Learning to Program with Haiku

Lesson 5

Written by DarkWyrm

Computer languages are a funny lot in that some of them are closer to human language and others are closer to machine code. Assembly is one step removed from machine code, yet COBOL is about as close to human language as it gets. C++ is somewhere in the middle, as you will get a hint of in this lesson.

Arrays

In addition to single variables, C++ also gives us the ability to use collections of variables called **arrays**. The individual members of these groups, called **elements**, are all the same type and are clustered together in the computer's memory. You could say that arrays are a little like egg cartons – containers of a bunch of little items of the same kind. Arrays are declared just like regular variables except for a pair of square brackets with a number inside that states how many items the array contains.

```
int thisIsAnArray[5];
```

The above example creates an array containing five integers that take up a contiguous block of computer memory. If you were able to look at a section of memory, the part of the computer's memory which holds our array would look a little like this:

[integer0] [integer1] [integer2] [integer3] [integer4]

Using arrays can take a little getting used to, however. Each element in an array is assigned a number. Accessing an element in an array is done by using this number inside a pair of square brackets.

```
int main(void)
{
	// Declare the array itself, which contains 5 elements.
	int intArray[5];

	// The first element in an array has an index of 0.
	// More on this in a moment.
	intArray[0] = 10;

	// The second element of the array
	intArray[1] = 11;

	// The rest of the elements in the array
	intArray[2] = 12;
	intArray[3] = 13;
	intArray[4] = 14;

	return 0;
}
```

In this example, we took our 5-element array and set the value of each element to a different number. Notice that the first element in the array has an index – that is, a reference number – of 0. While people start counting items with the number one, computers start counting at zero, so even though the array contains 5 elements, they're numbered 0 though 4. It's this kind of weirdness that programmers deal with that make regular people think programmers are brain damaged or something. Don't worry – it'll get worse.

Arrays require some special care and feeding. The memory used to hold them is set aside in chunks and arrays declared this way have a fixed size. The size of the array doesn't magically increase if we try to set `intArray[6]` to 16. The results are unpredictable, but most often the operating system aborts our program. Using a negative index will also have the same result. This particular kind of error is called a **segmentation fault**, or segfault for short. Anyone who remembers the days of Windows 95 probably saw the occasional "General Protection Fault" that commonly appeared. A GPF was just another name for a segfault.

Pointers

No discussion of arrays would be complete without also talking about pointers. Pointers are a fundamental concept of C and C++ which can initially sound even crazier than counting from 0. A **pointer** is a variable which contains (or *points to*) a memory address. You can always recognize the declaration of a pointer because its name is preceded by an asterisk (*).

```
char *somePointer;
```

Let's look at a code example which demonstrates how to use pointers.

```
#include <stdio.h>

int main(void)
{
	// This variable will be our "guinea pig."
	int myInt = 5;

	// Declare a pointer that we know doesn't point to a valid memory address.
	int *uselessPointer = NULL;

	// Set the value of this pointer to the memory address that myInt stores
	// its value in. Without the * in front of its name, it would be considered
	// a regular variable and this would generate a compiler error. The
	// ampersand (&) in front of myInt gets the address of myInt.

	// Note that space in between the * and the name of the pointer is OK.
	int *intPointer = &myInt;

	// %p prints the address of a pointer. This changes from one execution of
	// the program to another.

	// A pointer with a * in front of it returns the data its address
	// actually holds, so *intPointer in this case is 5.
	printf("intPointer's address is %p and contains the value %d\n",
		intPointer, *intPointer);
}
```

Pointers, like arrays, require care when used because they can make it really easy for a programmer to segfault a program. **Always initialize a pointer to NULL or a known-good address**. What's NULL, you ask? It's just another word for zero when referring to pointers. Even though a NULL pointer is just as unusable as an uninitialized pointer – one which points to a random address – you know for certain that it's unusable.

Strings

In the Lesson 3, we learned about the different kinds of information that can be stored in variables – types – but we left out an important one: strings. We have been using them in `printf()` statements – everything in between a pair of double quotes is a string.

Strings in C and C++ are very different from other data types. A string is an array of `char` values whose last character is a 0. `char` variables can be initialized with either an integer from 0 to 255 or with a character constant – a character enclosed by single quotes – 'a' or 'b'.

In addition to regular letters, there are also some special characters. These special characters all start with a backslash and for the purposes of memory requirements take up one byte even thought they take more than one character to type them. Note that these only work with a backslash – a slash which "leans" to the left – and not a forward slash.

Character	Character Code
Backspace	`\b`
Carriage Return	`\r`
Form feed	`\f`
NULL (string terminator)	`\0`
Newline	`\n`
Tab	`\t`
Backslash	`\\`
Single quote (')	`\'`
Double quote (")	`\"`

The first four are not commonly used in Haiku or UNIX/Linux programming. The carriage return (`\r`) is used instead the newline character (`\n`) to start a new line on Macintosh computers. Windows operating systems use both in combination for the same task – `\r\n`. Knowing this is handy when working with text files coming from other operating systems.

