# Learning to Program with Haiku

Lesson 6

Written by DarkWyrm



This time around we are mostly going to expand on things we already know. Back in Lesson 4, we learned about executing pieces of code only if certain conditions are met with `if` statements and we also saw how to repeat a set of instructions with `for` loops. Let's add a few more tools to our kit.

## *More Loops*

Although they are probably the most common, there are more kinds of loops than just `for` loops. In fact, there are two other closely-related loops: the `while` loop, and the `do-while` loop.

`while` loops are simpler, having just a loop condition that causes the code in the loop to be repeated until the loop condition is false. Let's take a look at a way to do our project from Lesson 5 using a `while` loop.

```
#include <stdio.h>

int main(void)
{
	char inString[1024];
	printf("Type some text and hit Enter:\n");
	gets(inString);

	int i = 0;
	while (inString[i])
		printf("String[%d]: %c\n",i,inString[i++]);

	return 0;
}
```

In the above example, we even take a shortcut so that curly braces aren't needed – a postincrement operator increases the index variable `i` after the message is printed. The condition for the loop checks to see if `inString[i]` is a valid character. Integers can be used for boolean logic, that is, true/false tests. Zero is treated as false and everything else is true, so this loop will repeat until the index points to the 0 that terminates the string. Pretty slick, eh?

Aside from the `while` statement and the accompanying condition block, the rest of the loop works just like a `for` loop, so we could just as easily substitute our `printf()` statement with a group of instructions inside a pair of curly braces. One caveat to using this kind of loop: it's pretty easy to end up in an endless loop if you are not careful to ensure that the loop condition is eventually false. In this instance, we would have an endless loop on our hands if the `inString[i++]` were just `inString[i]`.

The `do-while` loop is not used as often as others, but it does come in handy on occasion. It executes the loop instructions once and then checks afterward to see if they need to be done again. Here is the same example using a `do-while` loop.

```
#include <stdio.h>

int main(void)
{
	char inString[1024];
	printf("Type some text and hit Enter:\n");
	gets(inString);

	int i = 0;
	do
	{
		printf("String[%d]: %c\n",i,inString[i]);
	} while (inString[i++]);

	return 0;
}
```

We print a character, check to see if it's zero – which all proper strings end with – and print another character if it's not a zero. Note that a semicolon is required after the `while` condition in this kind of loop unlike the `for` and regular `while` loop. Luckily, if you should happen to forget it, the compiler will complain about it.

```
foo.cpp: In function ‘int main()’:
foo.cpp:13: error: expected `;' before ‘}’ token
```

There's a problem with using this loop for this job: if the user just hits Enter without typing anything in, it prints a blank character. If the user didn't effectively enter anything, we should skip printing it entirely. This particular task is better suited to using a `for` loop because it requires less work, but we can make using this one work by adding one more bit of code.

```
do
{
	// This will prevent problems caused by the user not typing anything
	if (!inString[i])
		continue;

	printf("String[%d]: %c\n",i,inString[i]);
} while (inString[i++]);
```

The `if` condition here checks to see if the current character is zero – by using a `!`, which evaluates as a boolean NOT. The zero character, which is normally considered false, will make the `if` statement condition true and the program will execute the `continue` statement. `continue` causes program execution to skip directly to the loop's condition block. The zero character that caused the jump to the loop condition will also cause the condition to be false and the loop will exit.

Using `continue` this way, though, is not something you'd normally see. Cases like this normally would call for a `break` statement, which jumps immediately out of the current code block. `break` would skip checking the loop condition and just exit the loop. We'll use `break` statements almost all of the time with our next concept: the `switch` block.

## *switch blocks*

Sometimes we have to deal with making a decision based on one of several possible values for a variable. This could be handled with a series of `if-else` statements, but C and C++ give us `switch` statements to do the job much more elegantly. Let's expand our string-printing project to provide a way to print characters that don't show anything when printed, like spaces and tabs.

```
#include <stdio.h>

int main(void)
{
	char inString[1024];
	printf("Type some text and hit Enter:\n");
	gets(inString);

	int i = 0;
	while (inString[i])
	{
		switch (inString[i])
		{
			case '\n':
			{
				printf("String[%d]: <carriage return>\n",i);
				break;
			}
			case '\t':
			{
				printf("String[%d]: <tab>\n",i);
				break;
			}
			case ' ':
			{
				printf("String[%d]: <space>\n",i);
				break;
			}
			default:
			{
				printf("String[%d]: %c\n",i,inString[i]);
				break;
			}
		}

		i++;
	}

	return 0;
}
```

The value we are evaluating is placed in the parentheses for the `switch` statement. Each value that we want to handle is taken care of by `case` blocks. These `case` blocks define the set of instructions to execute when the value we are evaluating matches the value specified for the `case`. The format for `case` blocks is the following:

```
case valueForCase:
{
      block of instructions
}
```

`break` statements are very important here because they jump out of the `switch` statement after handling the particular `case`. Without one at the end of a `case`, execution would "fall through" and continue into the next case. This is almost always not what we want. Using our above example, leaving out the `break` at the end of the `case` for spaces would cause it to be printed twice: once for the space `case` and once for the `default` case, looking something like this:

```
String[5]: <space>
String[5]:
```

The `default` case is a catch-all for all values which don't fit one of the specified cases. It must be placed at the end of the cases in a `switch` statement. Any cases placed after the `default` case will not be executed. It seems silly, but it's true.

