough, bool (where false gives an index of 0 and true gives an index of 1). An array subscript can be a literal value, a variable (constant or non-constant), or an expression that evaluates to an integral type.

Here are some examples:

```
int array[5]{}; // declare an array of length 5

// using a literal (constant) index:
array[1] = 7; // ok

// using an enum (constant) index
enum Animals
{
    animal_cat = 2
};
array[animal_cat] = 4; // ok

// using a variable (non-constant) index:
int index{ 3 };
array[index] = 7; // ok

// using an expression that evaluates to an integer index:
array[1+2] = 7; // ok
```

Fixed array declarations

When declaring a fixed array, the length of the array (between the square brackets) must be a compile-time constant. This is because the length of a fixed array must be known at compile time. Here are some different ways to declare fixed arrays:

```cpp
// using a literal constant
int numberOfLessonsPerDay[7]{}; // Ok

// using a constexpr symbolic constant
constexpr int daysPerWeek{ 7 };
int numberOfLessonsPerDay[daysPerWeek]{}; // Ok

// using an enumerator
enum Weekday
{
    monday,
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday,
    sunday,

    maxWeekday
};
int numberOfLessonsPerDay[maxWeekday]{}; // Ok

// using a macro
#define DAYS_PER_WEEK 7
int numberOfLessonsPerDay[DAYS_PER_WEEK]{}; // Works, but don't do this (use a constexpr symbolic
constant instead)
```

Note that non-const variables or runtime constants cannot be used:

```cpp
// using a non-const variable
int daysPerWeek{};
std::cin >> daysPerWeek;
int numberOfLessonsPerDay[daysPerWeek]{}; // Not ok -- daysPerWeek is not a compile-time constant!

// using a runtime const variable
int temp{ 5 };
const int daysPerWeek{ temp }; // the value of daysPerWeek isn't known until runtime, so this is a
runtime constant, not a compile-time constant!
int numberOfLessonsPerDay[daysPerWeek]{}; // Not ok
```

Note that in the last two cases, an error should result because the length is not a compile-time constant. Some compilers may allow these kinds of arrays (for C99 compatibility reasons), but they are invalid in C++, and should not be used in C++ programs. If your compiler allows these arrays, you probably forgot to disable compiler extensions (Lesson 0.10 -- Configuring your compiler: Compiler extensions).

A note on dynamic arrays

Because fixed arrays have memory allocated at compile time, that introduces two limitations:

- Fixed arrays cannot have a length based on either user input or some other value calculated at runtime.
- Fixed arrays have a fixed length that can not be changed.

In many cases, these limitations are problematic. Fortunately, C++ supports a second kind of array known as a dynamic array. The length of a dynamic array can be set at runtime, and their length can be changed. However, dynamic arrays are a little more complicated to instantiate, so we’ll cover them later in the chapter.

Summary

Fixed arrays provide an easy way to allocate and use multiple variables of the same type so long as the length of the array is known at compile time.

We’ll look at more topics around fixed arrays in the next lesson.

Next lesson
10.2 Arrays (Part II)

Back to table of contents

Previous lesson
9.x Chapter 9 summary and quiz

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X