In which our hero returns to function land, now triumphant, as our hero understands the essential machinery of function call and return, how stack variables are allocated and deallocated, and now is ready to write a function that can reach into the caller’s variables and change them as desired.

17.1 The Return Of The King

In the Fellowship of the Ring [T54], the last portion of a poem about Aragorn (son of Arathorn) reads as follows:

From the ashes a fire shall be woken,
A light from the shadows shall spring;
Renewed shall be blade that was broken,
The crownless again shall be king

There, Tolkien was writing about a rebirth of sorts, in this case, of a King. Here, we instead write about the rebirth of the function, but this time, with the ability to change the value of parameters that are not local to it. So how do we do it? The answer, as it always is in C, is with pointers.

17.2 Don’t Pass The Value, Pass The Pointer To Value

Imagine once more we are trying to write a function that incremented the value of a variable. The limitation that we need to overcome is that C passes by value; thus, we cannot simply pass the variable. Rather, we must pass a pointer to the value that we wish changed. Before proceeding, take out a piece of paper, and try to apply your knowledge of pointers to write a function that will change the value of a variable indirectly. Do it! And then turn the page and see if you got it right.
Here is the solution to our problem, in just a few short lines of C. Hopefully, you figured it out before peeking:

```c
void increment(int* x) {
    *x = *x + 1;
}

int main() {
    int f = 10;
    increment(&f);
    return 0;
}
```

As you can see, instead of passing the value of `f`, we pass its address, which is copied onto the stack as an argument to `increment()`. We then use an indirect reference to `f`, via the pointer `x`, to read the value of `f` (on the righthand side of the assignment), add one to it, and then to store the new value into `f`. Both the indirect read and write of `f` are accomplished via the dereference operator, `*`.

Figure 17.1 presents a graphical depiction. As shown, there is nothing special going on here. The same exact pass-by-value semantics are used, and the address of `f` (in the example, at address 1000) is placed on the stack as local variable `x` (a pointer).

### 17.3 A More Complex Example

Before we get too happy with ourselves (and watch out, we do get pretty happy sometimes), let’s take a look at a more complex example and see if it makes sense. In this scenario, let’s say we want to define a function that swaps the value of two pointers to integers. Here’s our first attempt at this code; can you spot what’s wrong?

```c
void swap_intp(int* p1, int* p2) {
    int* tmp = p1;
    p1 = p2;
    p2 = tmp;
}
```
Unfortunately, the assignments here (Lines 3, 4) do not work as desired. The reason is the same as before with the broken version of increment() — p1 and p2 are parameters to swap_intp() and hence changing them does nothing but change those parameter values on the stack. Figure 17.2 shows why this does not work.

So, what should we do to make swap_intp() work as desired? The answer is simple: if you want to change a variable of any kind, pass a pointer to that type of variable. Thus, when you wish to change an int in function, pass an int*. Here, we have a type int* and we wish to change it; thus, we should pass a pointer to a pointer to an int, an int**:

```c
void swap_intp(int** p1, int** p2) {
    int* tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
```

When we call this function, we do exactly what we did before, which is pass the addresses of the variables we wish to update:

```c
int main() {
    int a = 10;
    int b = 20;

    int* aptr = &a;
    int* bptr = &b;
    swap_intp(&aptr, &bptr);
    ...
}
```

Figure 17.3 shows what happens graphically on the stack during the correct version of swap_intp(). As you can see, by pass-
(a) Before Call

| bptr | 1000 |
| b   | 20   |
| a   | 10   |

| aptr | 1004 |
| p2  | 992  |
| p1  | 996  |
| a   | 1004 |
| b   | 20   |

(b) During Call

| tmp  | 1004 |
| ret addr | 996 |
| p2  | 992  |
| p1  | 996  |
| a   | 1004 |
| b   | 20   |

| Indirect refs to aptr/bptr |
| Original aptr/bptr changed |
| bptr | 1004 |
| b    | 20   |
| a    | 10   |

(c) After Call

| aptr | 1000 |
| p2  | 992  |
| p1  | 996  |
| a   | 1000 |
| b   | 20   |

Figure 17.3: Before, During, And After Correct swap_intp()

ing a pointer to the pointers we wish to change. The function can then use the dereference operator to change the values in the caller’s stack frame as desired. Spend the time and work through the value changes to make sure you understand how this works.

