Learning to Program with Haiku

Lesson 7

Written by DarkWyrm

Some of the most fun times when learning to program are those instances where a neat idea hits you and, after some playing around, you manage to write a program of your own which does something you want and you think "Holy smoke! Check this out! I just made the computer <*insert cool achievement here*>!" This lesson is probably not going to inspire any of those moments, but, like eating your vegetables, it's good for you. Make sure you understand this lesson well before moving on.

Memory: The Heap and The Stack

You are probably aware that every variable takes up some of the computer's working memory. If weren't aware before, you certainly are now. What you may not know is that each program executed by the operating system has two different pools of memory from which variables are created: the heap and the stack.

The **stack** is a fast pool of memory of limited size. When the operating system loads your code into memory before it is run, a section of main memory is set aside for your program's stack memory. Without getting too deep into the details, let's just say that while it is fast, it is also limited and if your program runs out of stack space, it will crash. Using the stack for lots of large arrays is a Bad Idea^{TM}. All variables we have used so far were allocated from stack memory.

The **heap** is the main part of the computer's system memory. It takes a little longer to get some memory from it, but you don't have to worry about running out unless you decide to allocate, well, *obscene* amounts of it. A drawback to using heap memory is that you have to take care of making sure that all memory you have received from the system is given back when you're done with it. Not doing so will result in a **memory leak**, which occurs when heap memory is not freed after being allocated. This causes your program to gradually take up more and more system memory until it is finally returned to the system when your program quits, gracefully or otherwise.

Up to now I've mentioned C and C++ in tandem. Everything we've learned so far is exactly the same for both languages. While both methods require pointers – we're dealing with memory addresses here – memory management is one instance where they are quite different. We'll look at the C++ way of doing things when we get to the C++-specific concepts much later on.

The C way of allocating memory is using `malloc()` to get a pointer to some memory for us to use and then calling `free()` on the pointer to return that block of memory to the system. It's also possible to use `realloc()` to change the size of a chunk of RAM originally given to us by `malloc()`. Using these functions properly will take some explanation, so let's have a look at their declarations and handle the details one function at a time.

```
void * malloc(size_t size);
void   free(void *pointer);
void * realloc(void *pointer, size_t newSize);
```

Before we even think about what each of these does, it's obvious that there are some types we haven't seen before. The first new type is `size_t`, but it's not really all that new. Instead, it's a new name for one of the integer types that we already know. This kind of trick is often done for **portability** – writing code so that it can be easily written on one operating system and compiled on another. Treat it just like an `int` and everything will be fine.

The other new type we see above is the `void` pointer. `void` pointers don't have a type of their own, which might sound kind of useless when you consider that we can't use the * operator on them to get their value. That's OK, though. Their usefulness comes from being able to pass around pointers of any type, and when we finally need to get their values, we can convert them to another type by typecasting them.

Typecasting, or just casting for short, is telling the compiler to treat one type as another. The values held in memory don't change, but how they are interpreted does. This will become especially useful later on when we start examining some of the features C++ has over C.

`malloc()` obtains a block of heap memory for us, given a specified size. The address of this block is returned to us as a `void` pointer, so normally when `malloc()` is called, its return value is typecast to another pointer's type. If `malloc()` can't get a block of the size requested, it returns `NULL`.

`free()` returns heap memory back to the system. When a function like `free` takes a `void` pointer as a parameter, it simply means that the function will accept a pointer to any data type. No casting is needed to pass, for example, a heap-allocated string to `free()`. Once a pointer is passed to `free()` to return its memory to the system, that pointer it can no longer be used without being reassigned to a valid address. Attempting to use a block of memory that has been freed will have unpredictable results, but normally causes a segfault. If a pointer is intended to be used later on after having its memory freed, it is normally assigned to `NULL` so that it is known to be invalid.

`realloc()` resizes a memory block previously obtained from `malloc()`. It is given the pointer to the block of memory to be resized and the new size desired. If the pointer is `NULL` and the size is greater than 0, then `realloc()` functions just like `malloc()`. If given a valid pointer and a size of 0, it works just like free. Both of these cases are not normally used because calling `malloc()` and `free()` directly is much clearer code. `realloc()` may or may not move the memory to a different address and, if it does, it returns the new address. Existing data in the memory block is preserved, even if the pointer is moved to a different address, but any new memory obtained with this call will contain random data.

