More on Functions

Now that you’ve completed Chapter 8, you have a good grounding in the essentials of creating and using functions. In this chapter you’ll build on that foundation by exploring how functions can be used and manipulated; in particular, you’ll investigate how you can access a function through a pointer. You’ll also be working with some more flexible methods of communicating between functions.

In this chapter you’ll learn:

- What pointers to functions are and how you use them
- How to use static variables in functions
- How to share variables between functions
- How functions can call themselves without resulting in an indefinite loop
- How to write an Othello-type game (also known as Reversi)

Pointers to Functions

Up to now, you’ve considered pointers as an exceptionally useful device for manipulating data and variables that contain data. It’s a bit like handling things with a pair of tongs; you can manipulate a whole range of hot items with just one pair. However, you can also use pointers to handle functions at a distance. Because a function has an address in memory where it starts execution (i.e., its starting address), the basic information to be stored in a pointer to a function is going to be that address.

If you think about it, though, this isn’t going to be enough. If a function is going to be called through a pointer, information also has to be available about the number and type of the arguments to be supplied and the type of return value to be expected. The compiler can’t deduce these just from the address of the function. This means that declaring a pointer to a function is going to be a little more complicated than declaring a pointer to a data type. Just as a pointer holds an address and must also define a type, a function pointer holds an address and must also define a prototype.

Declaring a Pointer to a Function

The declaration for a pointer to a function looks a little strange and can be confusing, so let’s start with a simple example:

```c
int (*pfuction)(int);
```

This declares a variable that is a pointer to a function. It doesn’t point to anything—yet; this statement just defines the pointer variable. The name of the pointer is pfuction, and it’s intended to point to functions that have one parameter of type int and that return a value of type int to the calling function. Furthermore, you can only use
this particular pointer to point to functions with these characteristics. If you want a pointer to functions that accepts a float argument and returns float values, you need to declare another pointer with the required characteristics. The components of the declaration are illustrated in Figure 9-1.

![Diagram showing how to declare a function pointer](image)

**Figure 9-1. Declaring a pointer to a function**

There are a lot of parentheses in a “pointer to function” declaration. In this example, the \*pf\(\text{unction} part of the declaration must be between parentheses. If you omit the parentheses, you’ll have a declaration for a function called \text{function}() that returns a value that’s a pointer to int, which isn’t what you want here. The second pair of parentheses just encloses the parameter list in the same way it does with a standard function declaration. A pointer to a function can point only to functions with a given return type and a given number of parameters of given types. You can assign whatever name you like, just as with any other pointer variable.

### Calling a Function Through a Function Pointer

Suppose you define a function that has the following prototype:

```c
int sum(int a, int b); // Calculates a+b
```

This function has two parameters of type int and returns a value of type int, so you could store its address in a function pointer that you declare like this:

```c
int (*pf\text{un})(int, int) = sum;
```

This declares a function pointer with the name \text{pf\text{un}} that will store addresses of functions with two parameters of type int and a return value of type int. The statement also initializes \text{pf\text{un}} with the address of the function \text{sum}(). To supply an initial value, you just use the name of a function that has the required prototype.

You can now call \text{sum}() through the function pointer like this:

```c
int result = pf\text{un}(45, 55);
```

This statement calls the \text{sum}() function through the \text{pf\text{un}} pointer with argument values of 45 and 55. You use the value returned by \text{sum}() as the initial value for the variable \text{result}, so the result will be 100. Note that you use the function pointer name just like a function name to call the function that it points to; no dereference operator is required.