A C Primer

Dan Rohr

December 9, 2014

Preface

There's a few things I feel I should say before I get into this. First among them is that I am not, by any means, a C expert. I'm not someone who could sit here all day and rattle off all the little subtle differences between K&R C vs C89 vs C99 vs C11. What I am is a student who has gone, with little formal instruction, from knowing nothing about C and having experience coding in high-level languages like Java and Python, to knowing enough about C to be an effective C programmer for a large student engineering project. Why does that give me the credentials to write something like this? Well, because I'm designing this for someone in a similar position to where I was: a curious student looking to get started with a new kind of programming.

That said, this is not a beginning programming guide. I expect anyone reading this to at least have some coding experience, ideally in another typed language (like, say, Java), though experience in untyped languages like Python and MATLAB is also ok (it'll just be another new paradigm to get used to). It's also worth noting that most of what I'll be discussing here is going to be stuff that carries well to other low-level languages. So if you're coming here as an expert C++ or Fortran programmer, a lot of this will be redundant.

Another important thing to note about this is that it is not a textbook, and certainly not a complete description of the C programming language. There's about a billion of those already on the market, many of which are written by language designers and implementers who know way more than I do.

So if this is an incomplete guide, how are you supposed to learn C? Here's something they don't teach you in school: being a programmer means being an autodidact. The fact is, outside of maybe your first programming course, nobody is ever going to sit you down and hold your hand through all the trivial syntatic details of a programming language.

So think of this text as more like "guided self-teaching" than a full C course.

What I ask at this point is that you come to this with an open mind. Your first instinct is going to be to try to relate everything you see to something in Java, or whatever language you're most comfortable with. There are plenty times this will not be inappropriate (since C heavily influenced pretty much every widely-used programming language that exists), but there are plenty of times where it will be a very bad idea. Also, forget everything you already know about C (except what's from a textbook or some other serious source). If you read this front to back, you should know what you need to know to move forward, and eventually get a formidable knowledge of this language.

Exercises

This stuff is mostly environment setup, not actual "exercises." The true coding work is going to come in the exercises for chapter 3.

1. Install a Unix-like operating system on your machine, either natively, or in a virtual machine. If you're running OSX, congratulations, you're done. Otherwise, you'll want to find a Linux distribution. I demand this for two reasons. 1. Microsoft C/C++ is a whole different animal, and I'm not qualified to teach it. 2. The use of a Unix-like interface is an **essential** skill for any programmer. It might even be good to install Linux in a VM on top of OSX, just to get exposure to it.

   If you're installing Linux, I reccommend using a virtual machine at first, since it gets you up and running faster, though you'll eventually want a native install. I reccommend Oracle's Virtualbox for the virtualization. It's free, and fairly good. As for distribution choice, Ubuntu is a common choice for beginners, though Fedora is just as good. Just make sure to choose one with a graphical installer (stay away from Arch and Gentoo for now). There are a million guides to installing in VMs online, read one. Just make sure your virtual hard drive is not dynamically sized. It saves disk space, but slows down the VM a **ton**. If your VM is still too slow, most Linux distributions have more lightweight variants, for example XUbuntu and LUbuntu are Ubuntu variants that use more lightweight GUIs.

2. install git on this machine and (if you haven't already) run the following command in a terminal

   ```
   git clone https://github.com/dr107/CPrimer.git
   ```

   I'll be referring to code there throughout this text. To install git on Ubuntu, do

   ```
   sudo apt-get install git
   ```

3. Make sure you have the GNU C compiler (`gcc`) installed, as well as GNU Make. Both are required to compile the example code. `clang`, which is installed with the OSX developer tools, should work as an alternative to `gcc`, but I can't guarantee it. On Ubuntu, running

   ```
   sudo apt-get install build-essential
   ```

   should get you everything you need. It's a series of developer tool packages that should include everything you need to compile C code.

