In which our hero learns about a built-in collection offered by C known as the array, marvels at its simplicity, and, because this is C, worries about its dangers.

18.1 Collections

Thus far we’ve only looked at single variables: an integer variable named \( i \) to count how far we’ve progressed in a loop, another variable named \( age \) to represent the age of some person. We can even check the \( age \) by comparing it to a number like 55; and, as the saying goes, if \( age > 55 \), well, you’re getting kind of old, pal. Have you really done all the things you’ve wanted to do? If not, put this book down and get to it, friend! Life is short, or so they claim.

However, many interesting programs don’t contain data only about a single thing but rather about many things. For example, imagine a program that tracks the ages of many people (not just one). With simple variables like \( age \), this program is hard to write; you could add variable \( age_1 \) for the first person, \( age_2 \) for the second, but how do you write code that can handle a large and potentially changing group of people?

For this reason, programming languages all have methods to enable collections of some kind, i.e., data structures that contain many different things inside them instead of just one. C, in particular, has one useful and ubiquitous collection type that is built in, the array.

In this chapter, we’ll introduce the array, including how to define and use one. We’ll also show how C arrays share a lot of similarities to C pointers, which can be confusing, but, once understood, is actually quite reasonable. Finally, we’ll close with some warnings about the dangers of sloppy array use. There are a lot of things you can be when you program with C, but sloppy ain’t one of ‘em, friend. Save your sloppiness for perl [C+12].
18.2 Defining An Array

Let’s start with the simplest of constructs, a single-dimensional array of integers. Later, we’ll see arrays of different kinds, but almost everything we learn about arrays of integers also applies to arrays of other types.

To define such a fixed-size array, you simply choose the type (e.g., `int`), a name (e.g., `a`), and the size of the array, surrounded by square brackets (`[]`). For example, the following code defines an array of ten integers on the stack:

```c
int main() {
    // defines array of 10 integers on the stack
    int a[10];
    ...
}
```

C ensures that an array defined as such is placed contiguously in memory, and this is a fact your program can rely upon if needed (as we’ll shortly see). Figure 18.1 shows the 10 integers of array `a` laid out in memory; we pretend here that `a` starts at address 1000 and occupies the next 40 bytes of memory, from 1000 through 1039 (inclusive). That’s it!

More generally, when you declare an array of size `N`, you are allocating \( N \times \text{sizeof(type)} \). Because we’ve only been focused on integers thus far, an array with `N` entries occupies exactly \( N \times 4 \) bytes.
18.3 Array Access

Defining an array is the first step, but we also need to be able to read and write its elements. Access is performed with a new operator, the array index operator denoted by square brackets ([ ]). Let’s look at a simple loop that sets the value of each array element:

```c
int main() {
    int a[10];

    // initialize - set each entry to some value
    int i;
    for (i = 0; i < 10; i++) {
        a[i] = i;
    }

    // ...}
```

As you can see, arrays are as easy to access as any variable; just specify the name of the array followed by the index (which is surrounded by brackets), and you can read or write array values with ease. The index, naturally, has to be an integer value, such as an integer literal (i.e., a number) directly in your code, a variable (like i in the example above), or in fact any expression that has an integer value (e.g., 1+foo(), assuming function foo() returns an integer value).

The precedence of the array access operator is high, making arrays natural to in expressions with other operators (see Figure 18.2). For example, the expression a[i] + b[i] will work as expected, as will a[i]++.
18.4 Address Calculation

When accessing an array by index, it is useful to understand that C is performing a little bit of math on your behalf to find the location of the desired data item. Specifically, if your code accesses an array \( a \) at index \( x \), the address of the entry is calculated as follows:

\[
\text{address}(x) = \text{start} + (x \cdot \text{element size});
\]

In this equation, \( \text{start} \) is the address of the start of the array, and \( \text{element size} \) is the size of each element (with integers, we’ll assume this size is 4).

If you examine Figure 18.1 (page 2), you can see how this calculation works. Assume we are looking to find where in memory \( a[5] \) resides. The starting address is 1000, the index 5, and thus the address of the desired data is \( 1000 + (5 \cdot 4) = 1000 + 20 = 1020 \), as seen in the picture.

18.5 Meet Array’s Cousin: The Pointer

In C, there is a little secret that all experienced programmers know about arrays, and it is this: in many (though not all ways), C arrays and C pointers are almost the exact same thing. Let’s look at some examples to make more sense of this amazing truth!

