Everything you need to know about pointers in C

Version 1.2.2.

Copyright 2005–2007 Peter Hosey.

This work is licensed under a Creative Commons Attribution 2.5 License.

This document comes with a companion example program, available as one fileor as multiple files (zipped).

Style used in this document

This is regular text. This is a `variable`, some `code`, and some `sample output`.

```
This is a line of code.
This is a comment.
This is also a comment.
```

```
This is output you'd see on your screen.
```

Table of contents

Definition of a pointer

A pointer is a memory address.

(Mmm, short paragraphs.)

Starting off

Say you declare a variable named `foo`.

```
int foo;
```

This variable occupies some memory. On a PowerPC, it occupies four bytes of memory (because an `int` is four bytes wide).

Now let's declare another variable.

```
int *foo_ptr = &foo;
```

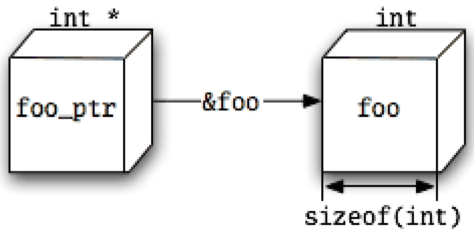
`foo_ptr` is declared as a pointer to `int`. We have initialised it to point to `foo`.

As I said, `foo` occupies some memory. Its location in memory is called its address. `&foo` is the address of `foo` (which is why `&` is called the ‘address-of operator’).

Think of every variable as a box. `foo` is a box that is `sizeof(int)` bytes in size. The location of this box is its address. When you access the address, you actually access the contents of the box it points to.

This is true of all variables, regardless of type. In fact, grammatically speaking, there is no such thing as a ‘pointer variable’: all variables are the same. There are, however, variables with different types. `foo`’s type is `int`. `foo_ptr`’s type is `int *`. (Thus, ‘pointer variable’ really means ‘variable of a pointer type’.)

The point of that is that the pointer is not the variable! The pointer to `foo` is the contents of `foo_ptr`. You could put a different pointer in the `foo_ptr` box, and the box would still be `foo_ptr`. But it would no longer point to `foo`.



The pointer has a type, too, by the way. Its type is `int`. Thus it is an ‘`int` pointer’ (a pointer to `int`). An `int **`’s type is `int *` (it points to a pointer to `int`). The use of pointers to pointers is called multiple indirection. More on that in a bit.

Interlude: Declaration syntax

The obvious way to declare two pointer variables in a single statement is:

```
int* ptr_a, ptr_b;
```

- If the type of a variable containing a pointer to `int` is `int *`,
- and a single statement can declare multiple variables of the same type by simply providing a comma-separated list (`ptr_a, ptr_b`),
- then you can declare multiple `int`-pointer variables by simply giving the `int`-pointer type (`int *`) followed by a comma-separated list of names to use for the variables (`ptr_a, ptr_b`).

Given this, what is the type of `ptr_b`? `int *`, right?

bzzt Wrong!

The type of `ptr_b` is `int`. It is **not** a pointer.

C’s declaration syntax ignores the pointer asterisks when carrying a type over to multiple declarations. If you split the declaration of `ptr_a` and `ptr_b` into multiple statements, you get this:

```
int *ptr_a;
int  ptr_b;
```

Think of it as assigning each variable a base type (`int`), plus a level of indirection, indicated by the number of asterisks (`ptr_b`'s is zero; `ptr_a`'s is one).

It's possible to do the single-line declaration in a clear way. This is the immediate improvement:

```
int *ptr_a, ptr_b;
```

Notice that the asterisk has moved. It is now right next to the word `ptr_a`. A subtle implication of association.

It's even clearer to put the non-pointer variables first:

```
int ptr_b, *ptr_a;
```

The absolute clearest is to keep every declaration on its own line, but that can take up a lot of vertical space. Just use your own judgment.

Finally, I should point out that you can do this just fine:

```
int *ptr_a, *ptr_b;
```

There's nothing wrong with it.

Incidentally, C allows zero or more levels of parentheses around the variable name and asterisk:

```
int ((not_a_pointer)), (*ptr_a), (((*ptr_b)));
```

This is not useful for anything, except to declare function pointers (described later).

Assignment and pointers

Now, how do you assign an `int` to this pointer? This solution might be obvious:

```
foo_ptr = 42;
```

It is also wrong.

Any direct assignment to a pointer variable will change the address in the variable, not the value at that address. In this example, the new value of `foo_ptr`(that is, the new 'pointer' in that variable) is 42. But we don't know that this points to anything, so it probably doesn't. Trying to access this address will probably result in a segmentation violation (read: crash).

(Incidentally, compilers usually warn when you try to assign an `int` to a pointer variable. `gcc` will say '`warning: initialization makes pointer from integer without a cast`'.)

So how do you access the value at a pointer? You must dereference it.

Dereferencing

```
int bar = *foo_ptr;
```

In this statement, the dereference operator (prefix `*`, not to be confused with the multiplication operator) looks up the value that exists at an address. (On the PowerPC, this called a 'load' operation.)

It's also possible to write to a dereference expression (the C way of saying this: a dereference expression is an lvalue, meaning that it can appear on the left side of an assignment):

```
*foo_ptr = 42; Sets foo to 42
```

(On the PowerPC, this is called a 'store' operation.)

Interlude: Arrays

Here's a declaration of a three-`int` array:

```
int array[] = { 45, 67, 89 };
```

Note that we use the `[]` notation because we are declaring an array. `int *array`would be illegal here; the compiler would not accept us assigning the `{ 45, 67, 89 }` initialiser to it.

This variable, `array`, is an extra-big box: three `int`s' worth of storage.

But here's a little secret: you can never refer to this array again.

'What?' you say. 'But the compiler lets me do that! Watch!'

```
printf("%p\n", array); Prints some hexadecimal string like 0x12307734
```

Ah, but what does `%p` mean?

It means 'pointer'.

When you use the name of an array in your code, you actually use a pointer to its first element (in C terms, `&array[0]`). This is called 'decaying': the array 'decays' to a pointer. Any usage of `array` is equivalent to if `array` had been declared as a pointer (with the exception that `array` is not an lvalue: you can't assign to it or increment or decrement it, like you can with a real pointer variable).

So when you passed `array` to `printf`, you really passed a pointer to its first element, because the array decays to a pointer.

Decaying is an implicit &; `array == &array == &array[0]`. In English, these expressions read '`array`', 'pointer to `array`', and 'pointer to the first element of `array`' (the index operator, `[]`, has higher precedence than the address-of operator). But in C, all three expressions mean the same thing.

Pointer arithmetic (or: why 1 == 4)

Say we want to print out all three elements of `array`.

```
int *array_ptr = array;
printf(" first element: %i\n", *(array_ptr++));
printf("second element: %i\n", *(array_ptr++));
printf(" third element: %i\n", *array_ptr);
```

```
 first element: 45
second element: 67
 third element: 89
```

In case you're not familiar with the `++` operator: it adds 1 to a variable, the same as `variable += 1` (remember that because we used the postfix expression `array_ptr++`, rather than the prefix expression `++array_ptr`, the expression evaluated to the value of `array_ptr` from before it was incremented rather than after).

But what did we do with it here?

Well, the type of a pointer matters. The type of the pointer here is `int`. When you add to or subtract from a pointer, the amount by which you do that is multiplied by the size of the type of the pointer. In the case of our three increments, each 1 that you added was multiplied by `sizeof(int)`.

By the way, though `sizeof(void)` is illegal, `void` pointers are incremented or decremented by 1 byte.

In case you're wondering about `1 == 4`: Remember that earlier, I mentioned that `int`s are four bytes on a PowerPC. So on a PowerPC, adding 1 to or subtracting 1 from an `int` pointer changes it by four bytes. Hence, `1 == 4`. (Programmer humour.)

Indexing

```
printf("%i\n", array[0]);
```

OK… what just happened?

This happened:

```
45
```

Well, you probably figured that. But what does this have to do with pointers?

This is another one of those secrets of C. The index operator (e.g. `array[0]`) has nothing to do with arrays.

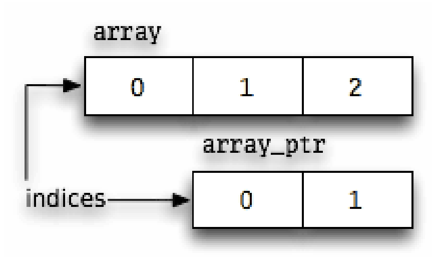
Oh, sure, that's its most common usage. But remember that arrays decay to pointers. That's a pointer you passed to that operator, not an array.

As evidence, I submit:

```
int array[] = { 45, 67, 89 };
int *array_ptr = &array[1];
printf("%i\n", array_ptr[1]);
```

```
89
```

That one might bend the brain a little. Here's a diagram:



`array` points to the first element of the array; `array_ptr` is set to `&array[1]`, so it points to the second element of the array. So `array_ptr[1]` is equivalent to `array[2]` (`array_ptr` starts at the second element of the array, so the second element of `array_ptr` is the third element of the array).

Also, you might notice that because the first element is `sizeof(int)` bytes wide (being an `int`), the second element is `sizeof(int)` bytes forward of the start of the array. You are correct: `array[1]` is equivalent to `*(array + 1)`. (Remember that the number added to or subtracted from a pointer is multiplied by the size of the pointer's type, so that '`1`' adds `sizeof(int)` bytes to the pointer value.)

Interlude: Structures and unions

Two of the more interesting kinds of types in C are structures and unions. You create a structure type with the `struct` keyword, and a union type with the `union`keyword.

The exact definitions of these types are beyond the scope of this article. Suffice to say that a declaration of a struct or union looks like this:

```
struct foo {
        size_t size;
        char name[64];
        int answer_to_ultimate_question;
        unsigned shoe_size;
};
```

Each of those declarations inside the block is called a member. Unions have members too, but they're used differently. Accessing a member looks like this:

```
struct foo my_foo;
my_foo.size = sizeof(struct foo);
```

The expression `my_foo.size` accesses the member `size` of `my_foo`.

So what do you do if you have a pointer to a structure?

```
One way to do it
(*foo_ptr).size = new_size;
```

But there is a better way, specifically for this purpose: the pointer-to-member operator.

```
Yummy
foo_ptr->size = new_size;
```

Unfortunately, it doesn’t look as good with multiple indirection.

```
Icky
(*foo_ptr_ptr)->size = new_size; One way
(**foo_ptr_ptr).size = new_size; or another
```

Rant: Pascal does this much better. Its dereference operator is a postfix ^:

```
Yummy
foo_ptr_ptr^^.size := new_size;
```

(But putting aside this complaint, C is a much better language.)

Multiple indirection

I want to explain multiple indirection a bit further.

Consider the following code:

```
int    a =  3;
int   *b = &a;
int  **c = &b;
int ***d = &c;
```

Here are how the values of these pointers equate to each other:

```
  *d ==   c; Dereferencing an (int ***) once gets you an (int **) (3 - 1 = 2)

 **d ==  *c ==  b; Dereferencing an (int ***) twice, or an (int **) once, gets
you an (int *) (3 - 2 = 1; 2 - 1 = 1)

***d == **c == *b == a == 3; Dereferencing an (int ***) thrice, or an (int **)
twice, or an (int *) once, gets you an int (3 - 3 = 0; 2 - 2 = 0; 1 - 1 =
0)
```

Thus, the `&` operator can be thought of as adding asterisks (increasing pointer level, as I call it), and the `*` and `[]` operators as removing asterisks (decreasing pointer level).

Pointers and `const`

The `const` keyword is used a bit differently when pointers are involved. These two declarations are not equivalent:

```
const int *ptr_a;
int const *ptr_b;
```

In the first example, the `int` (i.e. `**ptr_a`) is `const`; you cannot do `**ptr_a = 42`. In the second example, the pointer itself is `const`; you can change `**ptr_b` just fine, but you cannot change (using pointer arithmetic, e.g. `ptr_b++`) the pointer itself.

Function pointers

Note: The syntax for all of this seems a bit exotic. It is. It confuses a lot of people, even C wizards. Bear with me.

It's possible to take the address of a function, too. And like arrays, functions decay to pointers when their names are used. So if you wanted the address of, say, `strcpy`, you could say either `strcpy` or `&strcpy`. (`&strcpy[0]` won't work for obvious reasons.)

When you call a function, you use an operator called the function call operator. The function call operator takes a function pointer on its left side.

In this example, we pass `dst` and `src` as the arguments on the interior, and `strcpy`as the function (that is, the function pointer) to be called:

```
enum { str_length = 18U }; Remember the NUL terminator!
char src[str_length] = "This is a string.", dst[str_length];

strcpy(dst, src); The function call operator in action (notice the function
pointer on the left side).
```

There's a special syntax for declaring variables whose type is a function pointer.

```
char *strcpy(char *dst, const char *src); An ordinary function declaration,
for reference
char *(*strcpy_ptr)(char *dst, const char *src); Pointer to strcpy-like funct

strcpy_ptr =  strcpy;
strcpy_ptr = &strcpy; This works too
strcpy_ptr = &strcpy[0]; But not this
```

Note the parentheses around `*strcpy_ptr` in the above declaration. These separate the asterisk indicating return type (`char *`) from the asterisk indicating the pointer level of the variable (`*strcpy_ptr` — one level, pointer to function).

Also, just like in a regular function declaration, the parameter names are optional:

```
char *(*strcpy_ptr_noparams)(char *, const char *) = strcpy_ptr; Parameter
names removed — still the same type
```

The type of the pointer to `strcpy` is `char *(*)(char *, const char *)`; you may notice that this is the declaration from above, minus the variable name. You would use this in a cast. For example:

```
strcpy_ptr = (char *(*)(char *dst, const char *src))my_strcpy;
```

As you might expect, a pointer to a pointer to a function has two asterisks inside of the parentheses:

```
char *(**strcpy_ptr_ptr)(char *, const char *) = &strcpy_ptr;
```

We can have an array of function-pointers:

```
char *(*strcpies[3])(char *, const char *) = { strcpy, strcpy, strcpy };
char *(*strcpies[])(char *, const char *) = { strcpy, strcpy, strcpy }; Array
size is optional, same as ever

strcpies[0](dst, src);
```

Here's a pathological declaration, taken from the C99 standard. '[This declaration] declares a function `f` with no parameters returning an `int`, a function `fip` with no parameter specification returning a pointer to an `int`, and a pointer `pfi` to a function with no parameter specification returning an `int`.' (6.7.5.3[16])

```
int f(void), *fip(), (*pfi)();
```

In other words, the above is equivalent to the following three statements:

```
int f(void);
int *fip(); Function returning int pointer
int (*pfi)(); Pointer to function returning int
```

But if you thought that was mind-bending, brace yourself…

A function pointer can even be the return value of a function. This part is reallymind-bending, so stretch your brain a bit so as not to risk injury.

In order to explain this, I'm going to summarise all the declaration syntax you've learned so far. First, declaring a pointer variable:

```
char *ptr;
```

This declaration tells us the pointer type (`char`), pointer level (`*`), and variable name (`ptr`). And the latter two can go into parentheses:

```
char (*ptr);
```

What happens if we replace the variable name in the first declaration with a name followed by a set of parameters?

```
char *strcpy(char *dst, const char *src);
```

Huh. A function declaration.

But we also removed the `*` indicating pointer level — remember that the `*` in this function declaration is part of the return type of the function. So if we add the pointer-level asterisk back (using the parentheses):

```
char *(*strcpy_ptr)(char *dst, const char *src);
```

A function pointer variable!

But wait a minute. If this is a variable, and the first declaration was also a variable, can we not replace the variable name in THIS declaration with a name and a set of parameters?

YES WE CAN! And the result is the declaration of a function that returns a function pointer:

```
char *(*get_strcpy_ptr(void))(char *dst, const char *src);
```

Remember that the type of a pointer to a function taking no arguments and returning `int` is `int (*)(void)`. So the type returned by this function is `char *(*)(char *, const char *)` (with, again, the inner `*` indicating a pointer, and the outer `*` being part of the return type of the pointed-to function). You may remember that that is also the type of `strcpy_ptr`.

So this function, which is called with no parameters, returns a pointer to a `strcpy`-like function:

```
strcpy_ptr = get_strcpy_ptr();
```

Because function pointer syntax is so mind-bending, most developers use `typedefs` to abstract them:

```
typedef char *(*strcpy_funcptr)(char *, const char *);

strcpy_funcptr strcpy_ptr = strcpy;
strcpy_funcptr get_strcpy_ptr(void);
```

Strings (and why there is no such thing)

There is no string type in C.

Now you have two questions:

1. Why do I keep seeing references to ‘C strings’ everywhere if there is no string type?
2. What does this have to do with pointers?

The truth is, the concept of a ‘C string’ is imaginary (except for string literals). There is no string type. C strings are really just arrays of characters:

```
char str[] = "I am the Walrus";
```

This array is 16 bytes in length: 15 characters for "I am the Walrus", plus a NUL (byte value 0) terminator. In other words, `str[15]` (the last element) is 0. This is how the end of the ‘string’ is signalled.

This idiom is the extent to which C has a string type. But that’s all it is: an idiom. Except that it is supported by:

- the aforementioned string literal syntax
- the string library

The functions in string.h are for string manipulation. But how can that be, if there is no string type?

Why, they work on pointers.

Here’s one possible implementation of the simple function `strlen`, which returns the length of a string (not including the NUL terminator):

```
size_t strlen(const char *str) { Note the pointer syntax here
        size_t len = 0U;
        while(*(str++)) ++len;
        return len;
}
```

Note the use of pointer arithmetic and dereferencing. That’s because, despite the function’s name, there is no ‘string’ here; there is merely a pointer to at least one character, the last one being 0.

Here’s another possible implementation:

```
size_t strlen(const char *str) {
        size_t i;
        for(i = 0U; str[i]; ++i);
        When the loop exits, i is the length of the string
        return i;
}
```

That one uses indexing. Which, as we found out earlier, uses a pointer (not an array, and definitely not a string).