4. Install a text editor of your choice. For now, do not use an IDE. You won't need the features of one for now; it'll just be confusing. Stick to a plain text editor like `emacs` or `vim`. Both have long learning curves, but both are extremely powerful tools. If you don't care about efficiency or being a bamf, I guess Sublime Text is ok too.

5. Read the documentation for the printf function on `http://www.cplusplus.com/reference/cstdio/printf/`. It might look a little weird, but just accept the syntax for now. There's examples at the bottom of the page.

   By the way, that website is a great resource. It's named after C++, but since C++ is (almost) a superset of C, it has documentation for the C standard libraries.

Chapter 1

The Basics

The spirit of C

Every programming language (at any one people actually use) has a reasoning behind its existence. The ANSI C rationale gives the following five tenets as the "Spirit of C"

- Trust the programmer
- Don't prevent the programmer from doing what needs to be done
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

These tenets will begin to make more sense as you do more C programming, but let's have a look at a couple of them now. The first two concern you, the programmer. C gives you freedom that you probably aren't used to. That freedom gives you a lot of power, which you can use to write great, creative programs. It also means you have the freedom to write garbage, nigh-undebuggable code.

The C specification

Before we start, it's worth talking about just what C is in the first place. The C language is defined by a specification, which is a big fat document that rigorously defines what a valid C program is, and what it must do. This standard has evolved over the years since C was first developed in

1972. That's what we mean when we say c99, for example. It refers to the C standard developed in 1999. The code in the examples will be compiled under the C11 standard, since it's the most recent.

The importance of the specification to you, as a programmer, is that the C specification is different from those of high-level programming languages. Throughout this document, and other discussions of C, you'll see words like "unspecified" and "undefined." This refers to places where the C specification simply does not say what should happen.

For example, we'll talk about arrays later on, but one common source of undefined behavor in C is indexing an array outside of its bounds. The Java specification guarantees that an exception is thrown at runtime when this happens. No C specification guarantees anything of the sort. If you do it, anything might happen. Your operating system's kernel might throw a segmentation fault, or you might just read a value you don't want. Maybe a dragon flies out of your CD tray; C makes no guarantees.

Why is this so? Why wouldn't we want that exception? Well, C is a language that exists for breakneck speed, and the less the spec specifies, the more freedom the language implementer has to make optimizations. For example, if the spec guaranteed an exception for array indexing, the language would have to keep track of things like the size of the array, and be constantly checking for overflow every time an element is accessed. That's a slowdown. Instead, the onus is on **you**, the programmer, to ensure that this does not happen.

Going back to the ANSI rationale, "A standard is a treaty between implementor and programmer."

You might want to go and read the introduction to the ANSI rationale. It's not long and pretty informative.

Data types

Enough philosophical bullshit, it's time to talk about code. C is a stongly typed language, and its primitive data types should be familiar from other languages. You have stuff like int, float, double, and char. Note the abscence of strings. We'll talk about those later. What's a good idea to talk about now, is char.

If I asked you what a char is, you'd probably say "a character," and you wouldn't be wrong. The way you usually think of a char in a low-level language is more along the lines of "one byte of data." Huh? Well there's this thing called the ASCII character encoding. It maps the numbers 0 through 255, inclusive, to various characters, including all latin letters

and arabic numbers, as well as some other symbols and signals. Why 0 through 255? Well, 256 is 2^8, which means one ASCII character can fit into one byte of data.

Another interesting thing to note is that there is not explicit booean type. Instead, you use `int` to represent a boolean value, with 0 being `false` and anything else being `true`, for example:

```
if (0) {
  //never executes
}

if (1) {
  //always executes
}
```

The usual comparison operators (==, <, <=, etc) return ints for this reason.

Functions

For the purposes of this text, there's not much to say about functions in C, at least until we get to linking. First of all, the order you write functions in matters. If f_2 calls f_1, f_1 should be written first. What if they depend on each other? We'll get to that in chapter 5.

