Learning to Program with Haiku

Lesson 10

Written by DarkWyrm

Back when we first took a look at pointers, I mentioned that they are a powerful tool. Today we'll see how they can be used with functions and learn some different ways of passing variables to functions.

Default Values for Parameters

When calling functions, sometimes there are certain parameters that almost never change. Let's say that in a program we're writing we have a function to create a window on the screen that looks like this:

```
void MakeWindow(int left, int top, int right, int bottom, const char *title,
                int type, int look);
```

Each time we use this function, we end up having different values for the location, size, and title, but the type and the look don't change – almost all of the time the same numbers for `type` and `look` are used because we're making regular windows. This makes for extra typing, so we may want to use a default value for the type. To do this, we change the declaration slightly:

```
void MakeWindow(int left, int top, int right, int bottom, const char *title,
                int type = REGULAR_WINDOW_TYPE, int look = REGULAR_WINDOW_LOOK);
```

We are, in a sense, initializing the parameter so that we don't always have to type in a value when we call the function. Parameters with default values always go at the end of a parameter list in a function call. In this particular case, a lot less typing is involved for the usual case of a “regular” kind of window:

```
MakeWindow(100,100,500,400, “MyWindow”,REGULAR_WINDOW_TYPE,REGULAR_WINDOW_LOOK);
```

becomes

```
MakeWindow(100,100,500,400, “MyWindow”);
```

We won't see too many instances of default parameter values until we start writing programs that use Haiku's graphical toolkit, but it's good to know about these beforehand.

References

References, in this case, are not something you would find in the library. Instead, they are a hybrid of a pointer and a regular variable. They are treated like a regular variable in terms of not having to use the an asterisk to get its value, but they don't have their own memory for storing data. Instead, they use another variable's space. Here is an example of a reference declaration:

```
int myInt;
int &myRef = myInt;
```

The ampersand (`&`) in between `int` and `myRef` is the difference between declaring a regular variable and a reference. In this example, `myRef` is a reference to `myInt`. `myRef` is, for all practical purposes, just another name for `myInt`. Changing one will change the value of the other. This works with pointers that have a valid memory location, too. Observe:

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

```
int
main(void)
{
	// Create a pointer with some heap memory and initialize it
	int *myPointer = (int*) malloc(sizeof(int));
	*myPointer = 1;

	// Create a reference to the myPointer's location.
	int &myRef = *myPointer;

	myRef++;

	printf("The value at myPointer's location is %d\n",*myPointer);

	free(myPointer);

	return 0;
}
```

References are a feature that makes you want to say, "OK, that's nice. Let's move along now," but there's more than meets the eye here. First of all, you can't change the location that a reference points to, so `myRef` will always refer to the address that `myPointer` points to – even if `myPointer` changes – so trying to use `myRef` after `myPointer` has been freed will cause a segmentation fault. This unchangeability is actually a good thing. A reference has to be initialized and can't be uninitialized, so it will always be valid unless the memory it refers to has been freed or if the variable it refers to goes out of scope. This makes references a great deal safer than pointers at the expense of some flexibility.

Parameter Use: by Reference or by Value

There is more than one way to get data out of a function. The usual way is the value it returns, but references provide another. Normally, when a parameter is passed to a function, a copy of the variable is passed to the function – the *value* of the parameter has been sent to the function, not the variable itself. This means that you can change the parameter without causing trouble outside the function. When a reference to a parameter is given to a function, any changes that the function makes to it continue on after it exits. Changing a function to pass a parameter by reference is as simple as adding an ampersand (&) in front of the parameter's name.

```
const char * someFunction(int &integerByReference);
float someOtherFunction(const double &doubleByReference);
```

In this example, not only does `someFunction` spit out a string, but it can also tweak `integerByReference`, allowing us to get two values from one function call instead of just one. `someOtherFunction` uses a constant reference. While this might seem silly – passing a reference that you can't change – but it does have a purpose: it saves a copy. This is one way to speed up a function that is frequently called, especially if it has a lot of parameters or if a parameter takes up a lot of memory.

Just to make sure that we have a pretty good handle on the difference between passing a variable by value and passing one by reference, let's look at a code example that illustrates the difference.

```
#include <stdio.h>

// This function passes x by reference and y by value
int
myFunction(int &x, int y)
{
      x = x * 2;
      y = y + 5;
      return x * y;
}


int
main(void)
{
      // Create a couple of variables for our work
      int foo = 5;
      int bar = 10;

      int outValue = myFunction(foo,bar);

      printf("foo is %d, bar is %d, and myFunction(foo,bar) is %d\n",
                  foo,bar,outValue);

      return 0;
}
```

In this example `foo` starts with a value of 5, but because `myFunction()` changes it, its value is 10 when it is printed. `bar` is not changed because `myFunction()` changes a *copy* of `bar`.

Pointers to Functions

Just when you thought pointers couldn't get any weirder, it gets worse. It is possible to not only have a pointer whose address holds a value, pointers can point to an address containing code.

```
void (*functionPointer)(int value, int anotherValue);
```

This snippet of code is **not** a function. It is actually a declaration of a pointer called `functionPointer`. Parentheses are placed around the asterisk and name of the pointer to make sure that it is declared as a function pointer instead of a function which returns a `void` pointer.

Types for function pointers are very specific. The return value and the number and types of parameters are all part of a function pointer's type. These two function pointers are not the same type:

```
void (*integerFunction)(int value);
int (*anotherIntegerFunction)(int value);
```

Executing a function by way of a pointer is dead easy: Treat the pointer as the name of the function. This example calls the function held by the `integerFunction` pointer:

```
integerFunction(5);
```

Like references, the uses of function pointers are not immediately obvious. They add incredible flexibility to a program. Code can be bolted on or changed out just like parts on a car. Interpreted

languages, such as Python or Perl, make it very easy to change a program on the fly, but this is not very easy at all for a compiled language like C++. Although normally pretty rare, we'll use function pointers quite a bit when we look at program addons later on. For now, don't worry too much about them.

