In which our hero gains control of machine addresses and learns to manipulate them. But, like a sharp knife, these pointers can cut deeply and quickly if our hero isn’t careful...

14.1 Indirection

A famous computer scientist once said “Any problem in computer science can be solved by another level of indirection.” [S07] Like most generalizations, this one is also not quite right; while many problems are indeed solved by indirection (e.g., how virtual memory is constructed in an operating system, the way files are managed by a file system, the manner in which a flash-based SSD’s flash translation layer is built), not all problems are solved with this general idea. For example, when you write C code, you will introduce a lot of bugs in your code, some quite harmful; no amount of indirection is going to solve this problem, unless the indirection just gets you stop writing C code.

In this chapter, we’ll learn about our first type of indirection in C, a new data type generically referred to as a pointer. We’ve learned in previous chapters about memory addresses – the low-level way we refer to locations of various things in memory. A pointer is just exactly that: a variable that contains a memory address. And, as we learn more about C, we’ll see that pointers are an essential component of the language – to become a C expert, you must also be an expert with pointers. So don’t shy away; dive in and start pointing, and you will soon uncover their true power and utility.

14.2 Definition

So, how do you define a pointer? Given any data type foo, we can define a pointer to a foo as follows:

```c
foo* x;
```
The * symbol in this case indicates that you don’t want \( x \) to be of type `foo`, but rather a pointer to a `foo`. Of course, thus far, we’ve only learned about C integers, and thus the only real definition that would make any sense is:

```c
int* p;
```

Note that in the definition, C is not sensitive to the placement of the * as long as it is between the type (`int`) and the name you have chosen for this variable (\( p \)). As a result, another common way you’ll see a pointer defined is:

```c
int *p;
```

These two statements are, in C, exactly identical. We like to write it in the former manner, as it seems clearer to us what the * is modifying exactly; it is changing the type from an integer to a pointer to an integer. C in fact is fairly insensitive to whitespace; the following would also be legal C (if ugly):

```c
int * p ;
int *p;
```

Here’s an important thing to add to your mental model: defining a pointer to an integer does the same sort of thing that defining an integer does – it allocates space in memory for the pointer, and allows us to refer to that location in memory conveniently through the name of the pointer. Figure 14.1 shows how this might look in memory, with variable \( p \) located at memory address 1016.

For this example, and for many others in this book, we will make the following assumption: that we are using a **32-bit address space**. This means that each address in memory uses 32 bits (4 bytes), as shown in the diagram. Sometimes, those 32 bits are not enough, as we wish to have a program that uses more memory. In those cases, modern machines allow us to construct a program with much bigger addresses (64 bits, or 8 bytes); we’ll discuss this in further detail later.

![Figure 14.1: Pointer \( p \) Located in Memory](image-url)
14.3 Step One: Address-Of With &

Now that we’ve defined a pointer, how can we use it? Well, the first thing you can do is assign it a value. But how do you put a meaningful address into a pointer? We already know about assignment via the = symbol, but what can we assign a pointer to?

To perform a meaningful pointer assignment, we need a new operator. This operator allows a program to obtain the address of an existing variable. It is an & (ampersand), sometimes called address-of, and can be used as follows:

```c
int x = 100;
int* p;
p = &x;
```

In this tiny example, we first define two variables, x and p, the former just a plain old integer, the latter a pointer to an integer. We perform two assignments in the code: we set x to the value 100, and then, we perform an assignment to p, setting its value to be equal to the address of x by using the new address-of operator. Figure 14.2 shows what memory looks like now with these two variables within it. We can now officially say that pointer p points to integer x, or, simply p points to x = yes, p now indirectly refers to another variable.