A bigger issue (and source of bugs) is that arguments to functions in C are passed by copy. For example, have a look at this C code

```
void mutate_int(int x)
{
        x=5;
}
```

What does this function do? Absolutely nothing. In fact, if you called this function in your code, an optimizing compiler would probably just skip over it completely. This is because when you call this function on a certain integer, the value of that integer is copied into the variable x, which is local to the `mutate_int` function. When you mutate that value, you're only manipulating the local variable, not whatever variable you put in. This is pretty intuitive with integers, but it's also true when you're passing in more complex data, like structs and pointers. We'll see more of that in the sections on those data types.

Pointers

There's at least one kind of data that you're probably not used to at all. The pointer. This is going to be your first exposure to memory; I hope you're excited.

What the fuck is a pointer?

Memory in operating systems is byte-addressed. That is, each byte of memory available to the system is labelled with a number, which can be used to access the byte of data sitting at that location. A pointer is a variable that stores the memory address of some data, rather than the data itself.

Take a minute to wrap your head around that. A pointer is just a number, that signifies the location of some piece of data in memory. There really isn't any mystery to it, but they are a fundamental aspect of C programming, and will pervade both this text, and your C career.

Getting pointers

There's two new operators you need to know about to work with pointers: `*` and `&`. The former is called the dereference operator. When used on a pointer, it fetches whatever data is at the location addressed by it. Since C knows the size of data types, it knows how much data to fetch. The latter does the opposite; it gets the memory address of some data.

The syntax for declaring a variable as a pointer to data of a certain type, is to put a `*` character between the type name and the variable name. Stylistically, it's more common to put it next to the variable name. The fact that this is the same character as is used for the dereference operator can be a bit confusing at first. Try to keep them straight.

There's one pointer type that might look a little weird at first: `void*`. This type indicates a a pointer that points somewhere, but the data there is an unknown type. Pointers with this type cannot be dereferenced. After all, how much data should be fetched? One byte? Four? Twenty? Without knowing the size of the data being pointed to, it's impossible to know.

Navigate to the code examples, and go into the CH1 directory. Open up `deref.c`.[1]

[1] When you open this, you might ask, "What the fuck is that line with the hashtag?" For now, ignore it. It uses black magic to give us `printf`. This black magic will be

Look at the three functions. From the above, it should be clear what the first two do, provided you understand `printf`.[2]. The third function is just the function from above that does nothing.

Compile the code by running `make deref` from the chapter one directory. This is where it's crucial you have `gcc` and `make` installed.[3] Did you notice the compiler warning you get from our flawed `mutate_int` function? The compiler is trying to help you find bugs in your code, even though it's valid C code. **In production code, compiler warnings should be taken very, very seriously.** Most of the time, getting a compiler warning means there's something you're not thinking about, or that there's code you aren't using that should be removed anyway.

Run it by doing `./deref.o`. Look at the output. Is it about what you expected?

Pointer arithmetic

I said in earlier that a pointer is just a number. You often do arithmetic on numbers. Why not pointers? Now, one major thing to mention that's a little different from arithmetic with numbers is that you don't really want to do

```
int *ptr1=...
int *ptr2=...
int *ptr3=ptr1 + ptr2;
```

I can't possibly conceive of a situation where that would make sense. I'm talking about something more like this

```
int *ptr=...
ptr++;
```

What did that do? Well we know what ++ does to integers: it increments them by one. Does it do the same thing to pointers? Not exactly. It actually increments the pointer by as many bytes as the size of the

explained in detail in chapter 4. Also, try to ignore the content of `main`. I tried to avoid putting this kind of "please ignore for now" stuff in this, but it seems unavoidable in programming texts.

[2]There's one piece of syntax here worth discusing. What's with the `void` keyword doing as the "argument" to the functions with no arguments? Believe it or not, in C, a function declared with empty parentheses indicates the function can take any number of arguments. By putting the `void`, we are ensuring it takes no arguments, and any calling code that tries to call with arguments will not compile, which is what we want.

