Arrays

Arrays and pointers

Michael Burrell

March 12, 2024

Textbook readings

Chapter 2 — 2.3.2

Chapter 3 — 3.5, 3.6

Chapter 4 — 4.9

Chapter 6 — 6.1, 6.2

Goals for this set of slides

- Understand how to declare arrays
- Understand how to use `size_t` with arrays
- Understand the relationship between arrays and pointers
- Be able to use arrays with functions

Arrays

- An array is a sequence of elements
- Each element has the same type
- The elements are laid out contiguously in memory
  - One element is right beside the next element in memory

Operations on arrays

The basic operations which we will consider on arrays are:

Create an array — initially we will do this by declaring a new variable. Later in the course, we will see other ways

Set values in an array — by using the `[ ]` operator, like in Python

Retrieving values from an array — again, by using the `[ ]` operator, like in Python

Iterating through an array — we'll see a few ways of doing this

Declaring an array


```
1 int x[] = { 9, 7, 5, 3, 4 };
2 int y[5];
```

- These are generally the two most common ways of declaring an array
- The first has all values initialized and the second has no values initialized
- The length of the array *should be* a constant (fixed)
- The array may not be resized after the array is declared
- If declared inside a function, the array is declared on the call stack

Initialization to zero


```
1  int z[12] = { 3, 4, 5 };
```

- It is possible to give an incomplete initialization
- The compiler will initailize all other elements to 0
- The above is equivalent to:

```
1  int z[12] = { 3, 4, 5, 0, 0, 0, 0, 0, 0, 0, 0,
       0 };
```

Setting values

After being declared, individual elements can be set using an index (of any integer type, often `size_t`).

```
1 int x[5] = { 2, 3 };
2 x[3] = 1;
3 x[2] = 4;
4 for (size_t i = 0; i < 5; i++) {
5     x[i] = x[x[i]];
6 }
```

Retrieving

- Like with setting, retrieving a value from an array uses the `[ ]` operator
- Any index of any integer type may used as an index

```
1  int y = x[3];
```

Iterating

- We can iterate through an array using a simple `for` loop
- Knowing when to stop looping can be tricky, though....

```
1 int x[] = { 9, 12, 3, 6 };
2 int sum = 0;
3 for (size_t i = 0; i < 4; i++) {
4     sum += x[i];
5 }
```

Calculating the size of an array

- We can't use `len()` or `.length` to find the length of an array
- However, we *can* use some trickery to get the compiler to reveal to us the number of elements
- `sizeof x` tells us the number of characters (bytes) in the entire array
- `sizeof x[0]` tells us the number of characters (bytes) in one element of the array

Calculating the size of an array


```
1  int x[] = { 9, 12, 3, -6, 14, 8, 7, 3 };
2  for (size_t i = 0; i < sizeof x / sizeof x[0]; i++) {
3      cout << x[i] << endl;
4  }
```

Example

Class exercise

Let's write a program which finds the most commonly-occurring element in a list.

Conclusion

- Arrays are homogeneous (all of the same element type)
- Arrays are contiguous (each element beside the next in memory)
- Most basic operations are about the same between C++ and Python
- Things are about to get complicated. We need to understand how arrays actually work...

Pointer

- A *pointer* is a memory address
- In C++ (unlike in Python), we can assign a pointer to a variable and be explicit about what memory operations to perform via that address
- We will not be understanding pointers in *full* at this point in the course

Memory addresses

- Every value/variable in C++ has an address in memory
- If we know the address of a variable (using a pointer), we can manipulate that value indirectly
- Most of what happens in programming (regardless of language) happens via indirection/pointer operations
  - Some languages (assembly, machine code, C, C++, Rust) make this explicit
  - Some languages (Java, Python, Javascript) make this implicit

C++ syntax

- A warning first that the syntax for pointers in C++ is confusing
- $*$ and & each have (at least) *two* meanings with regards to pointers
  - $*$ in a type name and $*$ in an expression have two totally different, contradictory meanings
- **Your #1 goal at this point in the course is to become comfortable with what $*$ and & mean in different contexts**

& (in an expression)

- &, when used in an expression means "take the address of"
- &'s operand should be a variable[1]

```
1 int x = 5;
2 cout << x << endl << &x << endl;
```

The output of this program is, e.g.,:

```
5                 (<-- the value of x)
0x7fffe0311f84    (<-- the memory address of x)
```

[1] Technically an *lvalue*, meaning something that could appear on the left-hand side of an = sign

& (in a type name)

- In C++, & can be used in the name of a type
- We are *not* considering that at this point in the course
- I want to prevent confusion and limit the amount of new syntax
- We will see & in type names later in the course
- For now, & will only be used in expressions

Using &

- Every variable has a memory address
- This means every variable can be used with &
- The compiler doesn't generate any code for &
  - It simply uses its memory address (which it knows) instead of generating code to load its value

*

* has two different meanings

1. When used in an expression, it is the opposite of &
   - Its operand is a memory address, and it finds the value at that memory address (*dereferences* the memory address)
2. When used in a type, it means "pointer to"
   - We'll see examples of this in a couple slides

* counteracting &


```
1 int x = 5;
2 cout << x << endl
3     << &x << endl
4     << *&x << endl;
```

Output is:

```
5
0x7ffc9284bf34
5
```

We can see that * gets the value at the memory address &x.

