Lesson 6: Pointers in C

By Alex Allain

Pointers are an extremely powerful programming tool. They can make some things much easier, help improve your program's efficiency, and even allow you to handle unlimited amounts of data. For example, using pointers is one way to have a function modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that you can write programs that can handle nearly unlimited amounts of data on the fly--you don't need to know, when you write the program, how much memory you need. Wow, that's kind of cool. Actually, it's very cool, as we'll see in some of the next tutorials. For now, let's just get a basic handle on what pointers are and how you use them.

What are pointers? Why should you care?

Pointers are aptly name: they "point" to locations in memory. Think of a row of safety deposit boxes of various sizes at a local bank. Each safety deposit box will have a number associated with it so that you can quickly look it up. These numbers are like the memory addresses of variables. A pointer in the world of safety deposit box would simply be anything that stored the number of another safety deposit box. Perhaps you have a rich uncle who stored valuables in his safety deposit box, but decided to put the real location in another, smaller, safety deposit box that only stored a card with the number of the large box with the real jewelry. The safety deposit box with the card would be storing the location of another box; it would be equivalent to a pointer. In the computer, pointers are just variables that store memory addresses, usually the addresses of other variables.

The cool thing is that once you can talk about the address of a variable, you'll then be able to go to that address and retrieve the data stored in it. If you happen to have a huge piece of data that you want to pass into a function, it's a lot easier to pass its location to the function than to copy every element of the data! Moreover, if you need more memory for your program, you can request more memory from the system--how do you get "back" that memory? The system tells you where it is located in memory; that is to say, you get a memory address back. And you need pointers to store the memory address.

A note about terms: the word pointer can refer either to a memory address itself, or to a variable that stores a memory address. Usually, the distinction isn't really that important: if you pass a pointer variable into a function, you're passing the value stored in the pointer--the memory address. When I want to talk about a memory address, I'll refer to it as a memory address; when I want a variable that stores a memory address, I'll call it a pointer. When a variable stores the address of another variable, I'll say that it is "pointing to" that variable.

C Pointer Syntax

Pointers require a bit of new syntax because when you have a pointer, you need the ability to both request the memory location it stores and the value stored at that memory location. Moreover, since pointers are somewhat special, you need to tell the compiler when you declare your pointer variable that the variable is a pointer, and tell the compiler what type of memory it points to.

The pointer declaration looks like this:

```
<variable_type> *<name>;
```

For example, you could declare a pointer that stores the address of an integer with the following syntax:

```
int *points_to_integer;
```

Notice the use of the *. This is the key to declaring a pointer; if you add it directly before the variable name, it will declare the variable to be a pointer. Minor gotcha: if you declare multiple pointers on the same line, you must precede each of them with an asterisk:

```
/* one pointer, one regular int */
int *pointer1, nonpointer1;

/* two pointers */
int *pointer1, *pointer2;
```

As I mentioned, there are two ways to use the pointer to access information: it is possible to have it give the actual address to another variable. To do so, simply use the name of the pointer without the *. However, to access the actual memory location, use the *. The technical name for this doing this is dereferencing the pointer; in essence, you're taking the reference to some

memory address and following it, to retrieve the actual value. It can be tricky to keep track of when you should add the asterisk. Remember that the pointer's natural use is to store a memory address; so when you use the pointer:

```
call_to_function_expecting_memory_address(pointer);
```

then it evaluates to the address. You have to add something extra, the asterisk, in order to retrieve the value stored at the address. You'll probably do that an awful lot. Nevertheless, the pointer itself is supposed to store an address, so when you use the bare pointer, you get that address back.

Pointing to Something: Retrieving an Address

In order to have a pointer actually point to another variable it is necessary to have the memory address of that variable also. To get the memory address of a variable (its location in memory), put the & sign in front of the variable name. This makes it give its address. This is called the address-of operator, because it returns the memory address. Conveniently, both ampersand and address-of start with a; that's a useful way to remember that you use & to get the address of a variable.

For example:

```
#include <stdio.h>

int main()
{
    int x;            /* A normal integer*/
    int *p;           /* A pointer to an integer ("*p" is an integer, so p
                       must be a pointer to an integer) */

    p = &x;           /* Read it, "assign the address of x to p" */
    scanf( "%d", &x );          /* Put a value in x, we could also use p here */
    printf( "%d\n", *p ); /* Note the use of the * to get the value */
    getchar();
}
```

The printf outputs the value stored in x. Why is that? Well, let's look at the code. The integer is called x. A pointer to an integer is then defined as p. Then it stores the memory location of x in pointer by using the address operator (&) to get the address of the variable. Using the ampersand is a bit like looking at the label on the safety deposit box to see its number rather than looking inside the box, to get what it stores. The user then inputs a number that is stored in the variable x; remember, this is the same location that is pointed to by p. In fact, since we use an ampersand to pass the value to scanf, it should be clear that scanf is putting the value in the address pointed to by p. (In fact, scanf works because of pointers!)