[3]for more information about `make` see the appendix

data it points to. So if your implementation has 4-byte integers (as is common in most modern machines), it will actually increase the pointer's numerical value by four.

If you instead had

```
char *ptr=...
ptr++;
```

What would happen? Well, the size of char is 1, so that would increment the pointer by one. What if you had a pointer to some data type, but wanted to increment it by one. Have a look at the following:[4]

Line 4 of this block is your first exposure to typecasting in C. Typecasting does different things at different times, and we'll be coming back to it throughout this text. When you cast one pointer type to another, you are essentially asking the compiler to keep the same numberical value of the pointer, but treat it as pointing to a different datatype, so the meaning of things like arithmetic and dereferencing will be different for the casted data.

```
int x=0xdeadbeef;
int *ptr=&x;
int i;
char *chrptr=(char*) ptr;

for(i=0; i<sizeof(int); i++) {
  printf("%x", *chrptr);
  chrptr++;
}
printf("\n");
```

This code prints out the bytes of the integer x inddividually in hexadecimal, followed by a newline. Stare at the code for a while. Maybe read more about `printf` if that's unclear.

Once you've convinced yourself that code is correct, open up `arith.c` in the chapter one directory. The first function is some playing around with incrementing pointers. What do you think will happen when we increment `void*`? No reason you should know. What do you think it will print? The second function is the code from above ... sort of. [5]

[4] At this point, you should read up on the `sizeof` operator

[5] It turns out you need to tell the compiler to interpret the char as unsigned in order to get this to work as expected. It has to do with the bit representation of negative numbers. If you're interested, google around about it.

Compile with `make arith` and run with `./arith.o`. I'm sure the output of the first function makes sense. What about the second? Perhaps not. See the third exercise.

Hanging pointers

As you might be familiar with from other programming languages, variables have **scope** in c. Informally, this means you can't refer to a variable outside the block of code where you declared it. If you don't really know what this means, google around. In high-level languages, that's the end of the story, but in C, you have to worry about the pointers to these variables.

The pointer to a local variable should not be dereferenced outside that variable's scope. That should really make sense. The variable is gone, why should a pointer to it be meaningful? Open `hanging.c`. It's should be pretty clear what's happening there. Run `make hanging && ./hanging.o`. Did you notice the compiler warning? The compiler really tries to help with this stuff.

The explanation behind this behavior has to do with the execution stack. Read more about it at

`http://en.wikipedia.org/wiki/Call_stack#Structure`

When a function is done executing, its stack frame is popped, and its memory is free to be used for other things, so you can't predict what will be there. We'll see much more about both hanging pointers and call stacks in our discussion of arrays. Just bear in mind that as far as the language is concerned, the behavior is unspecified, so just don't do it.

Exercises

These still aren't really coding exercises. At this point, you still don't have most of the machinery you need to do true C coding.

1. This is a bit of a research question. Why does `main` in each example have return type `int` and not `void`, like in Java? Try removing the `return` from one of the examples and recompile. It compiles, doesn't it? What does it return? Type `echo $?` after the program exits to find out.

   For extra credit, change COPTS to say `-std=c89` instead of `-std=c11` in the makefile for that example, recompile and rerun. Now what is returned? This is one of those things that's different accross C

standards. The exit value of a `main` with no return is defined to be 0 in C11, but in C89, it's unspecified. Extra extra extra credit if you can guess why the returned value is what it is... (hint: what does `printf` return?

2. Go into the `Examples/CH1` directory. First of all, do a `make ex1` and note that it compiles without any warnings. Check out they type signature of the function. Did you read the footnote from before? Anyways, alter `mutate_int` so that it actually sets the value of an integer to 5. *Hint:* What type should the input be?

3. Another research project. What is endianness? Based on the output of `arith.o`, what is the endianness of your current setup? Stuff like this is why working with bits and bytes gets really, really confusing.

Chapter 2

Memory

malloc and free

All of the memory we've been using up to this point has been *stack-allocated*. Any program you run gets a certain amount of memory at startup to put its execution stack. Local variables are stored in this memory, and it is managed automatically. But what does "memory management" mean anyway?

