Learning to Program with Haiku

Lesson 9

Written by DarkWyrm

If you haven't noticed, we've come a long way since Lesson 1. We're steadily getting closer to being done with learning the basics of C and C++. This time around, we'll expand what we know about arrays and pointers.

Arrays: Pointers with Smoke and Mirrors

When we first looked at strings in Lesson 5, we saw how arrays and pointers are pretty closely related, but we didn't look close enough to discover how close they really are. Arrays are really just a friendly way of using pointers. Let's look at two different ways of changing the same string that demonstrate this.

```
#include <stdio.h>

int
main(void)
{
	char string[30];

	// Set the array to the lowercase alphabet using the way we already
	// know: brackets

	for (int i = 0; i < 26; i++)
	{
		// Character constants can be treated like integers. It saves us from
		// having to look up to see what integer value the lowercase letter
		// A has.
		string[i] = 'a' + i;
	}
	string[26] = 0;
	printf("%s\n",string);

	// Here's another way: using pointers and some math
	for (int i = 0; i < 26; i++)
	{
		char *index = string + (i * sizeof(char));

		// Use a capital A just for something different
		*index = 'A' + i;
	}
	string[26] = 0;
	printf("%s\n",string);
}
```

Both loops do the exact same thing except for using capital letters in the second one. The code for the second one looks really strange, though, so let's pick it apart, starting with `sizeof()`. `sizeof()` is a language feature which works like a function. It can be given a variable or the name of a type and it will return the size, in bytes, that one object of that type takes up in memory. For example, `char` objects are only 1 byte large, but an `int` takes of 4 bytes. Give `sizeof()` an array and it will return the amount of memory the entire array occupies. Because the size of a `char` is 1, we could have skipped the multiplication entirely, but it's a good idea to get into the habit of avoiding assumptions like this. You'll have a fewer gray hairs that way.

The next strange trick we see in the code above is adding a number to a pointer. Pointers are just variables that contain memory addresses, so adding an integer to a pointer just changes the memory

address. We can't change the array's address, so we've created a pointer that we can change. This allows us to add the size of an array element to it. As we do this over the course of the loop, the pointer will hold the address of each character in the array.

Array Index	Equivalent Pointer Math	Value
`string[0]`	`index`	`a`
`string[1]`	`index + (size * 1)`	`b`
`string[2]`	`index + (size * 2)`	`c`
`string[3]`	`index + (size * 3)`	`d`

We can use this close link between arrays and pointers to pull off other fancy tricks, like initializing an entire `bool` array to true. How? `memset()`, which we haven't used in quite some time.

```
int
MyFunc(void)
{
	bool bArray[100];
	memset(bArray, 1, sizeof(bool) * 100);
	return 0;
}
```

This little function just sets each byte in the array to a 1, which is true for Boolean logic. Not only is it less work, it is much, much faster than using a loop to set each element individually.

Multidimensional Arrays

It's really inconvenient that we can make arrays of any kind of data except strings. Strings are everywhere in everyday programming, but we can't make an array of arrays, can we? Actually, yes we can. To have arrays of strings, we must conquer one of the most potentially confusing parts of C and C++: multidimensional arrays. Not to worry, I'm going to make this as simple as possible.

Regular arrays are best thought of as just a series of elements in a row:

```
int myArray[10];
```

myArray:	0	1	2	3	4	5	6	7	8	9

Arrays can have more than one dimension – for example, a two dimensional array can be thought of as a grid – a group of rows. This one has two rows with five elements in each row.

```
int my2DArray[2][5];
```

my2DArray:	0	1	2	3	4
	5	6	7	8	9

The easiest way to think of an array with more than one dimension is to think in terms of spatial dimensions, working from right to left as we add another pair of brackets of each dimension. One dimension is a line, two is a rectangle, and three is a cube. Beyond three, it's better to think of the dimensions as groups of cubes or groups of cube groups. Declaring and accessing an array with more

than one dimension is just a matter of adding another set of brackets when declaring and accessing the array.

