

## Re: "Mastering C Pointers"....

**Source:** [http://coding.derkeiler.com/Archive/C\\_CPP/comp.lang.c/2003-11/0587.html](http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2003-11/0587.html)

---

**From:** Sheldon Simms (*sheldonsimms\_at\_yahoo.com*)

**Date:** 11/04/03

Date: Mon, 03 Nov 2003 23:33:58 -0500

On Tue, 04 Nov 2003 00:59:03 +0000, Alan Connor wrote:

> On Mon, 03 Nov 2003 18:50:09 -0500, Sheldon Simms <sheldonsimms@yahoo.com> wrote:

>>

>>

>> On Mon, 03 Nov 2003 21:29:12 +0000, Alan Connor wrote:

>>

>>> Got really fed up with the Richard fellow. Killfiled his arrogant ass for

>>> a while.

>>

>> You are of course welcome to killfile anyone you want, however you

>> should be aware that Richard is telling you the truth and Roose is

>> not. At least, he is not telling you the whole truth.

>

> And all the people that ARE, are doing nothing but confuse me.

Ok, let me try to tell you something about pointers that is (hopefully) precisely correct, but won't confuse you.

A pointer is a kind of variable that can "point to" some object.

But what does "point to" mean in this case? It means that it is possible to access the object by using the pointer to see where it is.

Let me try to show how it works with an example. For example, let me declare an integer variable:

```
int x; /* an integer named x */
```

Because of the declaration there is an object named x that has a value that is an integer. We can change the value of the object named x and we can retrieve the value of the object named x:

```
x = 3; /* change the value of the integer named x to 3 */  
printf("x == %d\n", x); /* print the value of the integer named x */
```

You might wonder why I'm being verbose and talking about changing "the value of the object named x" instead of "changing x". I'm doing that because it is possible to manipulate the object without calling it x. The integer in memory can have many names, only one of which is "x", although "x" could be considered the "main" or "canonical" name of the object. For example I can also refer to the object with a pointer:

```
int * p; /* a pointer named p to int */
p = &x; /* make p "point to" the object named x */
```

Now px refers to the same object as x and I can manipulate the value of that object by using the name "p".

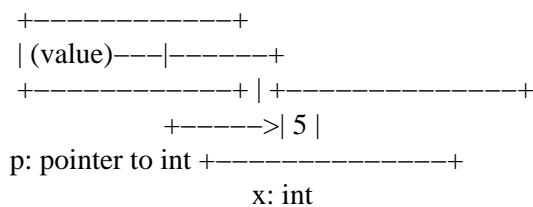
```
*p = 5;
printf("x == %d\n", x);
printf("*p == %d\n", *p);
```

This will produce the output:

```
x == 5
*p == 5
```

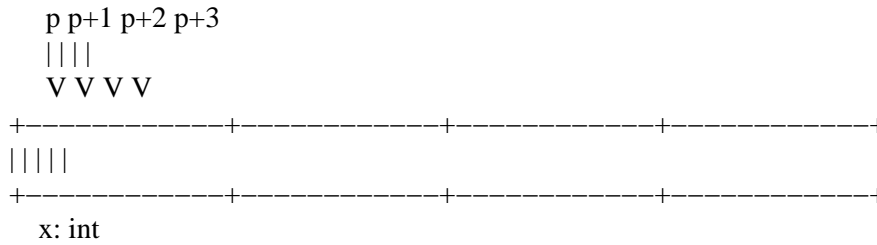
And you can see that the value of the object named x has been altered by using p. the object that p points to is the same object that is named x. This can be said more succinctly "p points to x". However, p is not the same as x. p is also the name of an object. That object has a type (pointer to int), and a value of some kind. The value of p "refers to" the object named x, but how it does that is of no concern to us, all that is important is that we can manipulate that value in certain ways which I will get to in a minute.

To try to make this clear before I talk more about the value of p, here's a diagram that attempts to show the objects named p and x in memory



There is an object named p, and an object named x. The object named x has the value 5 and the object named p has a value that points to x.

Now roose said "pointers are integers". This is wrong, but it's close to the truth. It's close to the truth because pointers act like integers in some ways. For example, we can add an integer to them. If we add an integer to a pointer, we get a pointer that points to the "next thing", where thing is whatever pointer points to. To illustrate this I'll make another diagram.



If the pointer `p` points to the integer `x`, then `p+1` points to the integer after `x`, and `p+2` points to the integer after that, and so on. C allows the programmer to view memory this way. If you have a pointer to some object, then the value of that pointer plus one is a pointer to another object of the same type that comes right after the first one.