The answer is complicated, and in fact is a big part of any operating systems course, but for the purposes of what we're doing here, think of the OS as the source of all memory. You borrow it to do your computing, then you have to give it back. Your program handles allocating and returning its stack memory, since local variables follow all kinds of rules about scope and such. But what if you want memory to be visible throughout your program? Then you have to go to the heap.

The heap is a part of the memory that the OS manages. You ask the OS for a certain amount of memory, and the OS goes out and finds it. When you're done, you have to tell the OS that that memory is no longer in use. The C standard library has two functions for these processes. `malloc` and `free`.

malloc

Go read the documentation for `malloc`. It's not long. All you do is ask the OS for some number of bytes of memory, and it goes out and finds that many contiguous bytes, and returns a pointer to the beginning. However, there's a few subtleties that you should be aware of.

Say you want three integers' worth of memory. You might say "ok, an `int` is four bytes long" and write the following:

```
int *buf=malloc(12);
```

That might work. But whether or not an `int` is four bytes is dependent on your machine. This is why we have the `sizeof` operator.

```
int *buf=malloc(3*sizeof(int));
```

By the way, even though `malloc` has return type `void*`, there's no need to cast. `void*` gets automatically promoted to whatever pointer type you assign it to. So now you have three ints worth of room, here's how you would assign them (feel free to refresh yourself on how pointer arithmetic works):

```
int *buf=malloc(3*sizeof(int));
*(buf)=1;
*(buf+1)=2;
*(buf+2)=3;
```

A couple of things to keep track of here. First of all, what is the initial value of data in the memory allocated? Until you set it, this is unspecified. So if you read it, you'll get whatever data happens to be sitting there. If you want to allocate memory and set it all to 0, `calloc` is also in the standard libraries, and does just that. Another thing to remember is that you, as the programmer, are responsible for keeping track of how many bytes are being pointed to, and not trying to read beyond that, which is also unspecified.

Finally, memory allocated by `malloc` can be accessed from anywhere in your program, so feel free to pass around pointers that it returns.

All of this will get way more clear soon.

free

Go read the documentation. Pretty simple, right? In principle, yes it is. But there's a lot to keep track of. If you free a pointer, you can't read from any of that memory any more. If you do, guess what? It's unspecified. So why free anything at all? You'll find if you never call `free`, your program ostensibly runs exactly as expected.

If you don't call `free`, the heap memory has no way of realizing that your program is done with its memory, or even that it has exited. So a user of your program will find their computer's memory getting sucked

up with no apparent explanation. This is called a memory leak. See the appendix section on `valgrind` for some information on tools to help with this.

arrays

About fucking time. I mean, seriously, this is like page 12 or something. Why did it take this long? Well, in order to understand C arrays, you really need to understand pointers. That's because an array in C is just a pointer to the beginning of a block of memory you've allocated. I'm going to repeat that, because a lot of people have trouble with it. **All arrays in C are just pointers.** There's two ways to allocate arrays in C. Say you wanted to allocate an array of three integers. You could do

```
int arr[3];
```

Or you could do

```
int *arr=malloc(3*sizeof(int));
```

At this point, you should be thinking "Wait, wasn't part of the spririt of C to just have one way to do an operation?" That's right. Even though the end result of these two operations is an array of three integers, they are two very different animals.