## *Conditional assignment*

We've seen some of the ways that C and C++ offer the programmer shortcuts to do certain common tasks, such as adding 1 to a variable. Another such shortcut is the only operator which has three sections. Here's one way to assign a value to a variable based on a condition.

```
int number;

if (someOtherNumber > 5)
      number = 1;
else
      number = 10;
```

Here's another, much shorter, way to do it.

```
int number = (someOtherNumber > 5) ? 1 : 10;
```

Right now, you might be thinking, *"Hold on there, cowboy! What's this here nonsense?"* and needing to lay off the *Bonanza* marathons. Then again, maybe not. The conditional operator has two parts, the question mark and the colon. The format for how it works is as follows:

```
condition ? valueIfConditionIsTrue : valueIfConditionIsFalse
```

Parentheses aren't required around the condition, but some people (like me) prefer to have them even if unnecessary to set the condition apart from the rest. If the condition is true, then the value between the question mark and the colon is returned, otherwise the value after the colon is returned. Its use can be limited, but it does come in handy now and then.

## *Bug Hunt*

### *Code*

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

char *ReverseString(char *string)
{
	// This function rearranges a string so that it is backwards
	// i.e. abcdef -> fedcba

	if (!string)
		return NULL;

	int length = strlen(string);
	int count = length / 2;

	for (int i = 0; i < count; i++)
	{
		char temp = string[length - i];
		string[length - i] = string[i];
		string[i] = temp;
	}

	return string;
}


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

	printf("Type a string to reverse:");
	gets(inString);

	printf("The reversed string is %s\n",ReverseString(inString));

	return 0;
}
```

### *Errors*

The program compiles just fine, but nothing is printed.

### *Help*

Normally no help is given for Bug Hunt sections, but this is harder than many. The bug itself is somewhere in `ReverseString()`. Try using `printf()` calls to print values in certain places to give yourself more information on what the program is doing, like printing the length, the count, etc.

## *Lesson 5 Bug Hunt Answers*

1. The last section of the `for` loop has just an `i`. It should be `i++`.
2. The variable `c` is missing from the end of the argument list in the `printf` statement.

## *Lesson 5 Project Review*

In the last lesson we were trying to make a program which asks for a word from the user and it prints out the integer value of each character.

We were given these steps to follow to write it:

1. Make a `char` array to hold the user input.
2. Call `gets()` to get the string from the user.
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 was the part we needed to fill in, so here's one way to do it.

	// Step 3.
	int strLength = strlen(inString);

	// Just some fluff to make this program a little nicer
	printf("The character codes for the string \'%s\'\n",inString);

	// Step 4.
	for (int i = 0; i < strLength; i++)
		printf("[%d]: %c\t%d (0x%x)\n",i,inString[i],inString[i],inString[i]);

	return 0;
}
```

This example even includes printing the character code in both decimal and hexadecimal.

## *Unit 1 Review Answers*

Lesson 1

1. Machine code is the computer's native language – a set of numerical instructions which the computer executes.
2. Source code is the human-readable text used to create a program.

Lesson 2

3. One line comments can be added using two forward slashes – `//` – and comments which can span multiple lines are placed between these markers: `/*  */`
4. The four tools used to turn source code into machine code are the preprocessor, the compiler, the assembler, and the linker.
5. The two classes of programming errors are syntactic and semantic. Syntactic errors are mistakes in the construction of the computer language, such as having the wrong number of parentheses in a statement. Semantic errors are mistakes in the program logic of code that has correct syntax – bugs in the meaning of the code.

Lesson 3

6. The `%` operator returns the remainder of the division of the two arguments. `10 % 4` is 2.
7. `++i` adds one before the rest of the expression is evaluated. `i++` does the addition afterward.
8. myVar *= 5
9. `==` is used for comparison because `=` is used for assigning a value to a variable.
10. `%f` is the placeholder for a float when using `printf`.
11. Parameters are information given to a function which it requires to perform its task.
12. The difference between int and long is that long typically can hold bigger values.
13. `10 / 4` is integer division, so the result is 2. `10.0 / 4.0` is floating point division, so the result is 2.5.

Lesson 4

14. Nothing will be printed. `i` has a value of zero, which, for the purposes of logic, is treated as false.
15. The boolean OR operator, `||`, should be used here.
16. The first section of the control block of a `for` loop sets the value of the index variable. The second section is the test to see if another iteration is called for, and the third modifies the index variable.

Lesson 5

17. A segmentation fault is an attempt to access memory that doesn't belong to your program.
18. Because array indices start at 0, the last valid index for a 10-element array is 9.
19. Pointers should be initialized to NULL or a known-good address so that you always can tell whether or not a pointer is valid and can, thus, avoid segfaults.
20. `'a'` is a character constant. `"a"` is a string, which amounts to the `'a'` character constant plus a NULL string terminator.
21. A string is an array of the type `char`.
22. `string.h` is the header which we have been using that contains helper functions for working with strings.