However, if you try this:

```

int x;
int * p;
p = &x;
p = p + 1;
printf("the integer after x is %d\n", *p);

```

Your program will probably crash. This is because there actually isn't an object after `x`. We know where the object after `x` would be if it existed, but since it doesn't exist, we can't access it. This is an example of undefined behavior. The C language does not guarantee any particular behavior and, in fact, allows pretty much anything to happen as a result. This problem can be solved by declaring an array of int:

```

int x[8];

```

Now we know that there are 8 integers all lined up right after each other. You may know that you can access these integers by using array notation like this:

```

int i;
int x[8];

for (i = 0; i < 8; ++i)
    printf("x[%d] == %d", i, x[i]);

```

But you can also use pointers to do the same thing, like this:

```

int * p;
int x[8];

p = &x[0];
printf("x[0] == %d\n", *p);
printf("x[1] == %d\n", *(p+1));
...
printf("x[7] == %d\n", *(p+7));

```

And, as you may expect, you can do this more easily with a for-loop:

```
int * p;
int x[8];

p = &x[0];
for (i = 0; i < 8; ++i)
    printf("x[%d] == %d", i, *(p+i));
```

Now we come to a somewhat tricky subject, which is the conversion of arrays to pointers. This isn't really that hard to understand, but it seems to get talked about a lot in comp.lang.c.

In most cases, if you use the name of an array like a pointer, it acts like a pointer. There are sometimes when this is not true, so you have to be quite careful about using this fact until you know when it is true and when it isn't.

To get back to our example, we can change the code like this:

```
int * p;
int x[8];

p = x; /* same as p = &x[0] */
```

We have used x as a pointer and it acts like a pointer. This does *not* copy the entire array anywhere. All it does is make p point at the first element of x. From this point we can continue as before:

```
for (i = 0; i < 8; ++i)
    printf("x[%d] == %d", i, *(p+i));
```

and everything will work the same way.

You might wonder however, if the name of an array sometimes acts like a pointer, can we use it to replace the variable p altogether in this code?

The answer is yes:

```
int x[8];
for (i = 0; i < 8; ++i)
    printf("x[%d] == %d", i, *(x+i)); /* using x as a pointer */
```

In fact, not only can you do this to access the elements of the array, this is **THE ONLY WAY** to do so. This is a very important fact about C. You are probably thinking, hey, we just used x[i] to access elements of the array named x, that's another way! But actually it's not another way. That's because:

Definition:

the meaning of x[i] is \*(x+i)

Because this is true, not only does `x[0]` access the first element of the array named `x`, `0[x]` does too! This may seem strange, but look at the definition above and then consider:

`0[x]` means `*(0+x)` equals `*(x+0)` means `x[0]`

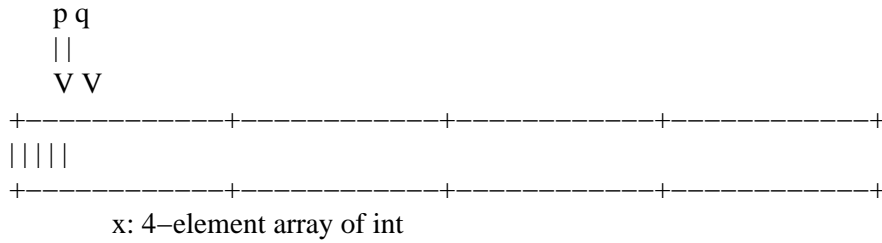
In mathematics `a+b` equals `b+a` when one is adding integers, and the same is true in C when you add a pointer and an integer together. In fact we can rewrite the example loop as follows:

```
int x[8];
for (i = 0; i < 8; ++i)
    printf("x[%d] == %d", i, i[x]); /* perverse, but valid */
```

Now I will tell you one last thing about pointers today: you can subtract one pointer from another, as long as they have the same type and are pointing to different elements of the same array. The result of doing this tells you how many elements ahead of one pointer the other one is. Consider the following:

```
int x[4];
int * p = x; /* using x like a pointer */
int * q = x+2; /* x+2 points at the thing "2 after" the first */
```

These declarations set up the following situation:



You can see that `p` is zero "elements ahead" of itself, and `q` is two "elements ahead" of `p`. You can find this out in a C program like this:

```
int difference = q - p;
printf("q is %d elements ahead of p\n", difference);
```

which will print

q is 2 elements ahead of p

Now I want to remind you, that this only works when both `p` and `q` are pointing to elements of the same array. If the code looked like this:

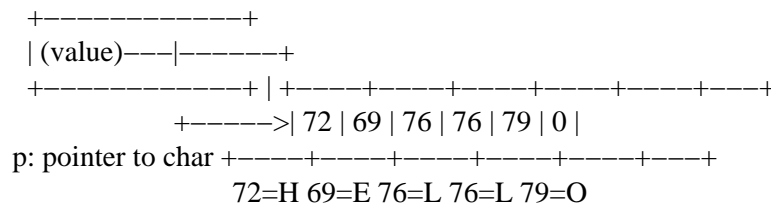
```
int x, y;
int * p = &x;
```

```
int * q = &y;
int difference = q - p;
printf("q is %d elements ahead of p\n", difference);
```

Then there's no guaranteeing what will happen. p and q both point to objects of type int, but they are not elements of a common array. The behavior of this program is undefined. We may find out how many integer objects can fit in the space between wherever the compiler puts the object named x and wherever the compiler puts the object named y, but the C compiler is not required to do this, it could format our hard disk instead. — harsh, to be sure, but allowed by the C standard.