The first declaration allocates an array on the stack. This has the same limitations of any stack-allocated data: limited scope. But it has one further limitation. It's size must be known at compile-time. So you can't write this function. The advantage is that, as we discussed in the previous section, the memory is managed automatically, so no need to `free`.

```
void makearray(int n)
{
  // This would cause a compiler error.
  int arr[n];
  ...
}
```

Heap-allocating your array using `malloc` fixes these problems. The memory can be accessed from anywhere, and this function

```
void makearray(int n)
{
  // this is ok
```

```
    int *arr=malloc(n*sizeof(int));
    ...
}
```

is totally fine, since the memory is managed by the OS at runtime. And you can even return `arr` if you feel like it. There's two disadvantages. First of all, you have to `free` it at some point. And that's hard to keep track of. The function above allocates memory. Say you call that function without realizing it's doing that, so you don't think to `free` it. That's a memory leak. The other disadvantage is that `malloc` is slow. As simple as it seems, it's actually very, very complicated to write `malloc` efficiently, and even then, it's not something you want to do all the time. In performance-critical applications, when a pointer is no longer needed, instead of getting freed, it'll get put into a pool of unused pointers, which the application will re-use at some point in the future. Essentially, the application does the bookkeeping usually done by the kernel. Technically, it's up to you how you want to allocate. However, the fixed size in stack-allocation is usually a dealbreaker, so you better get used to `malloc`

I still haven't mentioned how you set and get stuff in and out of arrays. Above, I wrote this code.

```
int *buf=malloc(3*sizeof(int));
*(buf)=1;
*(buf+1)=2;
*(buf+2)=3;
```

This would work, but the syntax is infuriating. That's why C gives you the following syntax, which will probably look familiar.

```
arr[i]
```

is equivalent to

```
*(arr+i)
```

That was a hell of a build-up to just index an array, huh? But now you know it top to bottom. You know how arrays get allocated, what it means to get and set information in them, and why it's not ok to go over their bounds.

By the way, now you know why C arrays are indexed at 0. An array pointer points to the beginning of its data and arr[0] is equivalent to *(arr+0).

strings

Another thing it took a damn long time to get to. Like I said in the data types section, C does not have strings as a primitive type. Instead, we have arrays of `char`. That's all a string is, in any language if you look under the hood, but C forces you to face it head-on. There's really only one not obvious thing about strings: null termination. The ASCII character with code 0 is called the null character. A well-formed C string is an array with some characters, ending with the null character. So the following code

```
char *str="abc";
```

Is exactly equivalent to

```
char str[4];
str[0]='a';
str[1]='b';
str[2]='c';
str[3]=(char) 0;
```

Other than that, you work with C strings precisely as you would any other array. For an example, open up `CH2/stringstuff.c`. I've implemented a string equality tester, and a couple of test cases for it. There's a couple of functions in there you'll implement in the exercises. Have a look around, compile with `make string` and run with `./str.o`.

Exercises

1. Read the documentation for `calloc` and `realloc`. These are both extremely useful.

2. Now read the docs for `memmove`. Say you had an array of four elements of type `float` and a buffer with room for four values of type `double`. What does the resulting array look like? Don't worry too much about what the output would exactly be.

3. In `stringstuff.c` refactor `streq` to use a `for` loop. Figure out the syntax to initialize and increment both characters in the top of the loop. I wrote it that way in the first place, but I thought the syntax might be too obtuse.

4. Write a function (whatever file you want) that heap-allocates an array of 100 numbers and iterates over it to fill it. Then write

another function that takes the array and a number of elements, that prints that many elements and then frees the array.

5. Write and unit test all the functions marked with a TODO in `stringstuff.c`.

The next chapter's exercises will give you much, much more practice with arrays.

Chapter 3

Structs

Why structs?

Say you're working on a C project to do numerical calculations for some kind of a physics or engineering project. Physics and engineering calculations usually use vectors. Say you were tasked with writing a three-dimensional vector computation library for the rest of your team to use. Bear in mind that your team members have varying knowledge of C, so you want to make this as easy as possible for them. With what you currently know, there's a couple of things you could do. You could use three variables of type `float`, or you could use arrays of three floats. From a design perspective, both leave a lot to be desired. So say you wanted to write a function that computes the dot product of two vectors.

The former is chaos. The psychologists say that people can only keep track of about seven things in their heads at a time, so once you have three vectors, it'll be hard for a programmer using your code to use it properly. Also, C functions can only return one value, so you would have to do some pointer hacking to return a new vector. The array option is slightly better, but means that any functions you write would allow an arbitrary-size buffer, so it would be easy for someone to mistakenly pass in an array of say, five floats.

