CHAPTER P

Pointers

AS PART OF WRITING an implementation of a sorting algorithm, we often need to exchange the values of two items in a data structure. Good programming practice suggests that when we have some commonly reused code we should wrap it in a function:

```
void swap(int a, int b) {
        int t = a;
        a = b;
        b = t;
}
```

and then we can call it from our program like `swap(a,b)`. However, if we initially set `x=3; y=5;` and run the above swap function, the values of `x` and `y` remain unchanged. This should be unsurprising to us because we know that when we pass parameters to a function they are passed "by value" (also known as "pass-by-copy").

This means that `a` and `b` contain the same values as `x` and `y` at the moment that the call to `swap()` occurs because there is an implicit assignment of the form `a=x; b=y;` at the call site. From that point on in the function `swap()`, any changes to `a` and `b` have no effect on the original `x` and `y`. Thus, our swap works fine inside the scope of `swap()` but once the scope is destroyed when the function returns, the changes are not reflected in the calling function.

Java Content

We then wonder if there is another way to write our `swap()` so that it succeeds. Our first inclination might be to attempt to return multiple values. In C, like in Java, this is not possible directly. A function may only return one thing. However, we could wrap our values in a structure or array and return the one aggregate object, but then we have to do the work of extracting the values from the object, which is just as much work as doing the swap in the first place. We soon realize that we cannot write a swap function that exchanges the values of its arguments in any reasonable fashion. If we are programming in Java, this is where our attempts must stop. However, in C, we have a way to rewrite the function to allow it actually work.

In fact, this should not be surprising because we are essentially asking a function to return multiple pieces of data, and we have already seen one function that can do that: `scanf()`. If we pass `scanf()` a format string like `"%d %d"` it will set two argument variables to the integers that the user inputs. In essence, it is doing what we just said could not be done, it is modifying the values of its parameters. How is that accomplished? The answer is by using pointers.

1.1 Basic Pointers

A **pointer** is a variable that holds an address. An **address** is simply the index into memory that a particular value lives at. Memory (specifically RAM) is treated as an array of bytes, and just like our regular arrays, each element has a numerical index. We haven't needed pointers until now because we have given our variables names and used those names to refer implicitly to these locations. We don't even know the actual locations because the compiler and the system automate the layout and management of many of our variables. However, unlike in Java, it is possible in C to ask for the location in memory of a particular variable.

Referring to a location via a name or an address is not something unique to C, or even to computers in general. For example, you can refer to the room in which I work at Pitt as "Jon's Office," which is a name, or as "6213 Sennott Square" which is an address. Both are ways of referring to the same location, and in fact, this duplication is known as *aliasing*.

1.1.1 Fundamental Operations

We can declare a pointer variable using a special syntax. Pointers in C have a type to them just like our variables did, but in the context of pointers, this type indicates that we are pointing to a memory location that stores a particular data type. For instance, if we want to declare a pointer that will hold the address in memory of where an integer lives, we would declare it as:

```c
int *p;
```

where the asterisk indicates that `p` is a pointer. We need to be careful with declaring multiple variables on one line because its behavior in regards to pointers is surprising. If we have the declaration:

```c
int *p, q;
```

or even:

```c
int* p, q;
```

we get an integer pointer named `p` and an integer named `q`. No matter where you place the asterisk, it binds to the next variable. To avoid confusion, it is best to declare every variable on its own separate line.