To finish up, I will present a function using some of these concepts. The function will take one argument, a string, and will return the length of the string. A string in C is an array of char containing values representing characters. The array has at least as many elements as characters in the string, plus one. The element of the array after the last character in the string has the value 0, to mark the end of the string. A character with the value 0 is called a NUL. In C, you can just write it like a normal zero (i.e, 0), but many people choose to show that it is a character by writing it inside ' quotes like this: '\0'. The backslash is necessary to make this a character with value zero, as opposed to the character which is used to print a zero on your screen, which is written '0'.

A string is usually manipulated in C by using a pointer to the first element of the array. A diagram should make this clear.



Here is a string containing the text "HELLO". Each object of type char in the array has a numeric value. Each number corresponds to some character. The mapping from number to character can be done in many different ways, and that fact can be very important when writing programs for international audiences. The mapping used above is a very common mapping though, used in the US and Europe most of the time these days.

In any case, we use a pointer to char to access our string, which is an array of elements of type char. The function to be presented will not declare the array itself, it will assume that the array has already been declared, but that does not change the fact that the array *must* be declared somewhere. We can only use a pointer that points to some object that really exists in memory, as I said way up above.

The function to be presented doesn't care what the values in the elements of the array are, it will just look for the value 0, which is the last

element of the string. So here's the function, at first, line-by-line.

```

unsigned int
/* the length of a string can't be negative, so return unsigned char. */

string_length (const char * string)
/* we will not change the values in the string, so declare the pointer
   as a pointer to const(ant) char. when this function is called,
   string will point at the first element of a char array that contains
   a string */
{
    const char * index;
    /* we will use this pointer to find the zero */

    index = string;
    /* string points at the first element, now index does too */

    while (*index != '\0')
    /* while index isn't pointing at the terminating NUL */
    {
        index = index + 1;
        /* make index point at the next character */
    }

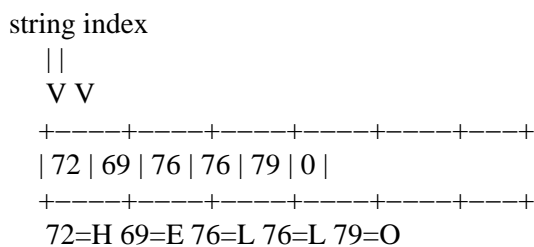
    /* when execution reaches this point, index points at the
       character with value zero, that is, at the NUL. That
       means that index is pointing at the end of the string. */

    return index - string;
    /* compute how many characters ahead of string index is, and
       return that value as the length of the string. */
}

```

The way the length of the string is computed means that "the length of the string" means "the number of non-zero characters in the string". You should be able to follow the code, now. To convince you that it is correct, the last diagrams...

At the beginning:



\*index is 72, not 0, so add one to index:

```

string index
  ||
  V V
+-----+-----+-----+-----+-----+-----+
| 72 | 69 | 76 | 76 | 79 | 0 |
+-----+-----+-----+-----+-----+
72=H 69=E 76=L 76=L 79=O

```

Keep going like that until \*index == 0:

```

string index
  ||
  V V
+-----+-----+-----+-----+-----+
| 72 | 69 | 76 | 76 | 79 | 0 |
+-----+-----+-----+-----+-----+
72=H 69=E 76=L 76=L 79=O

```

Now return the number of elements that index is ahead of string:

```

string index
  ||
  V 5 4 3 2 1 V
+-----+-----+-----+-----+-----+
| 72 | 69 | 76 | 76 | 79 | 0 |
+-----+-----+-----+-----+-----+
72=H 69=E 76=L 76=L 79=O

```

index – string == 5, which is the number of letters in the string "HELLO".

Here the code again, without comments, along with a main function that uses it:

```

#include <stdio.h>

unsigned int string_length (const char * string)
{
    const char * index;
    index = string;

    while (*index != '\0')
    {
        index = index + 1;
    }

    return index – string;
}

int main (void)
{

```

comp.lang.c: Re: "Mastering C Pointers"....

```
unsigned int length;  
length = string_length("hello");  
printf("the length is %u\n", length);  
return 0;  
}
```

-Sheldon