Number of Dimensions	Array Declaration
1	`my1DArray [elementCount];`
2	`my2DArray [numRows] [itemsInRow];`
3	`my3DArray [numGrids] [rowsInGrid] [itemsInRow];`
4	`my4DArray [numCubes] [numGrids] [numRows] [itemsInRow];`
5	`my5DArray [numCubeGroups] [cubesInGroup] [gridsInCube] [rowsInGrid] [elementsInRow];`

Now that we have a handle on how multidimensional arrays are declared, let's put them to use in some sample code.

```
#include <stdio.h>
#include <string.h>

int
main(void)
{
	// Declare and initialize an array with 4 rows of 5 items each – a
	// grid 5 elements wide and 4 elements high
	int integerArray[4][5];

	int value = 0;
	for (int y = 0; y < 4; y++)
	{
		for (int x = 0; x < 5; x++)
			integerArray[y][x] = value++;
	}
	return 0;
}
```

This code snippet declares an array and initializes it with a loop. Because the memory for the entire array is allocated in one large block, we can use `memset()` in combination with `sizeof()` to set every byte in the array to the same value. This also means that we can use a pointer to the same address as our two dimensional array to treat the whole thing as one long list.

```
int
main(void)
{
	// Declare and initialize a grid of integers which has 4 rows of 10
	// integers each.
	int intArray[4][10];

	for (int y = 0; y < 4; y++)
	{
		for (int x = 0; x < 10; x++)
			intArray[y][x] = (y * 10) + x;
	}

	// Even though it was declared as a grid, sometimes it's just a lot easier
```

```
	// to think in terms of one long list of 40 elements. This is just a
	// different way of looking at the same set of data. Because this is a
	// two-dimensional array, intArray by itself is of type int ** -- a
	// pointer to an integer pointer. Adding an asterisk makes it just an int *.
	int *pInt = *intArray;
	for (int i = 0; i < 40; i++)
		printf("%d\n",pInt[i]);
}
```

Even though we have initialized all of our arrays using `memset()` or a loop, it's possible – and sometimes necessary – to use many different arbitrary values. This is done using a comma-separated list of values inside a pair of curly braces. A pair of braces is needed for each dimension. Here is the same basic code changed to initialize the array without using a loop.

```
int
main(void)
{
	// Declare and initialize 4 integer arrays which have 10 elements each.
	int intArray[4][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
				{ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19},
				{ 20, 21, 22, 23, 24, 25, 26, 27, 28, 29},
				{ 30, 31, 32, 33, 34, 35, 36, 37, 38, 39} };

	int *pInt = *intArray;
	for (int i = 0; i < 40; i++)
		printf("%d\n",pInt[i]);
}
```

It's a lot more typing, but if we had a list of different values that didn't follow any pattern at all, this would be our only option. One drawback to this method is that we have to set the values for all of the elements – we can't just pick and choose which ones we want to set. It also gets increasingly difficult to read for arrays having more than two dimensions. If this were just a plain array, we'd just need one set of curly braces and a list of values in it, like this:

```
float someArray[3] = { 1.1, 2.2, 3.3 };
```

This is also one of the few instances where you have to have a semicolon outside a pair of curly braces.

To create a list of strings, we simply create a two dimensional `char` array. Although we could use a series of comma-separated character constants, C and C++ give us a couple of kinds of shortcuts to save a lot of typing and possibly some counting when initializing strings.

```
// This is the hard way. What a mess!
char myShortString[15] = { 'a', 'b', 'c', 'd', 'e', '\0' };

// Declare an array that can hold up to a 15 character string, including
// NULL terminator. This is MUCH better than using character constants and braces.
char myFastString[15] = "abcde";

// Leaving out the size tells the compiler to allocate enough memory to hold
// the string. This saves calling strlen() or counting it ourselves.
char myLongString[] = "This is some really long string I don't have to count.";

// We can leave out the size on one-dimensional arrays of other types if we
```

```
// initialize them.
int myIntArray[] = { 0, 1, 2, 3, 4, 5 };

// This has the same result as the declaration for myLongString. It is also
// the more common way of doing it.
char *anotherLongString = "This is some other really long string.";
```

One other useful tip: **don't** try leaving out the size on arrays with more than one dimension. It can only be done with the leftmost dimension listed when declaring a multidimensional array and mixing the two can only lead to confusion. Trust me on this one.

Once again, we've covered a lot in a short amount of time, so let's quickly review:

- Integers can be added to or subtracted from the address held by a pointer.
- `sizeof()` returns the size of a type, variable, or array, measured in bytes.
- You can use pointer math to get the value of an element in an array.
- The memory allocated for an array is contiguous.
- Arrays can be declared to have more than one dimension and accessed that way or reinterpreted as one long list of elements using a pointer.
- `char` arrays (strings) can be initialized using a regular string.
- Non-string arrays are initialized using curly braces.
- All elements in an array must be given a value when it is initialized.
- You can leave out the size in the brackets on one-dimensional initialized arrays if you just want enough memory reserved to hold the values given to it.

Answers from Lesson 8's Bug Hunt

1. The problem is that the parameter string is a `const char *` instead of a just `char *`. This means that the string given to `ReverseString()` can't be changed and the compiler complains when we try to change it in the `for` loop. Remove the `const` keyword and everything is happy.