The next line then passes *p into printf. *p performs the "dereferencing" operation on p; it looks at the address stored in p, and goes to that address and returns the value. This is akin to looking inside a safety deposit box only to find the number of (and, presumably, the key to) another box, which you then open.

Notice that in the above example, the pointer is initialized to point to a specific memory address before it is used. If this was not the case, it could be pointing to anything. This can lead to extremely unpleasant consequences to the program. For instance, the operating system will probably prevent you from accessing memory that it knows your program doesn't own: this will cause your program to crash. If it let you use the memory, you could mess with the memory of any running program--for instance, if you had a document opened in Word, you could change the text! Fortunately, Windows and other modern operating systems will stop you from accessing that memory and cause your program to crash. To avoid crashing your program, you should always initialize pointers before you use them.

It is also possible to initialize pointers using free memory. This allows dynamic allocation of memory. It is useful for setting up structures such as linked lists or data trees where you don't know exactly how much memory will be needed at compile time, so you have to get memory during the program's execution. We'll look at these structures later, but for now, we'll simply examine how to request memory from and return memory to the operating system.

The function malloc, residing in the stdlib.h header file, is used to initialize pointers with memory from free store (a section of memory available to all programs). malloc works just like any other function call. The argument to malloc is the amount of memory requested (in bytes), and malloc gets a block of memory of that size and then returns a pointer to the block of memory allocated.

Since different variable types have different memory requirements, we need to get a size for the amount of memory malloc should return. So we need to know how to get the size of different variable types. This can be done using the keyword sizeof, which takes an expression and returns its size. For example, sizeof(int) would return the number of bytes required to store an integer.

```
#include <stdlib.h>

int *ptr = malloc( sizeof(int) );
```

This code set ptr to point to a memory address of size int. The memory that is pointed to becomes unavailable to other programs. This means that the careful coder should free this memory at the end of its usage lest the memory be lost to the operating system for the duration of the program (this is often called a memory leak because the program is not keeping track of all of its memory).

Note that it is slightly cleaner to write malloc statements by taking the size of the variable pointed to by using the pointer directly:

```
int *ptr = malloc( sizeof(*ptr) );
```

What's going on here? sizeof(*ptr) will evaluate the size of whatever we would get back from dereferencing ptr; since ptr is a pointer to an int, *ptr would give us an int, so sizeof(*ptr) will return the size of an integer. So why do this? Well, if we later rewrite the declaration of ptr the following, then we would only have to rewrite the first part of it:

```
float *ptr = malloc( sizeof(*ptr) );
```

We don't have to go back and correct the malloc call to use sizeof(float). Since ptr would be pointing to a float, *ptr would be a float, so sizeof(*ptr) would still give the right size!

This becomes even more useful when you end up allocating memory for a variable far after the point you declare it:

```
float *ptr;
/* hundreds of lines of code */
ptr = malloc( sizeof(*ptr) );
```

The free function returns memory to the operating system.

```
free( ptr );
```

After freeing a pointer, it is a good idea to reset it to point to 0. When 0 is assigned to a pointer, the pointer becomes a null pointer, in other words, it points to nothing. By doing this, when you do something foolish with the pointer (it happens a lot, even with experienced programmers), you find out immediately instead of later, when you have done considerable damage.

The concept of the null pointer is frequently used as a way of indicating a problem--for instance, malloc returns 0 when it cannot correctly allocate memory. You want to be sure to handle this correctly--sometimes your operating system might actually run out of memory and give you this value!

Taking Stock of Pointers

Pointers may feel like a very confusing topic at first but I think anyone can come to appreciate and understand them. If you didn't feel like you absorbed everything about them, just take a few deep breaths and re-read the lesson. You shouldn't feel like you've fully grasped every nuance of when and why you need to use pointers, though you should have some idea of some of their basic uses.

Still not getting it? Ask an expert!
Quiz yourself
Previous: Switch/case
Next: Structures
Back to C Tutorial Index

Related
Still having trouble? Try this video