17.4 Common Mistakes And Issues

Before closing our introduction to functions (started back in Chapter 13), let’s go over some common problems that arise when using them. One common issue is trying to use a function that has not yet been defined or declared. For example, imagine the following bit of code:

```c
int main() {
    foo();
    return 0;
}

void foo() {
    ...
}
```

If you try to turn this code into an executable program (by compiling it, which we’ll talk more about soon), C will complain:

warning: implicit declaration of function 'foo'

The problem encountered here is that when `foo()` is encountered in `main()`, C doesn’t yet know anything about it; only below in the code is `foo()` defined (and declared). The rule that you must follow here is simple: you can only call functions that have previously been defined or declared.
Tip: To Change A Variable In A Function, Pass A Pointer To It

From the examples in the text, we can create a general rule: whenever you wish to change the value of a variable in a function, make sure to pass a pointer to that variable to the function, and dereference it to change its value. Note that this holds true of any type, including pointers themselves. Don’t get the rule wrong in your head; we’ve seen students see that a parameter is a pointer and say “good enough!” when in reality ...

Function definition is what we’ve mostly been focused on thus far; for example, in the code snippet above, two functions (main() and foo()) are both defined. Function declaration, on the other hand, does not specify the actual implementation of the function (the body of code between the braces). Rather, declaration simply specifies the return type, the name of the function, and the types of each parameter.

For example, we can transform the code above into legal C code by adding a declaration of foo() before its usage in main():

```c
// declaration of foo()
void foo(int);

int main() {
    foo(10);
    return 0;
}

void foo(int x) {
    ...
}
```

Later on, we’ll see that header files often contain lots of these types of declarations. The reason C demands them: a declaration allows the compiler to perform some type checking on your function call. With knowledge of what types (in this case, a single int) are being passed to the function, and what type it returns (in this case, void), C can make sure you are using the function properly (and not passing it the wrong arguments). We’ll talk more about type checking later, and how it improves the correctness of programs.

With knowledge of how a function is supposed to be called, C will not let you call a function with the wrong number of arguments, and will complain if you pass the wrong type to a function. For example, let’s look at the following:
One of the most important rules of C, which you must remember: check return codes of the functions you call. Only in rare cases where you really know what you are doing is it OK not to. We’ll discuss those cases as we see them; usually, it’s better to be safe than sorry.

```c
int add(int x, int y) {
    return x + y;
}

int main() {
    int x = add(3);
    return x;
}
```

If you try to turn this into an executable, C will complain:

```
error: too few arguments to function 'add'
```

However, C doesn’t mind if you ignore the return value of a function. Unfortunately, this is one of the cardinal sins in C programs — you should (almost) always check what a function returns, as often C library routines will signal errors in this manner.

For example, C will not complain about the following code snippet, even though it completely ignores the return value of `add()`:

```c
int main() {
    add(3, 4);
    return 0;
}
```

In fact, ignoring return codes (or not checking them properly, a different but related problem) was such a big problem in C that when the designers of Java created, well, Java, they made sure any errors were signaled in a way the programmer could not casually forget about (with `exceptions`) [G+96].

### 17.5 Summary

We focus here upon the machinery of the function call. Doing so carefully avoids a harder question: how to decide what code should go into a function? How can you, as a programmer, come up with the right abstractions? The best way to answer these tougher questions is to program, program, program; like any profession, the more you put in, the better you get. As Macklemore says, “10,000 hours” [M12].
References

Addison Wesley Publishing Company, 1996
The people behind Java. Java is a great language, except for many of its features. And, as Robert Pike said, Java feels bureaucratic, like you’re constantly filling out forms rather than writing code. But, some of those pesky bureaucratic features are quite useful, like the integrated exception handling mentioned above.

[M12] “Ten Thousand Hours”
Macklemore (with Ryan Lewis)
Macklemore LLC, 2012
Hip hop artists know that you’ve got to put in the time to get great at something. As Macklemore eloquently puts it:
“The greats weren’t great because at birth they could paint
The greats were great cause they paint a lot”

[T54] “The Fellowship of the Ring”
J.R.R. Tolkien
George Allen & Unwin, 1954
From the poem “All That is Gold Does Not Glitter”. We think Bilbo wrote it, but are less sure whether these hobbits are real.