Learning to Program with Haiku

Lesson 8

Written by DarkWyrm

Because we've been focusing on the structure of the C++ language so far, we haven't given much attention to getting information in and out of our program. Today we'll change this, but first, let's take a quick look at a few niggling details that we've been able to ignore until now.

Scope

This is not to be confused with mouthwash, though considering the personal hygiene habits of some developers, might not be a bad idea, either. Luckily, on the Internet, nobody knows you're a cat.

Scope in this case applies to variables. A variable's scope is the span of its existence. When a variable is declared, it has a definite beginning, and it can't be used before its declaration in the source code. It also disappears completely when execution reaches the end of the block in which it is declared.

There are three types of scope: local, global, and static. Local variables have the shortest lifespan: they are declared within a code block, such as a function, `for` loop, or `case` block, and disappear when execution leaves the block of code.

Global and static variables are declared outside of a function. They can always be accessed, but the order in which they are initialized is not certain, so be careful of globals that depend on the values of other globals. Global variables can be accessed from anywhere. Static variables have the same lifetime as global variables, but they can be accessed only from code within the file in which they are declared. In other words, code in other files can't get at them. All of this can get tricky, but it is easier to understand in code, so let's look at an example.

```
#include <stdio.h>

// This is a global variable, accessible from anywhere
int gAppReturnValue = 0;


int AreaOfSquare(int size)
{
	return size * size;
}


int main(void)
{
	int returnValue = 1;

	// We actually start from 1 in this loop because measuring the area
	// of a square of 0 inches is pretty silly.
	for (int i = 1; i <= 12; i++)
	{
		// i is only accessible from within this loop.

		// If we had a need to utilize returnValue, we could do it here.

		int area = AreaOfSquare(i);
		printf("The area of a square %d inches on a side is "
			"%d square inches\n",i,area);
	}
```

```
	// If we try to access the area variable outside the loop,
	// the compiler will complain with an error something like
	// "foo.cpp:30: error: ‘area’ was not declared in this scope"

	return gAppReturnValue;
}
```

Having seen the above example, you should be aware of what names you give your variables. The lone global variable in this code has a lowercase 'g' for a prefix. This is to signify that it is a global. Static variables use the prefix 's' instead of 'g'. Get into the habit of using these prefixes to avoid more than a few headaches.

If you're like me, you're probably wondering what kind of headaches I'm talking about. Take a look at this example and ask yourself what value would be printed.

```
#include <stdio.h>

// This is a static variable, accessible from anywhere in this file
static int returnValue = 1;

int main(void)
{
	int returnValue = 2;

	printf("The return value of this program is %d\n",returnValue);

	return returnValue;
}
```

In case you're too lazy to try this sample out, the value is 2. This could've been easily avoided by naming our static variable `sReturnValue`. Local variables always receive precedence when there are conflicts in scope. It's also possible to hide a parameter or a previously-declared local variable this way, such as a variable in an `if` block hiding a variable declared at the beginning of a function. *Be careful not to hide parameters with the variables you declare in a function.*

Global and static variables are much like spices in cooking: used sparingly, they can really make writing programs easier, but used too much, all they will do is help you make a big mess. When in doubt, don't use global variables to pass data around – use function parameters instead. Doing so will make it much easier to reuse code that you've written. Once again, work smarter, not harder.

Constants

Not everything is subject to being changed in C and C++. Sometimes we want to ensure that some data not be altered at all. There are other times when we want to use a variable to remember some arbitrary value – we don't care what the value itself happens to be, but we do want to remember its purpose. When we get to writing Haiku programs that use windows and buttons, we will use these quite a lot to specify behavior for controls that we use, such as how they will be resized.

The first kind of constant is the preprocessor definition. In case you don't remember from Lesson 2, the preprocessor is the first tool used when compiling source code into an executable. The preprocessor removes comments, includes header files, and other basic text insertion and substitution. They look like this:

```
#define SOMEDEF   "I like cheese!"
#define STRACE(x) printf x
```

The format is simple: #define <definition> <what gets put there>. They are simple textual substitutions, much like if you opened a word processor and made it replace all occurrences of SOMEDEF with "I like cheese!" As far as the compiler is concerned, you never even typed SOMEDEF. You can even make them look like functions, like STRACE. We'll look at this one in more detail at the end of the lesson, so just sit tight on that one.

Even more so than arrays, `#defines` need to be handled carefully. It is good practice to put them in all capitals so that they do not look like functions. They also are not type-aware, and one other word of caution: no matter how much they beg, no matter how much they cry, never, **NEVER put a semicolon at the end of a preprocessor definition**.

```
// Do NOT do this
#define THISISBAD 1;
```

Doing so will result in errors in your code that seemingly have nothing to do with the real problem. These are the kind of bugs that make you feel as if you're aging prematurely, even if you're not.