What if you could somehow duct-tape a bunch of variables together if they have some logical common purpose?

Defining and using structs

Turns out you can, and it's called a struct. You could write the following:

```
struct vec3d {
```

```
  float x;
  float y;
  float z;
}; //note the semicolon
```

This defines a type `struct vec3d`. The things in the curly braces are called fields. The way you read and write values of fields should be familiar from various object oriented languages.

```
struct vec3d v;
v.x = 2.5;
printf("v.x=%f\n", v.x); //prints 2.5
```

Your first instinct when you see this is probably to start thinking about classes. This is not all wrong, but it's dangerous. A class has *state* and *behavior*, fields and methods. A struct only has state. The other thing is that structs are missing is any kind of inheritance, and any of the associated ideas like abstract classes and interfaces, since none of these ideas make sense when you don't have member functions.[1].

An aside: typedef

Notice above that we just had our first exposure to defining a new type in C. There actually is another way that we haven't needed up until now, the `typedef`. In our hypothetical vector library, we could either use `float` or `double` to represent a real number. What if we choose `float` initially, but realize later that more precision is necessary? We'd have to hunt through the code looking for the `float` keyword, and change it. This kind of widespread editing is just begging for the worst kind of bugs. When you're playing around with single vs. double precision floats, there's a ton of really subtle problems that can show up. C provides `typedef` to avoid this issue. We can do

```
typedef float real_t;

struct vec3d {
        real_t x;
        real_t y;
        real_t z;
```

[1] if you're not that up on object-oriented programming and you don't know these words, it doesn't matter for C (but you should probably put learning OOP on your todo list)

```
};
```

The syntax for `typedef` should be clear from the first line. That first line is saying "use the name `real_t` as shorthand for the type name `float`." So if we want to move up to double precision, we'd only have to change one line, and all the other code would work out of the box, no fuss, no bugs (probably). Now we know two ways to define types in C. What if we combine them?

It's pretty annoying having to always say `struct` before `vec3d`. What if we alias the type name `struct vec3d` to a more convenient name?

```
typedef float real_t;

typedef struct vec3d {
        real_t x;
        real_t y;
        real_t z;
} vec3d;
```

This means "use the name `vec3d` as shorthand for the type name `struct vec3d`."

As usual, data of this new type can be allocated on either the stack or the heap. If you want one on the stack, you can do

```
vec3d v;
```

Whereas if you want one on the heap, you can do

```
vec3d *v = malloc(sizeof(vec3d));
```

By the way, here is a potential gotcha. Check out this code:

```
// allocate a vector's worth of memory on the heap
vec3d *vptr = malloc(sizeof(vec3d));
//mutate
(*vptr).x = 0.5;
// dereference that memory
vec3d v = *vptr;
//mutate
v.x=2.5;
```

What is the value of the x field of the vector that lives *in heap memory*? Think about it before you turn the page, I'll wait.

The answer is 0.5. What the code above is doing is allocating a buffer in heap memory, and mutating it. The dereference operator then fetches that memory from the heap, and copies it into a stack-allocated variable, which is then itself mutated.

By the way, a neat piece of syntax is the `->` operator. If you had some pointer `ptr` to some struct with a field `x`, `ptr->x` is equivalent to `(*ptr).x`. You'll see that used a lot in my examples.

Memory representation of structs

This section is only really relevant if you try to do stupid things, like cast a stream of bytes to a struct. That said, sometimes the situation calls for doing stupid things, and you'll need this understanding.

So what is one of these structs, under the hood? It's exactly what I said, a bunch of variables duct taped together. You have a buffer of memory of size greater than or equal to the sum of the sizes of the fields and you put the fields in that buffer.