To set the value of a pointer, we need to be able to get an address from an existing variable name. In C, we can use the address-of operator, which is the unary ampersand (`&`) to take a variable and return its address:

```c
int x;
int *p;

p = &x;
```

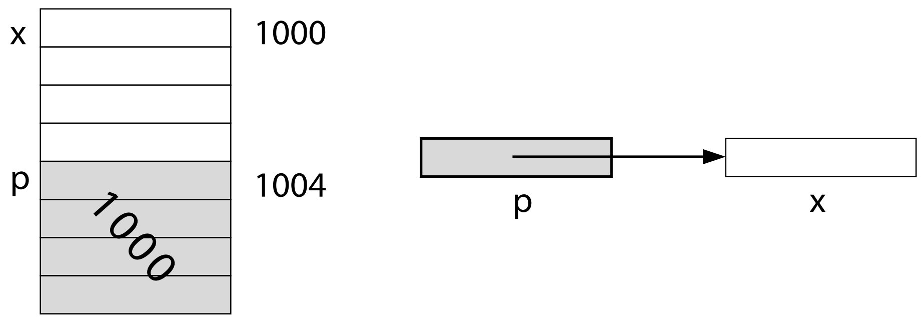

Figure 1.1: Two diagrammatic representations of a pointer pointing to a variable.

This code listing declares an integer x that lives someplace in memory and an integer pointer p that also lives somewhere in memory (pointers are variables too). The assignment sets the pointer p to "point to" the variable x by taking its address and storing it in p.

Figure 1.1 shows two ways of picturing this relationship. On the left, we have a possible layout of RAM, where x lives at address 1000 and p lives at address 1004. After the assignment, p contains the value 1000, the address of x. On the right, much more abstractly, is shown the "points-to" relationship.

Now that we have the pointer p we can use it as another name for x. But in order to accomplish that, we need to be able to traverse the link of the points-to relationship. When we follow the arrow and want to talk about the location a pointer points-to rather than the pointer itself, we are doing a **dereference** operation. The dereference operator in C is also the asterisk (*). Notice that although we used the asterisk in the pointer definition to declare a variable as a pointer, the dereference operator is different.

When we place the asterisk to the left of a pointer variable or expression that yields a pointer, we chase the pointer link and are now referring to the location it points to. That means that the following two statements are equivalent:

```
x = 4;              *p = 4;
```

Note that it is usually a mistake to assign a pointer variable a value that is not computed from taking the address of something or from a function that

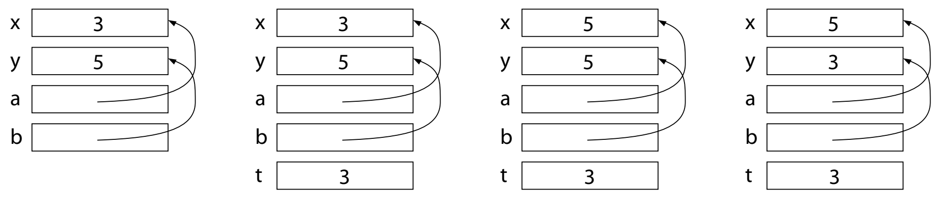

Figure 1.2: The values of variables traced in the `swap()` function.

returns a pointer. This general rule should remind us that `p = 4;` would not be appropriate because we do not normally know in advance where memory for objects will be reserved.

1.2 Passing Pointers to Functions

With the basic operations of address-of and dereference, we can begin to use pointers to do new and useful tasks. If we make a slight modification to our previous `swap()` function, we can get it to work:

```
void swap(int *a, int *b) {
        int t = *a;
        *a = *b;
        *b = t;
}
```

We also need to change the way we invoke it. If we have our same variables, `x` and `y`, we would call the function as `swap(&x, &y)`. After `swap()` returns, we now find that `x` is 5 and `y` is 3. In other words, the swap worked.

To better understand what happened here, we can trace the code constructing a picture like before. Figure 1.2 shows the steps. When the `swap()` function is called, there are four variables in memory: `x` and `y` which contain 3 and 5, respectively, and `a` and `b` which get set to the addresses of `x` and `y`. Next, a temporary variable `t` is created and initialized to the value of what `a` points to, i.e., the value of `x`. We then copy the value of what `b` points to (namely the value of `y`) into the location that `a` points to. Finally, we copy into the location pointed to by `b` our temporary variable.