Constant variables are the preferred means of remembering an arbitrary value because they have a defined type. This is done by placing the `const` keyword in the declaration of a variable. Because they are not allowed to be modified, you will almost always see a constant initialized when it is declared.

```
const int someConstantIntValue = 3;
```

Pointers can be made constant, as well, but it gets confusing *very* quickly if you're not careful. The `const` keyword can apply to the pointer's address, to the pointer itself, or both.

```
// This is just a constant integer that we'll use for some of the pointers below.
const int someVariable = 5;

// These are both pointers to an int where we can change the pointer's address,
// but not its value. These don't have to be initialized.
const int *ptrConstInt;
int const *anotherPtrConstInt;

// This is a constant pointer. The value itself can change, but we can't
// change the address to which the pointer points. It's pretty useless unless
// we initialize it.
int * const constPtrInt = &foo;

// These are constant pointers to a constant value. We can't change ANYTHING
// about it, which means that it has to be initialized to be of any use.
const int * const ptrReallyConstInt = &someVariable;
int const * const anotherPtrReallyConstInt = &someVariable;
```

What a mess! There is a rule of thumb which helps make sense of all of this confusion. **The `const` keyword applies to the element to its left. If there isn't anything there, then it applies to whatever is to the right.** In the first two examples above, every time you see `const int` or `int const`, it means that the pointer itself can be changed, but not the value in the address that it points to. Every time you see `* const`, it makes the pointer's address fixed, but the value at the pointer's address can be changed. The last two examples above combine these two techniques to make everything constant, both address and value. If you're still confused on this, don't worry too much – it's not just you. This is a hard topic.

Using Data From Outside: File Operations

The only way that we know how to get information into our program is `gets()`, and the only way to get it out is `printf()`. While `printf()` is just fine, `gets()` is actually dangerous and whenever the compiler encounters its use, it warns us. The reason is that there is no way to enforce how many characters are placed in the string passed to it. Crashing the program is as easy as typing more characters than the capacity of the array that is given to it. We're going to move on to a much better solution.

Moving information in and out of programs is normally done using streams. Information flows in or out of your program. Direct input from the user is a stream, and the screen is one also – one for output. Console programs utilize streams to get and print information, and they can even be linked together: the program that we use when we run Terminal, called bash, features an incredibly powerful ability to take what one program spits out and feed it to another one or dump it into a file.

There are three main streams that are available to every program: `stdin`, the standard input, `stdout`, the standard output, and `stderr`, the error output. Unless changed, a program that gets data from `stdin` will ask for input from the user just like we've done with `gets()` and sending anything to `stdout` or `stderr` will be printed on the screen.

Data can be brought into our programs by reading from a stream, and it can be sent out of our program by writing to one. Some streams are read-only, some are write-only, and some allow both reading and writing. `stdin` is read-only, so we can use it only for getting data. `stdout` and `stderr` are write-only, so we can only print to them. If we create a stream to operate on a file, we can choose either or both.

Each stream has an identifier, called a **handle**. In programming, a handle is just an arbitrary – but most often unique – number which is used to identify an object from others of its kind. Operating on a stream is as simple as obtaining a handle for the stream and calling the appropriate function. Let's take a look at the declarations for some of the functions that we will use in day-to-day C programming.

```
int printf(const char *format, …);
int fprintf(FILE *streamHandle, const char *format, …);
```

> An old standard. Given a string which defines the format for what is to be printed and an appropriate number of parameters afterward, prints a string to `stdout`. `fprintf()` requires a stream handle to be specified before the format string – making it possible to "print" directly to a file – but otherwise working exactly like `printf()`. When successful, both functions return the number of characters printed. A negative return value is used to indicate failure.

```
int ferror(FILE *streamHandle);
```

> This returns a 0 if everything is OK on the stream indicated by `streamHandle` and some other unspecified value if an error has occurred. Make sure that `streamHandle` is not `NULL` – it will segfault your program otherwise.

```
int feof(FILE *streamHandle);
```

This returns a 0 if everything is OK on the stream indicated by `streamHandle` and some other unspecified value if the stream has come to the end of the file. Make sure that `streamHandle` is not `NULL` – it will segfault your program otherwise.

```
char * fgets(char *array, int arraySize, FILE *streamHandle);
```

`fgets()` is the safe version of `gets()`. It reads in text from the stream handle until it encounters an endline (`'\n'`) character or it reads the number of characters equal to `arraySize`, making a segmentation fault possible if the programmer makes a mistake. As with `gets()`, when successful, `fgets()` returns the same pointer as `array`. If there is an error, it returns a `NULL` pointer. If `fgets()` comes to the end of the file – only encountered when reading from a file instead of `stdin` – the contents of the array remain unchanged and a `NULL` pointer is returned. Whenever a `NULL` pointer is returned by `fgets()`, use `feof()` and `ferror()` to figure out whether you have run out of data or something has gone wrong.

```
FILE * fopen(const char *filePath, const char *mode);
```

`fopen()` opens a file as a stream. If successful, it returns a stream handle which is used by other functions and must be eventually closed with `fclose()`. The mode strings are listed below.