Did you catch the "greater than or equal to?" Why didn't I say "equal to?" Why waste memory having a buffer that's too big? There is a property of a processor architecture called *word size*. If you have a 32 bit processor, your word size is four, if you have a 64 bit processor, it's 8. This number is essentially the size of the data packets that the processor's wiring carries. Processor instruction sets generally include instructions that fetch a whole word from memory, instead of just a single byte. So say you're on a 32 bit os, and you have the following struct

```
struct poop {
        int x;
        char c;
        int y;
};
```

The sum of the sizes of the fields of this struct would be 4+1+4=9. If C implemented structs in such a way that there was only the required memory, that last int would be a problem. `x` occupies a whole word by itself, but `c` only occupies a quarter of one. That means that `y` would be part in one word, part in another. So in order to read that field, the code has to read two words of memory and do a bunch of bit logic in order to read `y`. Not great.

Instead, the compiler inserts three bytes of padding around `c` so that all field boundaries fall on word boundaries, and the code is around ten times faster. Isn't that nice?
Warning: the exact placement of the padding is unspecified. Don't get too clever.

Now back to our vector library. Head over to the examples for this chapter have a look at `vec3d.c` to see how this looks. Open up the `Makefile` and figure out what to type to compile and run it. One of the exercises is going to ask you to add a rule to one of these makefiles, so you better get your shit together. There's an appendix entry about this by the way...

OOP(ish) in C

One thing you might have noticed about that vector library. The functions all hand the form `vec_function_name` with the first argument being a pointer to a vector, followed by the rest of the arguments. This is a very common paradigm in C programming, and closely mimics the fields and methods of a `class` in an object-oriented language. Of course, you still can't have an inheritance structure or any of the other nice features of OOP, but you can at least have something sort of like member functions.[2] If you really need those kinds of abstractions, you might be better served writing your project in C++, since it has all of those abstractions built-in.

There's another piece of syntax I want to introduce before we move on. In high-level languages like Java and Python, it doesn't matter what order you write your functions/methods in; the compiler figures it out. C needs some help. If a function tries to call another function that appears later in the code, you'll get a compiler error. What you can do is **declare** your function. Essentially, you write the return type, name and type signature of the function, but instead of curly braces and an implementation, you have a semicolon. So it'll look something like this:

```
void g(void);

void f(void) {
  printf("We're in f\n");
  g();
}
```

[2] Actually, this is how C++ compilers implement (non-virtual) member functions.

```
void g(void) {
  printf("Now we're in g\n");
}
```

Now with that in mind, check out `minheap.c`. Guess what it implements. That's right, a min heap of integers! I then use it to implement good 'ol heapsort. Use it as an example for the exercises.[3]. Note the only functions declared at the top are the ones an outside caller of this code would want to call, leaving helper functions buried in the implementation. We'll come back to this idea. I swear I'll eventually explain how to organize code.

Exercises

This is where the shit hits the fan. Seriously, I'm worried about how people will do when they get here, since it's mostly a test of my skill as a teacher. It's time for you to sit down, and code some nontrivial stuff in C. Here you are likely going to experience your first segfaults. Please refer to the appendix section on debugging for help there. Good luck, have fun, don't feed.
By the way, those who have taken CS3410 at Cornell might find some of the following exercises familliar.

1. When you run my heap program, you'll find that the heap prints a message when its buffer is full and it needs to `realloc`. I said earlier that dynamic memory allocation is slow. Change the heap's interface so that no matter how large `num_elems` gets, calls to `heapsort` never has to call `realloc`.

2. arraylist. Use `void*`. It's as close as you're gonna get to polymorphism without doing some more complex stuff like tagged unions (google it)

3. hashtable. first no rehash, then rehash. No boilerplate code.

[3] I know the code looks awful. We need to learn about the preprocessor before we can properly organize the code

Chapter 4

The Preprocessor

include

define

header guards

Exercises

Chapter 5

Code Organization & Style

The Linker

Abstract data types

Exercises

Chapter 6

Appendix

6.1 Make

6.2 Debugging

6.3 Memory analysis (valgrind)