Let's look at an example that uses just single characters.

```
#include <stdio.h>

int main(void)
{
	// This loop prints the alphabet in capitals
	for (char i = 65; i < 91; i++)
		printf("%c",i);

	char endline = '\n';
	printf("%c",endline);
}
```

There are tons of different ways to work with strings, so let's look at just a few for the moment. The fastest way to understand it all is with some code. Work slowly through this heavily-commented example to get a good handle on it all.

```c
#include <stdio.h>

// A new include! This one has a bunch of functions just for working with strings
#include <string.h>

int main(void)
{
	// Declare a string, aka an array of the char type
	char string[30];

	// Fill the string with 0's. While it might not seem intuitive to include
	// a "memory" function in string.h, it's often used for purposes like this.

	// memset: sets the value of all bytes in a block of memory to a
	// particular value
	// usage: memset(anArray, valueToAssign, sizeOfTheArray);

	// This call sets everything in our array to 0
	memset(string,0,30);

	// Another way to set values of characters in a string: as an array. Here we
	// Individually set the characters. A capital letter A has an integer value
	// of 65.
	for (char i = 0; i < 26; i++)
		string[i] = 65 + i;

	printf("String contains: %s\n",string);

	// Yet *another* way to set a string's value. sprintf() -- think
	// "string printf" -- prints to a string instead of the screen, but
	// otherwise works the same as printf(). Just be careful that what is
	// printed isn't larger than the string that you're printing to. If it is,
	// your program will happily crash into bits.

	// usage: sprintf(aStringVariable,formatString, argumentList)

	sprintf(string,"%f",3.1415927);

	// %s is the placeholder for a string in printf();
	printf("String changed. Now it contains: %s\n",string);

	return 0;
}
```

The reason for using `memset()` in this example needs a little more explanation. Strings, as previously mentioned, are `char` arrays that are expected to end in a 0 for almost all uses. When we called `memset()`, we set the entire array to zero so that when the first 26 elements is set to capital letters of the alphabet, the 27th element is the terminating null character (0). `sprintf()` automatically places a null terminator at the end. Without this terminator, we end up printing some garbage characters after our string.

Whew! That was a lot of stuff about arrays, pointers, and strings. Lets quickly recap:

- Arrays are declared with a fixed size using square brackets
- Array elements start counting up from 0
- Accessing memory outside the bounds of an array's allocated memory block will cause a segmentation fault (crash).
- An array can be used like a pointer by using the name of the array without the brackets or an index.

- Pointers are variables which hold memory addresses
- Pointers are declared with an asterisk in front: `int *myPointer`
- Pointers should always be initialized either to `NULL` (zero) or a known-good address
- The address of a variable can be obtained with an ampersand: `&myVariable`

- Character constants are enclosed in single quotes: 'a' or 'X'
- Char variables can be initialized with a character constant or a number from 0 to 255.
- There are special character constants that take more than one character to type, but are treated as one character, such as `\n`, which starts a new line.
- Strings are enclosed in double quotes: "This is a string"
- Strings are arrays of type `char` whose final character is a `NULL` (0) end-marker

Project

One advantage programmers have over other trades is being able to make tools to help them in their work. Let's make a program which asks for a word from the user and it prints out the integer value of each character.

In order to get information from the user, we'll need to use two new functions: `gets()` and `strlen()`. Both take a `char` pointer as the only parameter. We'll use a `char` array for both – remember that arrays can be used like pointers if you leave out the brackets and index number. Here are what the declarations of these two functions look like and a description of each:

```
char * gets(char *inString);
```

> `gets()` gets a string from the user. The user may type as much as he wants and presses the Enter key when finished. The final `\n` character the user types is replaced with a 0 to mark the string's end. `inString` is a `char` array which is to hold the user input. When the user finishes typing, `gets()` copies the user input into `inString` and returns it also. This doesn't sound like it makes much sense, but don't worry about it.

```
int   strlen(char *inString);
```

> `strlen()` calculates the length of the `NULL`-terminated string given to it. A word of warning: passing a `NULL` string to it will cause your program to crash.

Here are the basic steps for how we will write our program:

1. Make a `char` array to hold the information from the user
2. Call `gets()` to get the information from the user and store it into our array.
3. Make an `int` variable and set it to the string's length
4. Use a `for` loop to print each character in the string both as a character and its numerical value

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

int main(void)
{
	char inString[1024];

	printf("Type the text to convert and press Enter: ");
	gets(inString);

	// Here's where you come in.
	// Use the steps above to figure out what goes here.
	// Steps 1 and 2 have already been done for you.

	return 0;
}
```

Bonus: Make your project print the character code as hexadecimal (base 16) numbers and/or octal (base 8) in addition to regular (base 10) numbers. See Lesson 3 for more information.

Hints: Closely look over the code examples in the earlier section on strings for hints on how to do step 4. Also, have a second look at the list of placeholders used in `printf()` from Lesson 3.

Going Further

Whenever the compiler builds this project, it complains that `gets()` is dangerous and shouldn't be used. Why do you think this might be?

Bug Hunt

Hunt #1

Code

```
#include <stdio.h>

int main(void)
{
	int number = 0;
```

```
	for (int i = 1; i < 10; i)
	{
		number += i;
		printf("At step %d, the number is now %d\n",i,number);
	}
}
```

Errors

The code builds just fine, but it won't stop running printing stuff on the screen and the only way to stop it is either press Ctrl+C or close the Terminal window.

Hunt #2

Code

```
#include <stdio.h>

int main(void)
{
	int a;
	int b, c;

	a = 1;
	b = 2;
	c = 3;

	printf("a is %d, b is %d, and c is %d.\n",a,b);

	return a + b + c;
}
```

Errors

```
foo.cpp: In function ‘int main()’:
foo.cpp:12: warning: too few arguments for format
```

Lesson 4 Bug Hunt Answers

1. The `i++` in the for loop needs to be `i += 2`
2. The warnings come from using the `%d` – used for integers – for a `float` variable. Change the `%d` placeholders in the `printf` statements to `%f` and they'll go away.