Pointers to pointers

Yes, pointers can point to pointers. The address a pointer holds can easily be the address of another pointer. This is just a matter of adding a second asterisk when declaring a pointer.

```
char **somePointerToAPointer;
```

Don't forget that a pointer's declaration doesn't allocate any memory! The only thing that exists after this declaration is `somePointerToAPointer`. We can use it for different things, such as getting a pointer from a function without using a return value or creating a list of strings. Yes, this is the way you create a fixed list of strings on the heap. This is also how we get arguments from the command line.

Command Line Arguments

Just like functions taking parameters – or arguments... they're the same thing – programs themselves can have information passed to them. Take, for example, this Terminal command:

```
$ rm -f --verbose myFile
```

The command `rm` has three arguments: a filename and two switches. Command line switches are options that change the behavior of a program without being the information on which it operates. In this case, `rm` operates on the file `myFile`. The `-f` switch tells it to force removal, not asking for confirmation, and the `--verbose` switch tells it to print more information to the screen than it normally does.

Switches in Windows begin with a slash, but for Linux, OS X, and Haiku, they begin with a dash. As a general rule, switches with only one dash have only one letter and switches with two dashes are generally words and phrases separated by a dash. We're not going to focus too much on command line switches for the purposes of these lessons just because our projects will not be complex enough to warrant using them.

In order for a program to take advantage of command line arguments, the way we use `main()` must change a little bit:

```
int
main(int argc, char **argv)
{
	return 0;
}
```

Now `main()` takes two parameters: `argc`, which is the number of arguments from the command line and `argv`, which is a list of strings which contain the command line arguments. Treat `argv` like an array – if `argc` is 2, then `argv` will have elements numbered 0 and 1. `argv[0]` always contains the name of the program when it was run. This program will print the command line arguments passed to it.

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

```
int
main(int argc, char **argv)
{
	// Iterate through all arguments and print them.
	for (int i = 0; i < argc; i++)
		printf("Program argument %d: %s\n", i, argv[i]);

	return 0;
}
```

The place to look closely is the end of the `printf()` statement. Using a pointer to a pointer is literally just like using a multidimensional array. Strings are just special `char` arrays, so we are accessing lists of characters individually here. If we wanted to use just the second character of the first argument, we'd use `argv[0][1]`. In case it's a little fuzzy, bracket order goes from the largest group to the smallest from left to right, so we would be accessing element 0 in the list of lists and choosing element 1 – the second character – from that list.

Project

With everything we've learned in these ten lessons thus far, we have the capacity to do a great deal. This will be our first project of notable value. We are going to write a simple version of the command line utility `cat`, which concatenates, or joins together, files by printing them to `stdout` in the order that they are specified from the command line.

For this project, we will need to use two new functions: fread and fwrite.

```
size_t fread (void *buffer, size_t size, size_t count, FILE *stream);
size_t fwrite (void *buffer, size_t size, size_t count, FILE *stream);
```

`fread` reads in data from a file stream. This function will try to read in `size * count` bytes and place the data into `buffer`. This is really handy because you can allocate any kind of array for `buffer`, use the `sizeof()` function on its type for `size`, and send the number of elements in the array as `count`. `fread` returns the number of elements actually read. If it is not the same as what was requested, there was either an error or the end of the file was reached.

`fwrite` works the same way as `fread`, but in reverse. Data in `buffer` is written to `stream` and the number of elements written are returned.

Conceptually, this will involve what we have just learned about using command line arguments in combination with file operations from Lesson 8 plus `fread` and `fwrite`. Use a `for` loop to execute the following set of steps on each argument.

1. Try to open the argument for reading as a file.
2. If the open fails, skip to the next iteration.
3. If the open succeeds, try to read a chunk of data from the file, storing the number of bytes read into a variable.
4. Use a `while` loop to read sections of data from the file, repeating while the number of bytes read is greater than 0.
    a. Write the number of bytes read to `stdout`.
    b. Try reading some more data from the file handle, storing the number of bytes read.

5. Close the file's handle

Hints, Warnings, and Advice

- Printing file errors using `ferror` might be a nice touch.
- The chunk of memory you store the file data in can be from the stack or from the heap.

Because we haven't done much actual writing of code, I'll do a little to get you going. All you'll have to do is replace each of the comments with the corresponding code. It might be a good idea to write a little bit at a time and recompile. Writing code incrementally and testing it helps keep bugs small and easy to locate.

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

int
main(int argc, char **argv)
{
	for (int i = 1; i < argc; i++)
	{
		// open a file handle for reading from argv[i]

		// if the file handle is NULL or there is an error, continue to
		// the next iteration.

		// create a data buffer -- an array to hold our data. Size isn't
		// terribly important, but it should be at least a few hundred bytes
		// and no more than about 4000 bytes. You can create it on the stack
		// or use malloc, whichever you prefer.

		// create a variable to store the number of bytes actually read

		// read data from the file handle and store the number of bytes
		// read into the variable that we just created.

		// Start our while() loop. Loop while the number of bytes read is
		// greater than zero and if ferror does not indicate an error on
		// the file handle
		{
			// write the number of bytes read to stdout

			// read more data and put the number of bytes actually read
			// into the variable we created above.
		}

		// free the buffer here if you used malloc, never mind if you put
		// the buffer on the stack.

		// close the file handle here
	}

	return 0;
}
```