Memory locations changing

- Note: every time you run your program, you will likely see the memory locations of variable changing
- Modern operating systems try to make the locations of your variables unpredictable for security reasons
- Don't pay attention to the actual numeric value of your pointers
- Just know that the compiler is able to figure out where variables are in memory

Pointer types

- We've seen how to create a pointer using &
- We've seen how to use a pointer using *

Pointer types

- We've seen how to create a pointer using &
- We've seen how to use a pointer using *
- Now, to store a pointer in a variable we need a new type, which also uses *
  - `int*` is a "pointer to an int"
  - Any variable of a * type is a pointer (memory address)
  - *** in a type name is <u>not</u> the same as * in an expression**

Storing pointers in a variable


```
1  int x = 5;     // x is of type "int"
2  int* y = &x;   // y is of type "pointer to int"
3  cout << x << endl
4       << y << endl
5       << *y << endl;
```

Output is:

```
5
0x7ffc9284bf34
5
```

* and &


```
int* y = &x;
cout << *y;
```

- Declare a new variable `y`
- `y`'s type is "pointer to `int`"
- Initialize `y` to be the memory address of `x`
- Then, print out `*y`
- `*y` is the `int` value stored in the memory address pointed to by `y`

Changing a value through a pointer


```
1 int x = 5;
2 int* y = &x;
3 *y = 7;
4 cout << x << endl;
```

The output is:

Changing a value through a pointer


```
1 int x = 5;
2 int* y = &x;
3 *y = 7;
4 cout << x << endl;
```

The output is:

7

Conclusion

- We can get the memory address of a value
- We can store that memory address in a variable (a *pointer*)
- We can get/change values using pointers

Arrays and pointers

- We talked about pointers for a while just to understand the close relationship that arrays and pointers have in C and C++
- In most contexts, an array *decays into a pointer to its first element*
- Pointers and arrays are interchangeable in many contexts (though not *all* contexts
- In order to use arrays effectively, we have to be able to use pointers effectively

Decay example


```
1 int x[] = { 2, 3, 4, 5 };
2 cout << x << endl;
```

The output is:

0x7fff1e7f06a0

Why?

Arrays in memory

- In the previous slide, we tried to print out the *array* `x`
- We got a *memory address* of 0x7fff1e7f06a0

[2] Assuming `int` has size 4

Arrays in memory

- In the previous slide, we tried to print out the *array* `x`
- We got a *memory address* of 0x7fff1e7f06a0
- Actually 0x7fff1e7f06a0 was the memory address of the first element of `x`
- 0x7fff1e7f06a4 is the memory address of the second[2] element of `x`
- 0x7fff1e7f06a8 is the memory address of the third element of `x`
- ...

[2] Assuming `int` has size 4

Decay example


```
1  int x[] = { 2, 3, 4, 5 };
2  cout << x << endl
3      << &x[0] << endl
4      << &x[1] << endl
5      << &x[2] << endl
6      << &x[3] << endl;
```

The output is:

```
0x7ffca35d1f10
0x7ffca35d1f10
0x7ffca35d1f14
0x7ffca35d1f18
0x7ffca35d1f1c
```

Note `x` and `&x[0]` are *exactly* the same!

From pointers to arrays

- We just saw that arrays decay into pointers
- This happens in almost every operation that's performed with arrays
- There is an equivalence that works in the other direction
  - If we have a pointer, we can treat it as an array, sort of

Using a pointer


```
1 int x = 5;
2 int* y = &x;
3 cout << y[0] << endl;
```

The output is:

5

Pointers as arrays

- In C++, pointers and arrays have a tight relationship due to the fact that that's how arrays work at the machine level
- In assembly/machine code, arrays are just values at a memory location, and array operations are loading from a memory location
- This is going to become useful for us very soon

Pointers as arrays

- In C++, pointers and arrays have a tight relationship due to the fact that that's how arrays work at the machine level
- In assembly/machine code, arrays are just values at a memory location, and array operations are loading from a memory location
- This is going to become useful for us very soon
  - It is not possible to pass an array as an argument to a function

Pointers as arrays

- In C++, pointers and arrays have a tight relationship due to the fact that that's how arrays work at the machine level
- In assembly/machine code, arrays are just values at a memory location, and array operations are loading from a memory location
- This is going to become useful for us very soon
  - It is not possible to pass an array as an argument to a function
  - But it is possible to pass a pointer...

Conclusion

- Arrays are stored contiguously in memory
- Arrays decay into pointers
- Pointers can be used to do array operations
- We will need to be comfortable with this relationship when using function calls with arrays