Accessing an Array

We can access the elements of an array in a couple different ways (which are actually the same under the hood!). We have already learned that the name of the array is a pointer to its first element, that we can do arithmetic on pointers, and that we can dereference pointers to access the data at the end of the arrow. We can put these concepts together to see one way that we could access elements in the array.

Accessing an array element using pointer arithmetic works fine, and sometimes is the natural way to access elements. However, sometimes we just want the *nth* element of an array, and it would be cumbersome to declare a pointer, add *n* to it, then dereference it. We can accomplish this goal more succinctly by indexing the array. When we index an array, we write the name of the array, followed by square brackets containing the number of the element we want to refer to: e.g., *myArray*[3]. Indexing an array names the specific box within the sequence, and can be used as either an lvalue or an rvalue. It is important to note that in C (and C++ and Java), array indexes are *zero-based* — the first element of the array is *myArray*[0]. Some programming languages have one-based array indexing, but zero-based is generally more common.

We will note that accessing an array out of bounds (at any element that does not exist) is an error that the compiler cannot detect. If you write such code, your program will access some box, but you do not know what box it actually is. This behavior is much the same as the erroneous behavior we discussed when we talked about pointer arithmetic. In fact, pointer arithmetic and array indexing are exactly the same under the hood, the compiler turns *myArray*[*i*] into *(*myArray* + *i*) (This definition leads to a "stupid C trick" that you can use to perplex your friends: *i*[*myArray*] looks ridiculous, but is perfectly legal. Why? *i*[*myArray*] = *(*i* + *myArray*). Since addition is commutative, that is the same as *(*myArray* + *i*), which is the same as *myArray*[*i*]. This trivia is completely useless beyond amazing your friends at parties and winning bets with people who know a little C, but not this fact.).

Note that a consequence of this rule is that if we take &myArray[i], it is equivalent to *&**(*myArray* +*i*), and the *&* and * cancel (as we learned previously, they are inverse operators), so it is just *myArray* + *i*. This result is fortunate, as it lines up with what we would hope for: *&myArray*[*i*] says "give me an arrow pointing at the *ith* box of *myArray*" while *myArray* + *i* says "give me a pointer i boxes after where *myArray* points"—these are two different ways to describe the same thing.

✓

Completed