In this code snippet, we first assign a pointer \( p \) to point to the beginning of the array \( a \), and then update the first few entries of the array using the pointer:

```c
int main() {
    int a[4]; // 4-element array on stack
    int* p; // pointer p also on stack

    p = a; // assign p to beginning of array
    // could also have written this as:
    // p = &a[0];
    *p = 1; // this updates a[0] to 1
    p = p + 1; // increments p by 1 using ptr arith
    *p = 2; // this updates a[1] to 2
    ...
}
```

Figure 18.3 (page 5) shows how values in memory change as the code snippet above runs, assuming the 4-entry array is located at address 1008 on the stack, and that the pointer \( p \) is located just below it at address 1024. The figure highlights, in light gray, the memory location that is updated after each statement executes.

First, on the left, \( p \) is assigned to \( a \) (Line 5). Logically, we would say that \( p \) “points to” \( a \), whereas physically you can see that this just
means that the variable $p$ holds the address of the beginning of the array $a$ (i.e., $p$ is now equal to 1008).

Second, $p$ is dereferenced (i.e., $*p = 1$; Line 8), which in this usage sets the value of whatever $p$ points to the value 1 (Line 8). As you’ll recall, dereferencing a value on the left side of an assignment updates the value of the thing the pointer indirectly refers to.

Third, the variable $p$ itself is incremented (Line 9). This statement doesn’t affect the contents of the array, $a$; rather, it simply increments the contents of the container $p$. However, do recall that pointer arithmetic increments the pointer by the size of the type it is pointing to. Here, because $p$ is a pointer to an integer, incrementing it by one increases its size so as to point to the next pointer, and thus actually increments the value of $p$ by four.

Finally, the next element of the array ($a[1]$) is set to the value 2 (Line 10), once again by dereferencing $p$ (i.e., $*p=2$;).

As you gain comfort with pointer arithmetic, you’ll notice that you can directly refer to any element of an array using the following simple syntax. Specifically, assuming $p$ points to the beginning of an array $a$, you can access $a[i]$ with this small bit of code: $*(p+i)$. In English, we should say $*(p+i)$ first increments the pointer $p$ by $i$, and the dereferences it; note that the parentheses are necessary because the precedence of the dereference operator $*$ is higher than the precedence of the addition operator $+$.

We will informally call this sameness the **Law of Pointer/Array Equivalence** and we will say it in our deepest and most serious of voices. In fact, we will even write it down here so it stands out a bit in the text:

![Figure 18.3: Accessing An Array Via Pointer](image-url)
The Law of Pointer/Array Equivalence

\[ a[i] \Leftrightarrow *(p+i) \]

In fact, when you see \( a[i] \) in C code, what C really does before anything else is to transform it to \( *(a+i) \). The bracket notation (i.e., \( a[i] \)) is really just a programmer convenience; instead of writing \( *(a+1), *(a+2) \), etc., you can just write \( a[1], a[2] \), and so forth.

You can even treat a pointer \( p \) as if it were an array, and use the bracket operator to access it, i.e., \( p[2]=100; \) is appropriate and reasonable C code. However, you cannot treat the array \( a \) as if it were a pointer. Specifically, you cannot assign a statically-defined array (such as \( a \) in all the examples above) to a new value. If you try to, C will yell at you, saying something like this:

```
error: array type 'int []' is not assignable
```

More formally, we would say that \( a \), in this case, is not a proper lvalue (i.e., something that can be put on the left side of an assignment statement). And thus, we realize our second (corollary) law:

The Law of Pointer/Array Non-Equivalence

Arrays are not identical to pointers. One big difference: you cannot change their value (they are like “constant” pointers).

Although there are many places where pointers and arrays can be used interchangeably, they are not identical constructs.

18.6 Easy Initialization With Compound Literals

In more recent forms of C, there are some easy ways provided to both define a new array and initialize it to a particular set of values. This simplicity is achieved with something called a compound literal, which is just a fancy way of saying “a grouping of a bunch of numbers in your code”.

For example, let’s say we want to define an array with five elements in it, and initialize the elements to values 1, 2, 3, 4, and 5:

```c
int i, b[5];
for (i = 0; i < 5; i++)
    b[i] = i + 1;
```

---

1Even weirder: because C does this so literally, you can even write the nonsensical \( i[a] \) in your C code, which translates to \( *(i+a) \), which, as you can perhaps guess, is exactly the same as \( *(a+i) \) [vdL94]. However, don’t do this in your code, as it is weird.