Working with these memory functions isn't difficult. Let's have a look at some of them in action with a simple example.

```c
#include <stdio.h>

// Another new include! malloc.h contains definitions for
// malloc and related memory-allocation routines
#include <malloc.h>

int main(void)
{
	// We are just going to print a string in this example, but
	// we are going to use heap memory this time instead of the stack.


	// This pointer will hold the address of the memory block for our string.
	// We don't have any memory for it just yet, so we'll just set it to NULL
	char *string = NULL;

	// Now we will request a block of heap memory from the operating system
	// and give our string pointer its address.

	// The (char *) is what we need to do to cast our block of memory from
```

```
        // having no type (void *) to a char pointer. 50 bytes have been requested,
        // which is enough to hold 49 characters and the NULL terminator
        string = (char *)malloc(50);

        // sprintf is an easy way to convert numbers into strings
        sprintf(string,"The number pi is approx. %f",3.1415927);

        printf("%s\n",string);

        // This will give the 50 bytes of heap memory that we requested earlier
        // back to the system.
        free(string);

        // The address that our pointer holds is NO LONGER VALID. Any attempt
        // to use it will have unpredictable results, but most likely will cause
        // a segmentation fault. We're done in this example, but if we were going
        // to need the pointer again, we'd need to assign it to a valid address

        return 0;
}
```

Beyond Base 10: Binary Math

Now let's take some time to learn some math that we will need in future lessons. Just as computers and people are different in what number they start counting with, the numbering systems that each uses to count are worlds apart.

People use the decimal system, also known as base 10. It is called base 10 because there are 10 possible values in each column, from zero to nine. When you were very young, you may have heard your arithmetic teacher talk about the "ones place," the "tens place," and the "hundreds place." The name for the column is the same as 10 raised to the power of the number of columns moving from right to left, counting from zero.

"Thousands Place"	"Hundreds Place"	"Tens Place"	"Ones Place"
$1000 = 10^3$	$100 = 10^2$	$10 = 10^1$	$1 = 10^0$

For the number 5,234, the total is (1000 x 5) + (100 x 2) + (10 x 3) + 4.

The math involved at the bare metal level of interaction with computers is far removed from anything that we, as people, would have any use for. Computers use binary math – base 2 counting. The only values which exist are 1 and 0. The digit "places" in base 2 math look like this:

"Eights Place"	"Fours Place"	"Twos Place"	"Ones Place"
$8 = 2^3$	$4 = 2^2$	$2 = 2^1$	$1 = 2^0$

Each individual digit in binary math is called a **bit**. It's just a bit of information, and it doesn't do much on its own. Bits are grouped together into bundles of 8, called **bytes**. Whenever we do any binary math when programming, it will involve at least one byte, and often more, but for now we'll just stick to one to keep things as simple as possible for this confusing topic.

Remember that binary numbers are just a different way of writing numbers, like the difference between the number 68 and the Roman Numeral LXVIII.

To convert a number from binary to decimal, you add up the values in each place. Each digit in a byte is simply a power of two:

Bit Number	7	6	5	4	3	2	1	0
Decimal Value	128	64	32	16	8	4	2	1

For each column that has a binary 1 in it, you add the power of 2 that is assigned to that column. For example, binary 10000000 is decimal 128. The only column that has a 1 in it is the first one, which has a decimal value of 128. Binary 10000111 is decimal 135. How? 128 + 4 + 2 + 1.

It's possible to do addition, subtraction, and any other regular mathematical operation in binary, but it's almost never necessary, so we won't cover it here. There are, however, some other operations that are commonly used which are specific to binary math, and C and C++ provide operators for them. They are remarkably similar to the Boolean logic operators we've already learned.

Operator	Name	Description
&	Bitwise AND	Turns bits off
|	Bitwise OR	Turns bits on
^	Bitwise Exclusive OR (XOR)	Flips bits
~	Bitwise NOT	Flips all bits in a number

This table doesn't have the space to give all of the information. Much more explanation is required. These are special mathematical operations with specific purposes. The Boolean AND, OR, and NOT operators are used for program logic, i.e. linking together conditions for `if` statements and the like. The bitwise operators in the table above are for manipulating the bits in numbers.

Bitwise AND

Bitwise AND is used to turn off bits in a number, that is, set certain 1 bits in a number to 0. This is done by comparing the corresponding bits in each number and applying Boolean logic to determine whether or not the bit should be a 1 or 0. Here are two examples that help show how this works.

Example 1: 255 & 240 = 240

	Decimal	Binary
	255	`11111111`
AND	240	`11110000`
	240	`11110000`

Example 2: 240 & 170 = 150

	Decimal	Binary
	240	`11110000`
AND	170	`10101010`
	150	`10100000`

The only time a bit stays on is when it is a 1 in the first AND second number. This is why the bitwise AND operator is used for turning bits off.

To turn a specific bit off, we have to figure out what number is needed to turn off only the bit desired and leave the rest on. This is as simple as using a number which has all bits on except for the one we want to turn off.

Let's say that we have a variable which has the value 199 and we want to turn off bit 2 and only bit 2. We'll start with the number 255 – which has all bits on – and subtract 2^2, which is 2 to the power of the number of the bit we want to turn off. The number we want to use with AND is 251, that is, 255 - 4. The result of 199 AND 251 is 195.

199 & 251 = 195

	Decimal	Binary
	199	`11000111`
AND	251	`11111011`
	195	`11000011`

Bitwise OR

Bitwise OR is used in the opposite way to AND – its purpose to turn *on* bits in a number, that is, set certain 0 bits in a number to 1.

