6.8 — Pointers and arrays

BY ALEX ON JULY 11TH, 2007 | LAST MODIFIED BY ALEX ON JULY 26TH, 2017

Pointers and arrays are intrinsically related in C++.

Similarities between pointers and fixed arrays

In lesson **6.1 -- Arrays (part i)**, you learned how to define a fixed array:

```
int array[5] = { 9, 7, 5, 3, 1 }; // declare a fixed array of 5 integers
```

To us, the above is an array of 5 integers, but to the compiler, array is a variable of type int[5]. We know what the values of array[0], array[1], array[2], array[3], and array[4] are (9, 7, 5, 3, and 1 respectively). But what value does array itself have?

The variable array contains the address of the first element of the array, as if it were a pointer! You can see this in the following program:

```
#include <iostream>

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };

    // print the value of the array variable
    std::cout << "The array has address: " << array << '\n';

    // print address of the array elements
    std::cout << "Element 0 has address: " << &array[0] << '\n';

    return 0;
}
```

On the author's machine, this printed:

```
The array has address: 0042FD5C
Element 0 has address: 0042FD5C
```

Note that the address held by the array variable is the address of the first element of the array.

It's a common fallacy in C++ to believe an array and a pointer to the array are identical. They're not. Although both point to the first element of the array, they have different type information. In the above case, array is of type "int[5]", whereas a pointer to the array would be of type "int *". We'll see where this makes a difference shortly.

The confusion is primarily caused by the fact that in many cases, when evaluated, a fixed array will **decay** (be implicitly converted) into a pointer to the first element of the array. All elements of the array can still be accessed through the pointer, but information derived from the array's type (such as how long the array is) can not be accessed from the pointer.

However, this also effectively allows us to treat fixed arrays and pointers identically in most cases.

For example, we can dereference the array to get the value of the first element:

```
int array[5] = { 9, 7, 5, 3, 1 };

// dereferencing an array returns the first element (element 0)
cout << *array; // will print 9!

char name[] = "Jason"; // C-style string (also an array)
cout << *name; // will print 'J'
```

Note that we're not *actually* dereferencing the array itself. The array (of type int[5]) gets implicitly converted into a pointer (of type int *), and we dereference the pointer to get the value at the memory address the pointer is holding (the value of the first element of the array).

We can also assign a pointer to point at the array:

```cpp
#include <iostream>

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };
        std::cout << *array; // will print 9

        int *ptr = array;
        std::cout << *ptr; // will print 9

    return 0;
}
```

This works because the array decays into a pointer of type int *, and our pointer (also of type int *) has the same type.

Differences between pointers and fixed arrays

There are a few cases where the difference in typing between fixed arrays and pointers makes a difference. These help illustrate that a fixed array and a pointer are not the same.

The primary difference occurs when using the sizeof() operator. When used on a fixed array, sizeof returns the size of the entire array (array length * element size). When used on a pointer, sizeof returns the size of a memory address (in bytes). The following program illustrates this:

```cpp
#include <iostream>

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };

    std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length

        int *ptr = array;
        std::cout << sizeof(ptr) << '\n'; // will print the size of a pointer

    return 0;
}
```

This program prints:

```
20
4
```

A fixed array knows how long the array it is pointing to is. A pointer to the array does not.

The second difference occurs when using the address-of operator (&). Taking the address of a pointer yields the memory address of the pointer variable. Taking the address of the array returns a pointer to the entire array. This pointer also points to the first element of the array, but the type information is different (in the above example, int(*)[5]). It's unlikely you'll ever need to use this.

Revisiting passing fixed arrays to functions

Back in lesson **6.2 -- Arrays (part ii)**, we mentioned that because copying large arrays can be very expensive, C++ does not copy an array when an array is passed into a function. When passing an array as an argument to a function, a fixed array decays into a pointer, and the pointer is passed to the function:

```cpp
#include <iostream>

void printSize(int *array)
{
    // array is treated as a pointer here
    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 sizeof(int) * array length

    printSize(array); // the array argument decays into a pointer here

     return 0;
}
```

This prints:

```
32
4
```

Note that this happens even if the parameter is declared as a fixed array:

```
#include <iostream>

// C++ will implicitly convert parameter array[] to *array
void printSize(int array[])
{
    // array is treated as a pointer here, not a fixed array
    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 sizeof(int) * array length

    printSize(array); // the array argument decays into a pointer here

     return 0;
}
```

This prints:

```
32
4
```

In the above example, C++ implicitly converts parameters using the array syntax ([]) to the pointer syntax (*). That means the following two function declarations are identical:

```
void printSize(int array[]);
void printSize(int *array);
```

Some programmers prefer using the [] syntax because it makes it clear that the function is expecting an array, not just a pointer to a value. However, in most cases, because the pointer doesn't know how large the array is, you'll need to pass in the array size as a separate parameter anyway (strings being an exception because they're null terminated).

We lightly recommend using the pointer syntax, because it makes it clear that the parameter is being treated as a pointer, not a fixed array, and that certain operations, such as sizeof(), will operate as if the parameter is a pointer.

Recommendation: Favor the pointer syntax (*) over the array syntax ([]) for array function parameters.

An intro to pass by address

The fact that arrays decay into pointers when passed to a function explains the underlying reason why changing an array in a function changes the actual array argument passed in. Consider the following example:

```
#include <iostream>

// parameter ptr contains a copy of the array's address
void changeArray(int *ptr)
```

```
{
    *ptr = 5; // so changing an array element changes the _actual_ array
}

int main()
{
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << "Element 0 has value: " << array[0] << '\n';

    changeArray(array);

    std::cout << "Element 0 has value: " << array[0] << '\n';

     return 0;
}
```

```
Element 0 has value: 1
Element 0 has value: 5
```

When changeArray() is called, array decays into a pointer, and the value of that pointer (the memory address of the first element of the array) is copied into the ptr parameter of function changeArray(). Although the value in ptr is a copy of the address of the array, ptr still points at the actual array (not a copy!). Consequently, when ptr is dereferenced, the actual array is dereferenced!

Astute readers will note this phenomena works with pointers to non-array values as well. We'll cover this topic (called passing by address) in more detail in the next chapter.

Arrays in structs and classes don't decay

Finally, it is worth noting that arrays that are part of structs or classes do not decay when the whole struct or class is passed to a function. This yields a useful way to prevent decay if desired, and will be valuable later when we write classes that utilize arrays.

In the next lesson, we'll take a look at pointer arithmetic, and talk about how array indexing actually works.

6.8a -- Pointer arithmetic and array indexing

Index

6.7a -- Null pointers

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

129 comments to 6.8 — Pointers and arrays

« Older Comments 1 2

David
April 2, 2018 at 4:54 pm · Reply

Hi! Suppose I want to assign an array to another array. Something like this doesn't work: