Pointers made simple(r)

An introduction and unfuddling of pointers in C

Dominic van Berkel

1 Intro

Pointers are a tremendously important topic in lower-level languages, prominently C and C++. They give the programmer access to heap memory[1] and give you almost full control over the memory you allocate, which in turn allows you to optimize memory management for your application rather than relying on black-box storage libraries and garbage collection.

If you're coming from a higher-level language such as Python or C# this may sound a little like an old coal-powered steam engine that just stops working when you're not careful with the fuel. There's a *reason* new languages handle all that stuff for us, right? And you'd be right: in a lot of situations today there's no need for manual memory management. Even the new 2011 C++ specification makes life easier in this regard with automatic pointers and an extensive library of storage classes. Modern computers have enormous amounts of memory and processing power, so for a lot of applications it's not even very interesting to make optimal use of anything.

All that said, there are still good reasons to learn and perhaps even worry about memory allocation. First of all: someone has to write all those fancy frameworks that make life easier. Second: having some idea about how memory works on a lower level can be tremendously helpful in understanding why some code works better than some other code. And third: in a lot of contexts – high-performance on a supercomputer or in the confined spaces of embedded systems – keeping track of what your memory is doing is still immensely relevant.

Purpose of this document Pointers are a tremendously important topic in lower-level languages, but they can also be a big hurdle for a lot of people. Students, teachers and professionals alike can find them very confusing, and even those who use them daily may struggle to explain some things clearly. It's not even very complicated: words just tend to get in the way. I will try to explain things from scratch, pointing out and circumnavigating potential pitfalls on the way.

This article assumes that you're using the C programming language, as that's where you're most likely to run into these things. A lot of time is spent explaining and untangling C syntax. Much of it also applies to C++, and conceptually it should transfer very well to other low-level environments.

2 The what, the how, and the ugly

2.1 What is a pointer?

First and foremost, a pointer *is a variable*. Don't let anything convince you otherwise: a pointer is a variable, just like an `int` or `char`. Just like any other variable a pointer is a little area on

[1] Program memory is typically divided in a stack of fixed size and a heap that can expand as necessary. In C, normal ("automatic") variables go on the stack, while anything allocated with malloc() and friends goes on the heap.

the computer's memory that contains a value. What's special about pointers is that their value is the address of another variable. How is that useful?

Consider the following function:

Listing 1: Set large numbers to zero, sans pointers

```c
int clear_if_large(int num) {
    if (num > 10)
        return 0;
    else
        return num;
}
```

It returns 0 when the provided integer is larger than 10, otherwise it returns the argument unchanged. Not tremendously useful, but it'll do for an example. When this function is called the value of the argument is copied to the stack where the function body can access it. The entire integer is duplicated to make this happen.

pass by value

That doesn't sound so bad, does it? An integer is only a handful of bytes, my computer can easily spare that. That's true – but only for such small datatypes. Imagine if an image editor or an audio tool had to copy all the source data around for every little operation – your computer would slow down to a crawl just keeping track of where everything is. Wouldn't it be nice if we could just tell the function where to look for the data? Enter pointers:

Listing 2: Set large numbers to zero, with pointers

```c
void clear_if_large(int * num) {
    if (*num > 10)
        *num = 0;
}
```

What's that? It doesn't return anything! True, but that's because it doesn't have to. Thanks to the use of pointers the integer isn't copied over, but the address of the integer is. The function then has a reference to the integer, knows where to look and can modify the variable without having to copy all of it over. Again, not very exciting when you're dealing with a single integer, but a 100 MB image you'd rather keep in one place.

pass by reference

In other languages Many higher-level languages don't have (Python, Javascript) or very rarely need (C#) pointers. To be clear, that doesn't mean that they're constantly copying enormous blobs of data around – the behaviour enabled by pointers in C is handled automatically there. Often that means that all objects are passed by reference in any function call and are only copied when you really want to.

Terminology Giving a function the address of a variable rather than the value of the variable is called *pass-by-reference*, to be contrasted with *pass-by-value* where the value is copied. This is very much a real distinction, but it's also misleading: in pass-by-reference a value is still copied and passed: the value of the *address* of the variable. Remember: a pointer is just a variable that happens to contain the address of another one.

This isn't the only thing you can do with pointers by a long shot. Some more *thrilling* possibilities are described in section 3.

2.2 Using pointers

Now we know what pointers can do for us. How do we make them do it? One character you'll frequently see when dealing with pointers is `*`. Three of the four lines in Listing 2 have one of those. What may surprise you is that it does two different things.

Variable declaration In line 1 the function arguments are declared. Here the asterisk belongs with the variable type. What's more, it's *part of* the variable type: the function takes one argument called `num` of type `int *`, which you can read as "pointer to an integer".

Indirection operator *or* **dereference operator** Two names for the same thing. The variable `num` is declared as a pointer to an integer, which means that the value of `num` is an address. But we don't care about addresses, we want integers! The dereference operator *dereferences* a pointer: it gets you access to the variable at the address contained by the pointer; access to the variable the pointer points at. In lines 2 and 3 that's what `*` does: it gives you access to read and modify the value at the address stored in `num`.

It's possible that much of the confusion about pointers stems from this difference: these two meanings of the asterisk are very similar yet so very different, and few texts spend any time making this explicit.

The important thing to remember is that in `int * num` – be it a function argument or a variable declaration – the asterisk is part of `num`'s variable type, `int *`, while in `*num = 0` it dereferences the pointer `num`.

`num` is of type `int *`

From a different perspective: in the argument list it doesn't even make sense to think of it as a dereference operator. The dereference operator makes an integer out of a pointer to an integer – `num` is a pointer to int, `*num` is an integer. `clear_if_large(int * num)` is a function that takes one argument of type `int *`, and `*num` has nothing to do with that.

`*num` is of type `int`

Optional reading ahead Unfortunately it's not entirely true that in `int * foo` the asterisk ties to `int` to make a compound type of `int *`. While it *is* a convenient way to think about the notation, part of the C syntax disagrees. It only really becomes apparent when you declare multiple variables on a single line:

```
char * one, two, three;
```

If the asterisk fully tied to the char you'd expect this to result in three variables of type `char *`, so three pointers to `char`. In reality, this code will leave you with one pointer to `char` and two bog-standard `char`s. If you want three pointers it becomes:

```
char * one, * two, * three;
```

The C Programming Language[1, p. 94] describes the declaration syntax as "a mnemonic [that] mimics the syntax of expressions in which the variable might appear". This is a little strange as `int * foo` can be assigned an address, while `*foo` can only be assigned an integer value. Additionally, even the C language specification[2, p. 43] refers to `float *` as having the type "pointer to `float`".

Whatever the reasons are for this situation and whether you think it makes sense or not, this is how it is. Generally it's easiest to think of `long * foo` as declaring a variable `foo` of type `long *`, but in a handful of situations that can't be relied on.

2.3 Creating pointers

Pointers are variables, so they can be created the same way as other variables. Let's try it out:

Listing 3: Creating a pointer

```
#include <stdio.h>
```

```c

void main() {
    int * foo;  // Create a variable "foo" of type "int *"
    *foo = 4;   // Assign 4 to the address it points at
    printf("%d\n", *foo);
}
```

Looks great! Compile, run:

```
dominic@nienna:~/coding> ./a.out
Segmentation fault
```

Ouch. That didn't work. "Segmentation fault" means that the program tried to access memory it shouldn't, such as other applications' memory space or the address 0. What happened here is that while the pointer was *created*, it wasn't made to *point* somewhere. In other words: room was made for the pointer-to-int, but not for an int to point to.

2.3.1 Pointing to existing variables

There are two main ways to get hold of an address that a pointer can store: getting the address of an existing variable, or allocating new space in memory. The former looks like this:

Listing 4: Pointing a pointer

```c
#include <stdio.h>

void main() {
    int bar = 45;
    int * foo;   // Create a variable "foo" of type "int *"
    foo = &bar;  // Set foo (the pointer to int) to the address of bar
    printf("%d\n", *foo);
}
```

Let's see. Compile, run:

```
dominic@nienna:~/coding> ./a.out
45
```

Much better, but what happened? This snippet introduces a new operator: the *address-of operator* `&`. As mentioned before, a pointer is really just a variable that contains the address of another variable. This operator allows you to get the address of another variable and stick it into a pointer. In this particular case it takes the address of `bar` and stores it in `foo`. As with other variables, you don't have to split declaration ("this variable exists") and assignment ("its value is..") over two lines:

Listing 5: Pointing a pointer, short version

```c
#include <stdio.h>

void main() {
    int bar = 45;
    int * foo = &bar;
    printf("%d\n", *foo);
}
```

See the difference? What may confuse here is that it might look like you're assigning `&bar` to an integer called `*foo`. That is not the case! The longer version assigned `&bar` to `foo`, not to `*foo`. It's the same here: `&bar` is assigned to `foo`, which happens to be a variable of type `int *`.

2.3.2 Pointing to the heap

As mentioned in the introduction, normally declared variables (e.g. `int num = 5`, or function arguments) are typically stored on the stack. The stack is a fixed-size chunk of memory allotted to every program and thread. Often this will be cached in registers on the processor itself so access to the stack is as fast as it gets. Space is limited, though, so it's easy to run out of room when you're processing a lot of information or doing a lot of recursive calls.

The heap, on the other hand, might be described as "the rest of the memory". The heap is another section of memory available to the program that can grow and shrink as needed, giving you access to as much memory as the operating system will allow. To get data on the heap you have to explicitly allocate space. When you're done with the allocated space you should deallocate it, otherwise it won't be available until your program has finished running – more on that in subsection 2.5.

Listing 6: Allocating heap memory

```c
#include <stdio.h>
#include <stdlib.h>

void main() {
    int * heapptr = malloc(sizeof(int));
    *heapptr = 42;
    printf("%d\n", *heapptr);
}
```

When you want a pointer to the heap there's no other variable to get the address of, so you'll have to ask the system to make some room for you. The standard library `<stdlib.h>` defines – among many others – the function `void * malloc(size_t size)`, which returns a pointer to some available space of the specified size. You may not have seen the type `size_t` before, but don't worry: it's just an integer type of a size that's appropriate for the system you're compiling for. Usually you'll get a `size` argument by using the `sizeof` operator, which can be applied to variables (integers, chars, but also structures) as well as to literal type identifiers (`int`, `long`) and yields the size of the type in bytes.

Space allocated with `malloc` is never initialized. Unless you put something there the value will be whatever happened to be at that memory location, which is usually not very useful. There's another library function `void * calloc(size_t nobj, size_t size)` that allocates space for `nobj` objects of size `size` and also initializes the space to zero bytes.

Potential errors But what happens when `malloc` can't find enough space in memory? Rest assured: nothing catches fire, and *trying* to allocate too much memory doesn't make anything crash either. What does make things crash is trying to use memory that isn't available to you – remember the `Segmentation fault`. When `malloc` or `calloc` can't find enough space to fulfill your request they return `NULL`, which is why it's always a good idea to check for that before trying to use the results of a memory allocation.

2.4 In context

So now you've seen a lot of isolated examples and if you're anything like me you're probably wondering: how does it all fit together? For that, we'll return to the earlier example of `clear_if_large(int *)`.

Listing 7: Set large numbers to zero, with pointers

```c
void clear_if_large(int * num) {
    if (*num > 10)
```

```c
        *num = 0;
}
```

Before you read on Try to figure out what the function call would look like. The function asks for one pointer to an integer, or `int *`.

Ready? Here are three slightly different ways to go about it.

Listing 8: Providing pointer arguments, take 1

```c
#include <stdio.h>

void main() {
    int highnum = 286;        // Create an ordinary integer
    clear_if_large(&highnum); // Pass its address
    printf("%d\n", highnum);
}
```

Listing 9: Providing pointer arguments, take 2

```c
#include <stdio.h>

void main() {
    int highnum = 286;        // Create an ordinary integer
    int * highptr = &highnum; // Create a pointer to it
    clear_if_large(highptr);  // Pass the pointer
    printf("%d\n", *highptr);
}
```

Listing 10: Providing pointer arguments, take 3

```c
#include <stdio.h>
#include <stdlib.h>

void main() {
    int * highptr = malloc(sizeof(int)); // Create a pointer, allocate space
    *highptr = 286;          // Fill the space with data
    clear_if_large(highptr); // Pass the pointer to that space
    printf("%d\n", *highptr);
}
```

Three different ways to pass a pointer to a function, but they're really all the same: in the end, every single one of them passes the function the address of some memory space that contains an integer. In the first and second, the address is derived from another variable, in the second and third the address is stored in a pointer.

Pay close attention to how `*` is used in all three cases as well as how it's not used. Remember: our function takes a pointer-to-int, while `printf` needs an integer for `%d`.

2.5 Cleaning up

When you allocate memory on the heap, that memory is marked as unavailable to anything else until it's deallocated. There are two ways to deallocate memory: by ending the program that's using it, or by explicitly telling the system that you're not going to need it anymore. A lot of things that only run for a few seconds or minutes can rely on the former, but if a program stays on for a while or uses a lot of memory it's a good idea to keep track of what you do and don't need anymore.

Freeing memory is easy: <stdlib.h> provides a function void free(void *ptr) that you can pass a pointer to previously allocated memory which will then be freed up. The argument does have to match a pointer that was originally returned by malloc and friends – you can't just pass it any address in a block of allocated space, and you can't free stack memory. So what does it look like?

Listing 11: Cleaning up heap memory

```
 1 #include <stdlib.h>
 2 
 3 void main() {
 4     int * lots = malloc(4000 * sizeof(int));
 5     /* do things with all that space */
 6     free(lots);
 7     lots = NULL;
 8     float * otherlots = malloc(2000 * sizeof(float));
 9     /* do things with all those floats */
10 }
```

Because all memory is automatically released when a program ends it's not necessary to free any leftover memory just before the end. It's still a good habit to do so, as you might want to add more code at the end later on.

Notice the NULL assignment in line 5? It's not necessary, but it's frequently considered good practice to set pointers that don't go anywhere useful right now to NULL so you're less likely to try and access garbage memory. Additionally, free does nothing when presented with a NULL pointer, so this way it doesn't matter if you accidentally try and free the pointer's target twice.

2.5.1 What's the problem?

As you can see, freeing allocated memory isn't very hard. The hard part is keeping track of all the memory you've allocated. It's beyond the scope of this document to explain the various tricks, but keep two things in mind:

Make sure you never lose the last reference When you have a function that returns a pointer to newly allocated memory and you pass that directly to another function which stores, prints or otherwise processes the data at that location, how do you free it afterwards? By using an intermediate variable you'll always keep a reference that you can deallocate at your leisure.

Keep track of responsibility When you're writing a library or just a small collection of functions, think about and document clearly which part "owns" the pointer to a given block of memory. If you know who's responsible for deallocating the memory you don't (shouldn't) have to worry about it.

3 Complicating matters

Don't worry, this part isn't actually going to be hard. Up until now, we've only dealt with pointers to single integers or chars, but that's pretty boring – there's so much more! For instance, Listing 11 allocated 4000 * sizeof(int) for a single int *, how does that make sense? And since a pointer is just a variable, can you in fact have a pointer to a pointer? Sure!

Nothing of this is going to be *hard*. Much of it is a little more complex than previous sections, but it can still be explained with everything you already know.

3.1 Variable-size arrays

This one's pretty essential. Up until the C99 standard you had to specify the size of arrays at compiletime, meaning that you couldn't base it on a function argument or any other runtime information. C99 allows for this, but the other solution – that you'll still see plenty of – is to allocate a block of space with pointers and `malloc`.

Listing 12: Using pointers for variable-size arrays

```c
#include <stdio.h>
#include <stdlib.h>

void main() {
    int * thelist = calloc(4000, sizeof(int));
    int i;
    for (i = 0; i < 4000; i++) {
        thelist[i] = 2 * i;
    }
    printf("%d\n", thelist[476]);
}
```

Compile, run:

```
dominic@nienna:~/coding> ./a.out
952
```

Note that once it's created, you can use the same notation as for arrays to get at the values in the allocated block - you don't even need an asterisk to dereference. This is because `thelist[i]` is equivalent to `*(thelist + i)`, where adding `i` to a pointer means adding `i * sizeof(int)` to the address. In other words, it takes the address stored in `thelist`, increments it by enough to get to the `i`th integer value and dereferences that address. Incidentally, this is why arrays start at zero in C: the first item has the same address as the start of the list.

3.2 `char` pointers

One common use for pointers in C is to store strings. Where higher-level languages have a String class or similar that can store text of pretty much any length, in C that has to be stored in character arrays. Typically a function will receive a pointer to the start of that array and continues reading from there until it encounters a nullbyte `'\0'`.

Listing 13: Character arrays

```c
#include <stdio.h>

void main() {
    char * onestring = "Hi there!";
    printf("%s \n", onestring);
    printf("%s \n", &onestring[3]);
}
```

Compile, run:

```
dominic@nienna:~/coding> ./a.out
Hi there!
there!
```

The printf function doesn't know about strings, only about character pointers – so there's nothing wrong with passing it the address of the third character in the string, as seen in line 5. It'll just keep reading from there to the nullbyte that marks the end of the string. In this case that nullbyte is added automatically in line 3, but in other situations you might have to place it yourself or risk a segfault.

For more information on character arrays, look up the <string.h> library for related functions.

3.3 Pointers to pointers

In my experience this is where a lot of people get confused when they're learning C. Keep one thing in mind: a pointer is just a variable. You can create a pointer to it which then contains the address of the first pointer as its value.

Listing 14: Pointers to pointers

```
#include <stdio.h>

void main() {
    int foo = 23;
    int * bar = &foo;
    int ** quux = &bar;
    printf("quux at %u, value %u\n", &quux, quux);
    printf("bar  at %u, value %u\n", &bar, bar);
    printf("foo  at %u, value %d\n", &foo, foo);
}
```

Compile, run:

```
dominic@nienna:~/coding/pointersimple> ./a.out
quux at 3219582196, value 3219582200
bar  at 3219582200, value 3219582204
foo  at 3219582204, value 23
```

Your results will most likely be very different, but the pattern is the same: the value of quux is the address of bar, and the value of bar is the address of foo.

The notation is the same as always - a pointer to type looks like type * foo, so a pointer to int * becomes int * * foo which is typically written as int ** foo. If it helps, you can substitute another name for int *:

Listing 15: Pointers to pointers, substituted

```
void main() {
    typedef int * intptr; // Define intptr to mean "int *"
    int foo = 23;
    intptr bar = &foo;
    intptr * quux = &bar; // A pointer to intptr is really just "int **"
[...]
```

You may encounter functions that deal with pointers to pointers, for instance when you want to get the address of something. That may befuddle a little, but always remember that a pointer is just another value - if it helps, substitute and rethink.

Listing 16: Passing the address of a pointer

```
#include <stdio.h>

void make_string(char ** out) {
```

```c
    char * ret = "Here's a string!";
    *out = ret;
}

void main() {
    char * onestring;
    make_string(&onestring); // Pass the address of onestring
                             // so the function can write a new value there
    printf("%s\n", onestring);
}
```

In make_string, out is a variable of type char **. That means that **out is a char, and *out is a char *. Because make_string receives the address of a pointer, it can change the address stored *in* that pointer. Again, if this doesn't immediately make sense, replace char * with another name:

Listing 17: Passing the address of a pointer, substituted

```c
#include <stdio.h>

typedef char * string;  // Define string to mean "char *"

void make_string(string * out) {
    string ret = "Here's a string!";
    *out = ret;
}

void main() {
    string onestring;
    make_string(&onestring); // Pass the address of onestring
                             // so the function can write a new value there
    printf("%s\n", onestring);
}
```

And now it looks just like you're passing any other variable – because that's all it is! Compare this to Listing 8 in subsection 2.4 where a pointer to int was passed.

3.4 Function pointers

Pointers don't only point to data: they can also point to functions. In C, this is the main way to pass functions around as arguments. A good example of why this is useful is the qsort function (defined in <stdlib.h>), which has the following signature:

```c
void qsort(void * base, size_t n, size_t size,
    int (*cmp)(const void *, const void *))
```

Explaining the details is a little beyond the scope of this article, but the last argument to qsort is a pointer to a function that takes two pointers. This argument allows you to tell qsort how to sort the array: cmp is expected to compare the values pointed at by its arguments, return 1 if the first one should be placed after the second (typically because it's larger) and otherwise return 0.

4 Tidbits

4.1 About

I first thought of writing this article after trying (and failing) to explain an assignment to a classmate. I'd breezed through the assignment myself, but while I could *use* pointers and related concepts it turned out I didn't grok[2] it yet. Writing it out like this would be an opportunity to clear up my own misconceptions and mental blocks. It did take a lecturer getting confused by his own explanation for me to actually start writing.

The first version of this article is heavily influenced by my own challenges with the subject matter, but I hope that it can be helpful for any who are new to or confused by programming with pointers.

The most recent version of this document is available at: `https://github.com/barometz/pointersmadesimple`.

We Value Your Input If you have a suggestion or think there's something wrong with the article, be it a factual error, an omission or a spelling mistake, please contact me. I wrote this to learn as much as to teach, and teaching without learning is a dead end anyway. If there's something right with the article, please contact me immediately. Send your threats and love letters to <dominic@baudvine.net>.

Further reading By now, *The C Programming Language*[1] (often referred to as K&R) is 34 years old. Even so it remains one of the most recommended books for learning the basics of C programming and then some. The second, most recent edition was released in 1988, so don't expect to see anything from the 2011 standard. What you *can* expect is a thorough introduction to the basics of the language, including a lot of exercises and even some reimplementations of library functions so you really know what's going on under the hood.

4.2 Legalish

Please don't use any information from this document to program nuclear bombs and/or shelters. Any of it might be wrong and under no circumstances do I guarantee that it will not get you killed or work as advertised at all.

Copying This work is licensed under the Creative Commons Attribution- NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-sa/3.0/`.

If you would like to do something not explicitly allowed by the license, please contact me. I'm easy about these things.

References

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 2nd Edition, 1988.

[2] ISO/IEC 9899:201x, *Programming languages – C*, draft N1570, april 2011. Available from `http://www.iso-9899.info/wiki/The_Standard`.

[2] to grok, v.: To fully and completely understand something in all its details and intricacies – `http://en.wiktionary.org/wiki/grok`