2Later, we’ll be able to do this with other types, but just numbers for now.
However, C now lets you define and initialize an array with one simple line:

```c
int b[] = {1, 2, 3, 4, 5};
```

In this example, C is smart enough to know exactly what you mean: define an array of size five, set the 0th element to value 1, the 1st element to 2, etc. And thus, in some cases, compound literals can simplify your code.

### 18.7 Passing Arrays To Functions

As you can with any data type, you can pass an array to a function. So what exactly gets passed? To answer this question, you just need to think back to the law of pointer/array equivalence – that in most cases, arrays and pointers behave quite similarly. And they do here, too; passing an array is just like passing a pointer to the first element of the array.

Syntactically, you can define a function that takes an array as input in one of two ways. The first is with square brackets. Here is an example of a function that takes in an array and increments each of its elements by one:

```c
void array_increment(int v[], int size) {
    int i;
    for (i = 0; i < size; i++)
        v[i] += 1;
}

int main() {
    int a[4] = {0, 0, 0, 0};
    array_increment(a, 4);
}
```

Now, if you’ve been paying a little attention, you might be worried that this function doesn’t work; C, after all, passes arguments by value and thus isn’t a copy of the array pushed onto the stack, and modified uselessly?

The good news here: the function works exactly as you want. When you pass an array to a function (as is done in `main()` above), all you are doing is passing the address of the beginning of the array. As a result, when you access the array inside the function, you are accessing it via dereference, and thus you are accessing the original array (not a copy). The only thing that is copied during the function call is the address of the array.

Recall further that writing `v[i]` is equivalent to writing `*(v+i)`. This rewrite should make it even clearer that the access done within the function will update the array in place.
Figure 18.4: Passing An Array To A Function

Figure 18.4 shows the state of the stack before, during, and after the call to `array_increment()`. As you can see, regardless of the size of the array, passing an array to a function is highly efficient—all C does is push the address of where the array begins.

One final note: if you want a function to instead operate on a copy of the array, you’d have to make a copy yourself and that pass the copy to the function. We’ll leave that as an exercise to the reader.

---

### 18.8 Arrays: Now In 2D And 3D!

So far we’ve only examined single-dimensional arrays. C supports multiple dimensions just as readily. For example, to define and initialize a two-dimensional array, you just need to do this:

```c
int v[3][4];
int i, j;
for (i = 0; i < 3; i++)
    for (j = 0; j < 4; j++)
        v[i][j] = 0;
```

We could once again use a compound literal instead of this cumbersome nested loop, as you might have guessed, or perhaps even hoped for:

```c
int v[3][4] = { {0,0,0,0}, {0,0,0,0}, {0,0,0,0} };
```

We can think of an array like this in matrix form, with rows and columns. The first row consists of all the `v[0][0...3]` elements, the second with the `v[1][0...3]` elements, and the third with the `v[2][0...3]` elements:
Arrays

With multi-dimensional arrays, one thing that is always of interest is the exact layout in memory. C lays arrays in row-major order, so the rows above are contiguous in memory, with row 0 \( (v[0][0], v[0][1], v[0][2], v[0][3]) \) before row 1 before row 2. Figure 18.5 shows a depiction of memory containing this array, starting at address 1000.

This layout has implications for C, which must be able to properly compute the address of \( a[row][col] \) given the address of the beginning of the array in memory. To compute the address, we can use the following formula:

\[
\text{address}(row, col) = \text{start} + (((\text{ncols} \cdot \text{row}) + \text{col}) \cdot \text{element size});
\]

In this formula, \( \text{start} \) is the starting address of the two-dimensional array, \( \text{ncols} \) is the number of columns in the array (in this example, 4), and \( \text{element size} \) is the size of each element in the array (in this case, an integer, and thus size is 4).

Let’s see if this address calculation works. Assume we wish to access \( a[2][1] \). Assuming the layout shown in Figure 18.5, the starting address of this array is 1000, and the number of columns is 4. Thus, the address of \( a[2][1] \) is 1000 + (((4 · 2) + 1) · 4) which equals 1036. As you can see in the figure, this result matches our expectation.

We can generalize this discussion to higher dimensional arrays. However, we leave the details to our dearest of friends, you, the reader, once more. Thanks for your help!