Our swap function is an example of one of the two ways that pointers as function parameters are used in C. In the case of `swap()`, `scanf()`, and `fread()`, among many others, the parameters that are passed as pointers are actually acting as additional *return values* from the functions.

The other reason (which is not mutually exclusive from the previous reason) that pointers are used as parameters is for time efficiency in passing large objects (arrays, structs, or arrays of structs). Since we have already established that C is pass-by-value, if we pass a large object to a function, that object would have to be duplicated and that might take a long time. Instead, we can pass a pointer (which is really just integer-sized), thus taking no noticeable time to copy. This is why `fwrite()` takes a pointer parameter even though it does not change the object in memory.

1.3 Pointers, Arrays, and Strings

With our knowledge of pointers, we now can understand why we had to do certain things such as prefix variable names with an ampersand for `scanf()`. However, you may recall there was an exception to that rule. String variables did not need (in fact would not work) the address-of operator. To understand why this is the case, we need to explain a fundamental identity in C:

$$\text{pointer} \equiv \text{name of an array} \equiv \text{name of a string}$$

Imagine we declare an array: `int a[4];`. If we wish to refer to a particular integer in the array, we can subscript the array and write something along the lines of `a[i]`, which represents the i^{th} item. An alternative way to view it is that `a[i]` is i integers away from the start of the array. This is valid because we know that all elements of an array are laid out consecutively in memory. Thus, in essence, if we knew where the entire array started, we could easily add an offset to that address and find a particular element in the array.

To express it mathematically, if we have an array with address base, then the i^{th} element lives at base $+ i \times$ sizeof(type) where type is the type of the elements in the array. This simple calculation is so convenient that C decides to enable it by making the name of the array be a pointer to the beginning of the array in memory. This means that if we have an array element indexed as `a[3]`, it lives at the address $a + 3 \times \texttt{sizeof}(\texttt{int})$.

However, when we convert this formula to C code, there is a slight adjustment we must make. The appropriate way to rewrite `a[3]` using pointers and offsets is `*(a+3)`. Where did the `sizeof(int)` term go? The answer lies in how C does **pointer arithmetic**.

1.3.1 Pointer Arithmetic

Pointers are allowed three mathematical operations to be performed on them. Addition and subtraction of an integer offset is allowed to support the aforementioned address/offset calculations. The third operation is that two pointers of the same type are allowed to be subtracted from each other in order to determine an offset between the pointers.

In the case of addition or subtraction of an integer offset, C recognizes that if you start with a pointer to a particular type, you will still want a pointer to that same type when you do your arithmetic. If the expression `a+1` added one byte to the pointer, we'd now be pointing into the middle of the integer in memory. What C wants is for `a` to point to the next int, so it automatically scales the offset by the size of the type the pointer points to.

Because addition and subtraction are supported, C also allows the pre- and post-increment and -decrement operators to be applied to pointers. These are still equivalent to the +1 and -1 operations as on integers.

This means that a particularly sadistic person could write the following function:

```
void f(char *a, char *b) {
        while(*a++ = *b++) ;
}
```

This function is one we have already discussed in the course. It is `strcpy()`. The way that it works is that the post-increment allows us to walk one character at a time through both the source and destination arrays. The dereferences turn the pointers into character values. The assignment does the copy of a letter, and yields the right-hand side as the result of the expression. The right-hand side is then evaluated in a boolean context to determine if the loop stops or continues. Since every character is non-zero, the loop continues. The loop terminates when the nul-terminator character is copied, because its value is zero and thus false. The loop needs no body, the entirity of the work is done as side-effects of the loop condition.

1.4 Terms and Definitions

The following terms were introduced or defined in this chapter:

Address The index into memory where a particular value lives.

Dereference To follow the link of a pointer to the location it points to. In C, the dereference operator is *.

Pointer A variable that holds an address.

Pointer Arithmetic Adding or subtracting an offset to a pointer value. In C, this offset is automatically scaled by the type the pointer points to.