Mode string	Function
"r"	Open a file for reading. The file has to exist.
"w"	Open a file for writing. If the file exists, its contents are erased and it is treated as a new empty file.
"a"	Open a file for writing. Any data written to it is added to the end of the file. If the file doesn't exist, it is created
"r+"	Open a file for updating, supporting both reading and writing. The file must already exist.
"w+"	Open a file for updating, supporting both reading and writing. If the file exists, its contents are erased and it is treated as a new empty file.
"a+"	Open a file for reading and appending. Reading can be done from anywhere in the file, but all writes are tacked onto the end of the file.

```
int fclose(FILE *streamHandle);
```

Close up an opened stream handle.

Wow. That's a lot of different functions! The scary part is that this is only a very small portion of the functions available. A programmer should always be learning. Once he is comfortable with a language, he is often learning about available functions that are new to him and new ways to use the functions that he knows. The best way to learn how to use functions that are new to you is to use them.

Let's have a look at a program which prints a test file to `stdout` and creates that file if necessary.

```
#include <stdio.h>

int
FileExists(const char *path)
{
	// This function tests for the existence of a file by trying
	// to open it. There are better ways of doing this, but this
	// will work well enough for our purposes for now.

	// If we were given a NULL pointer, we will return a -1 to
	// indicate an error condition.
	if (!path)
		return -1;

	// Attempt to open the file for reading
	FILE *file = fopen(path,"r");

	// our return value will be 1 if the file exists and 0 if
	// it doesn't. ferror() will return a nonzero result if
	// there was a problem opening the file and a 0 if the file
	// opened OK.
	int returnValue;

	// ferror will crash if given a NULL pointer
	if (!file || ferror(file) != 0)
		returnValue = 0;
	else
	{
		fclose(file);
		returnValue = 1;
	}

	return returnValue;
}


int
MakeTestFile(const char *path)
{
	// Always check for NULL pointers when dealing with strings
	if (!path)
		return -1;

	// Open the file and erase the contents if it already exists.
	FILE *file = fopen(path,"w");

	if (!file || ferror(file))
	{
		// We have a different error code if we couldn't create the
		// file. This makes it possible for us to know if we messed up
		// by passing a NULL pointer or if there was a file-related error.
		fprintf(stderr,"Couldn't create the file %s\n",path);
		return 0;
	}
```

```
	// The stream handles for stdout, stdin, and stderr are already defined
	// for us, so we can use them without any extra work, like in the if()
	// condition above and where we put data into our file below.
	fprintf(file,"This is a file.\nThis is only a file.\n"
	                         "Had this been a real emergency, do you think I'd "
	                         "be around to tell you?\n");
	fclose(file);
	return 1;
}


int
main(void)
{
	int returnValue = 0;
	
	// Let's use a test file in /boot/home called MyTestFile.txt.
	const char *filePath = "/boot/home/MyTestFile.txt";
	
	// Make the test file if it doesn't already exist and bail out of
	// our program entirely if there is a problem creating it.
	if (!FileExists(filePath))
	{
		returnValue = MakeTestFile(filePath);
		if (returnValue != 1)
			return returnValue;
	}
	
	printf("Printing file %s:\n",filePath);
	
	// We got this far, so it's safe to print the file
	FILE *file = fopen(filePath,"r");
	
	if (!file || ferror(file))
	{
		fprintf(stderr,"Couldn't print the file %s\n",filePath);
		return 0;
	}
	
	char inString[1024];
	
	// fgets will return a NULL pointer when it reaches the end of the
	// file, so this little loop will print the entire file and quit
	// at its end.
	while (fgets(inString,1024,file))
		fprintf(stdout,"%s",inString);
	
	fclose(file);
	
	return 0;
}
```

Whew! This is our longest example yet. It's also our closest code to a "real" program. Some idioms, like `if (!file)`, are very common to C and C++ programming, so get used to seeing them. Read over the code in this example and make sure that you understand what each line does.

There are some small changes in the style, as well. Attention to good style is a quirk of the Haiku ecosystem. The Haiku developers, in particular, are notably picky about code which adheres to the OpenTracker code style guidelines and with good reason. Style is partly a matter of opinion, but good code style can also help with debugging and avoiding errors. Bad code style can make it much, much harder. The style we will use throughout the rest of these lessons does not hold to the official Haiku guidelines in certain ways, but it does follow them pretty closely.

Bug Hunt

Hunt #1

Code

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

char *ReverseString(const 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

```
foo.cpp: In function ‘char* ReverseString(const char*)’:
foo.cpp:18: error: assignment of read-only location ‘*(string + ((unsigned int)
((length + -0x00000000000000001) - i)))’
foo.cpp:19: error: assignment of read-only location ‘*(string + ((unsigned
int)i))’
foo.cpp:22: error: invalid conversion from ‘const char*’ to ‘char*’
```

Answers from Lesson 7's Bug Hunt

1. The pointer `combinedString` doesn't point to a valid memory address. It needs to either be given heap memory by `malloc()` – which is later freed – or declared on the stack as an array.
2. The size of `binaryString` array in `main()` is too small. It needs to be at least big enough to hold 1 character per bit in a byte plus 1 for the `NULL` terminator, so `binaryString` must be at least 9 characters instead of 6.