14.4 Step Two: Indirect Access With *

Assigning a pointer to the address of some other variable isn’t especially useful. We need a way to manipulate the contents of that other variable. Usually the way we set a variable’s value is through assignment, but, as we already saw above, assigning a pointer just changes its value (in the example, setting it to the address of some other variable). We need something more. Can you sense another new operator coming?\(^1\)

---

\(^1\)If so, you have a weird sixth sense, “operator predictivity”, which is rarely useful.
That something more is provided by yet another operator, commonly called the dereference operator (or, sometimes, the indirection operator), and it is represented by the * symbol\(^2\).

Let’s use our new dereference operator to indirectly read or write the value of a variable. Here is a simple example of dereferencing p in an expression to read the value it is pointing to:

```c
int x = 100;
int* p = &x;
x = *p + 10;
```

In this code snippet, we first set the variable x to 100. We then define p, a pointer to an integer, and set it equal to the address of x. Finally, we take the value that p refers to indirectly (100), add 10 to it, and store the result back in x.

We can equally well use p on the left side of an assignment statement, as in this tiny example:

```c
int x = 100;
int* p = &x;
*p = 7;
```

Here we just set the location to which p refers to the value 7. If we then printed out the value of x, that’s what it would print!

### 14.5 Precedence Of & and *

Because we have introduced new operators, we also have to think about their precedence. Figure 14.3 shows how these new operators fit into the precedence table.

As you can see from the table, the precedence of both the address-of operator (\&) and the dereference or indirection operator (*) are fairly high, much like the unary operators + and -.

### 14.6 A Pointer To A Pointer To A ...

Before closing, we’ll note one additional fact (or, perhaps, reiterate it): you can create a pointer to any type, more or less. One type you’ve seen already is a pointer to an integer:

```c
int* x;
```

\(^2\)Note that this is the exact same symbol we use for multiplication! However, the rules of precedence allow us to make sense of expressions without trouble, so don’t worry too much about this.
So, can you create a pointer to this type? In other words, can you create a pointer to a pointer to an integer? The answer, once again, is sure. Here is a code snippet that creates three variables, each one “pointier” than the next:

```c
int x = 100;
int* xp = &x;
int** xpp = &xp;

**xpp = 101;
```

Of course you could also change the value of `xp` indirectly by setting `*xpp` to something. Figure 14.4 shows these three variables, and their values, in memory.

Note that while applying two dereferences in C code is common, we (generally) cannot do the corollary with the address-of operator, `&`. Specifically, writing the expression `&(&x)` isn’t meaningful\(^3\). The variable `x` is allocated somewhere in memory; in C, we generally refer to it as `x`, but really it’s just some bits sitting at an address somewhere. Therefore, it is completely logical to obtain its address. But what would it mean to try to obtain the address of that address? Thus, if you want to create a pointer to a pointer, the most straightforward way to do so is as above, in two steps\(^4\).

So have a cup of tea, and let all of this indirection sink in. It’s weird, cool, and, as it turns out, something you’ll definitely need to

---

\(^3\)We have to include parentheses to even try to express a double address-of operator, otherwise C would think we are expressing a logical-and (`&&`).

\(^4\)C99 does let you do something like this with compound literals, but generally this is probably something to avoid [LS10].
understand to be a C master. And if you don’t find that cool, spend another minute thinking about it. And then one more minute. Finally, stop thinking about it, and go outside. It’s sunny! Why are you wasting your time staring at **xpp? But then come back and puzzle through the details until it all makes sense.

14.7 A Special Value: NULL

Beyond setting a pointer to point to a specific item in memory, there is another reasonable thing we can set a pointer to, and that is the value NULL. Setting a pointer to NULL just means that you are pointing it to absolutely nothing at the moment; we’ll see why this is useful much later, when we build interesting data structures.