<table>
<thead>
<tr>
<th>Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0][0]</td>
</tr>
<tr>
<td>a[0][1]</td>
</tr>
<tr>
<td>a[0][2]</td>
</tr>
<tr>
<td>a[0][3]</td>
</tr>
<tr>
<td>a[1][0]</td>
</tr>
<tr>
<td>a[1][1]</td>
</tr>
<tr>
<td>a[1][2]</td>
</tr>
<tr>
<td>a[1][3]</td>
</tr>
<tr>
<td>a[2][0]</td>
</tr>
<tr>
<td>a[2][1]</td>
</tr>
<tr>
<td>a[2][2]</td>
</tr>
<tr>
<td>a[2][3]</td>
</tr>
</tbody>
</table>

Figure 18.5: Two-Dimensional Array Layout
18.9 Passing Multidimensional Arrays To Functions

One last comment about multi-dimensional arrays: passing them to functions requires a bit more work than you might expect. The problem that arises is that the function must have enough information to perform the address calculation shown above. In the case of a two-dimensional array, for example, C wants to know how many columns there are.

Thus, if you are writing a function that takes a statically-defined two-dimensional array with (for example) 5 rows and 10 columns as input, you’ll need to write something like this in your code:

```c
void foo(int a[][10]) {
    ...%
}
```

You could also fully describe all the dimensions as follows:

```c
void foo(int a[5][10]) {
    ...%
}
```

Without the knowledge of how many columns are in the array, C does not know how to perform the address calculation described above. If you instead are not specific enough, C will complain, saying something like this when you try to get the program working:

```
array type has incomplete element type ‘int[]’
note: declaration of ‘a’ as multidimensional array must have bounds for all dimensions except the first
```

That’s a pretty handy error message (thanks, gcc!) and tells you exactly what you need to do to make your code work.

18.10 Common Mistakes And Other Issues

Before closing, let’s discuss a few limitations and common problems with C arrays. The first, and most important: C provides no boundary checks on arrays; thus, you must be cautious when using an array so as not to run off the end of it and access memory in a way you don’t intend.

For example, let’s look at the following buggy code. Can you spot the bug?

```c
int a[4];
int i;
for (i = 0; i < 10; i++)
    a[i] = 0;
```
Hopefully, the bug in the code is obvious: the loop initializes \( a[0] \) through \( a[9] \) while the code above declares the size of the array to be 4, not 10. C will not complain if you do this, but the resulting behavior when the code is run is undefined – we cannot know for certain what will happen when this code executes. Even more strangely, the program might even seem to work! Or, it might crash (halt when you access one of those out-of-bound array elements such as \( a[4] \) through \( a[9] \). Or something else weird could happen [R10].

The real point here: C leaves it up to you not to access array elements that don’t exist. If you do so, your program will work as expected. If you don’t, you are playing with fire and bad things will likely happen. So be careful!

Another issue is that C provides no way to resize statically-defined arrays. To add more flexibility to arrays, we’re going to need some more machinery, specifically a way to dynamically allocate (and free) memory ourselves as need be. We’ll see how to do this later on; for now, we’ll just have to be content with static arrays.

One last confusion appears when examining a function parameter signature, as follows:

```c
void foo(int* a) {
    ...
}
```

Does the function \( \text{foo}() \) expect a pointer to a single integer, or a pointer to an array of integers of some kind? Unfortunately, there is no easy way to tell in C. If it’s an array, it’s true that the programmer could have written this equivalent form instead, and thus hint at the need to pass an array into \( \text{foo}() \):

```c
void foo(int a[]) {
    ...
}
```

It’s even worse with characters, as a `char*` could mean a pointer to a single character, a pointer to an array of characters, or a proper string (an array that is terminated by a `\0`). Thus, when you see a pointer type defined anywhere, make sure to know how a pointer is being used: as a pointer to one of something or an array of somethings.

18.11 Summary

We’ve seen the introduction of the static C array, in both single and multiple dimension forms. We’ve also learned that in many ways, C arrays behave like pointers, and that pointers can be treated
like they are arrays. Finally, we’ve learned that C arrays can be quite dangerous, as no bounds checking is performed by the language itself. As always in C, the programmer must beware; with a little carelessness, it is easy to write a program that looks mostly fine but performs the most awful of behaviors.
References

[C+12] “Programming Perl”
Tom Christiansen, brian d foy, Larry Wall, Jon Orwant
O’Reilly, February 2012, 4th Edition
Oh Perl isn’t that bad of a language. And honestly, you can’t really be sloppy in any language. But if you want to see some of the most unreadable code in the world, just look for Perl code examples, and wash your eyes when you are done.

John Regehr
http://blog.regehr.org/archives/213
An excellent introduction to undefined behavior. We’ll come back to this later, but if you’re interested, take a look and start learning now.

Peter van der Linden
Prentice Hall, June 1994
A terrific book about how C actually works. Unfortunately, slowly going out of date, but still a terrific resource for those interested.