Example 1: 192 | 15 = 207

	Decimal	Binary
	192	`11000000`
OR	15	`00001111`
	207	`11001111`

Example 2: 8 | 64 = 72

	Decimal	Binary
	8	`00001000`
OR	64	`01000000`
	72	`01001000`

A bit is turned on whenever either bit is a 1. The math is easier when turning on a specific bit. It is merely a matter of ORing the number we want to change with a value of 2 to the power of the number of the bit. If we have a variable containing 36 and we need to turn on bit 1, then we OR our variable with 2^1, which is 2.

36 | 2 = 38

	Decimal	Binary
	36	`00100100`
OR	2	`00000010`
	38	`00100110`

Bitwise XOR

Bitwise XOR is probably the weirdest of all of the bitwise operators. XOR stands for eXclusive OR. It is used for inverting bits because it returns 1 if either bit is a 1, but 0 if they have the same value.

36 ^ 255 = 219

	Decimal	Binary
	36	`00100100`
XOR	255	`11111111`
	219	`11011011`

XOR has the occasional useful application, but most of the time, it isn't needed.

Bitwise NOT

Bitwise NOT also flips bits like XOR does, but with less control. It flips all of the bits in a number, just like the example above, but it does not require a second value. Here is an example of how it can be used:

```
int a = 5;
printf("The value of ~%d is %d\n", a, ~a);
```

Shift Operations

In addition to flipping bits on and off, C and C++ provide shift operators which let us quickly do some fast and fancy multiplication and division.

a << b
: Shift the value of `a` to the left by `b` places. This multiplies `a` by 2^b

a >> b
: Shift the value of `a` to the right by `b` places. This divides `a` by 2^b

It's a little easier to figure out why it's called shifting by seeing what it actually does to the bits themselves.

Code	Mathematical Equivalent	Result	Value in Binary (before)	Value in Binary (after)
`5 << 2`	$5 * 2^2$	20	`00000101`	`00010100`
`32 << 1`	$32 * 2^1$	64	`00010000`	`00100000`
`64 >> 1`	$64 / 2^1$	32	`01000000`	`00100000`
`7 >> 2`	$7 / 2^2$	1	`00000111`	`00000001`

Knowing shift operations is especially handy when operating with values at the bit level because they allow us to do specific kinds math very, very quickly. Multiplying or dividing by a power of 2 is common in these situations and the equivalent calls to `pow()` or using regular division are far slower.

Bug Hunt

Hunt #1

Code

```c
#include <stdio.h>
#include <string.h>
#include <malloc.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 - 1 - i];
            string[length - 1 - i] = string[i];
            string[i] = temp;
      }

      return string;
}

int main(void)
{
      char firstString[100],
            secondString[100];
      char *combinedString = NULL;

      printf("Enter your first word: ");
      gets(firstString);

      printf("Enter your second word: ");
      gets(secondString);
```

```
	sprintf(combinedString,"%s %s",ReverseString(secondString),
			ReverseString(firstString));

	printf("If you saw these two words in a mirror, it would read '%s'\n",
			combinedString);
}
```

Errors

When run, the program prints "`segmentation fault`" and nothing else.

Hunt #2

Code

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

void MakeBinaryString(char *outString, char valueToConvert)
{
	// Convert a 1-byte value to a string that shows its value in
	// binary. We do this by checking to see each bit is on and
	// if it is, setting the character value to '1' or '0' if not.
	for (int i = 0; i < 8; i++)
	{
		// Is the bit a 1?

		// The shifting is needed to quickly generate a power of 2
		// so that we check only 1 bit at a time, starting with bit 7
		// and working its way down.
		if (valueToConvert & (1 << (7 - i)))
			outString[i] = '1';
		else
			outString[i] = '0';
	}

	outString[8] = '\0';
}


int main(void)
{
	char value = 5;
	char binaryString[6];

	MakeBinaryString(binaryString,value);

	printf("The binary equivalent of %d is %s\n",value, binaryString);

	return 0;
}
```

Errors

When run, the program prints the following:

```
The binary equivalent of 48 is 00000101
Segmentation fault
```

Lesson 6 Bug Hunt Answers

There was only one hunt because of how hard it was. It's a kind of error called an off-by-1 error. These are sometimes are to track down, like this one was. The errors and corrections are commented.

```
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++)
      {
            // This line from Lesson 6 is incorrect
//          char temp = string[length - i];
            char temp = string[length - 1 - i];

            // So is this one
//          string[length - i] = string[i];
            string[length - 1 - i] = string[i];
            string[i] = temp;
      }

      return string;
}
```

The reason why this causes the program to not print anything requires a little thought. The first line in the loop is supposed to save the last character of the string. By not subtracting 1 from the length, it saves the `NULL` terminator for the string into `temp` instead. The rest of the loop continues on pretty much as it should. `abcdef\0` becomes `\0fedcba`. Because the `NULL` terminator is at the very beginning of the string, when `printf()` goes to print it, it sees the terminator and stops, resulting in the reversed string not getting printed.