Here is some code that defines a pointer p and sets it equal to NULL, in one compact line:

```c
int* p = NULL;
```

The nice aspect of setting a pointer to be NULL is that your code can test such the pointer before using it. For example, you might see code such as this (assuming p is a pointer to an integer):

```c
if (p != NULL)  
    *p = 10;
```

This code only updates the value that p points to if p is not NULL. You’ll even sometimes see a shorthand form of this same code:

```c
if (p)  
    *p = 10;
```

As you might recall, the expression with the parentheses is true if it’s value is not equal to zero. Because we have claimed these two forms are equivalent (i.e., that writing \((p != NULL)\) is the same as writing \((p)\)), you can now conclude that yes, the value of the symbol NULL is actually just zero.
The symbol NULL is used to represent the value of a pointer that does not point to anything. However, in reality, it is just the value zero. Thus, your code can safely assume this fact. Most often, this comes up in writing expressions. For example, in short form, we can write:

```c
while (p) {
    ...
}
```

More verbosely, we would instead write:

```c
while (p != NULL) {
    ...
}
```

### 14.8 Common Mistakes

When you write C code, you will use pointers. And you will make mistakes with pointers. In fact, it’s probably safe to say that pointers are one of the most error prone aspects of C. So learn to be careful with them!

One of the most common mistakes is to dereference a pointer that hasn’t yet set to point to something. In C, you must **never dereference a pointer before it has been initialized**. For example, the following code is buggy:

```c
int* p;
*p = 10;
```

The output of this program is **undefined** – literally anything could happen, because dereferencing an uninitialized pointer is not a defined acceptable behavior in C. Likely, your program will crash, but sometimes, stranger things will happen, like some other value in your program will get overwritten, or even odder, your program will seem to work! If it does work, don’t feel too happy; it might not the next time. Thus, strive to avoid bugs like this one.

Note that assignment of pointers is always legal. For example, we can quite safely write:

```c
int x;
int* p;
p = &x;
```

Doing so, of course, sets the value of the container we call \( p \) to the address of the container we call \( x \). It is never a problem to assign a
pointer to some value; it is always a problem to dereference a pointer that has not been pointed to anything.

A related mistake that is common is to dereference a NULL pointer. In C, **never dereference a null pointer**, because the resulting behavior is also undefined. In most cases, such a program will crash.

A third mistake is to assume that because a pointer is not equal to NULL, that it must indeed be a valid pointer. Unfortunately, this is definitely not true in C – **do not assume a non-NULL pointer is valid**. Because C lets you set variables to pretty much any value you choose, it is easy to put something into a pointer that really shouldn’t be there.

We’ll see many more examples of bugs and mistakes with pointers in future chapters as we create more sophisticated examples of pointer usage. For now, let’s just keep it simple: make sure your pointers point to something before you dereference them; don’t dereference uninitialized or NULL pointers; don’t assume that because a pointer isn’t NULL that must indeed be a valid and correct value.

### 14.9 Summary

We started with a quote about indirection, and how it’s a solution to many problems in Computer Science. But, we didn’t include the full quote, which ends with “... but that usually will create another problem” [S07]. All forms of indirection can yield new problems and issues. As we’ll soon C, pointers solve some very fundamental problems in C, but are not without their costs. Some of those costs are not too bad – performance slowdowns, perhaps – but others, like the many subtle bugs pointers allow you to introduce into your programs, are real and painful still today. The short form of this story: learn to use pointers with great care, and avoid the costs that this power provides.
References

[S07] “Another Level of Indirection”  
D. Spinellis  
Beautiful Code: Leading Programmers Explain How They Think  
O’Reilly and Associates, 2007  
A cool book with tons of lessons from experienced programmers of all kinds.

[LS10] “Initialize pointer to pointer using multiple address operators”  
Answered by Kirill V. Lyadvinsky and Schot  
stackoverflow.com/questions/3810658/initialize-pointer-to-pointer-using-multiple-address-operators-in-c-or-c  
As Schot shows us in this fine stackoverflow answer, you can do some weird things with compound literals in C99. For example, you can even define p as a pointer to a pointer to an integer and set it to the address of the address of 1 with the following line: int ** p = & (int *) & (int *) {1};.  
However, you probably shouldn’t do this.